Е.В. Киргизова

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ: ОТ ТЕОРИИ К ПРАКТИКЕ

Министерство науки и высшего образования Российской Федерации
Сибирский федеральный университет

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ: ОТ ТЕОРИИ К ПРАКТИКЕ

Рекомендовано УМО РАЕ по классическому университетскому и техническому образованию в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлению 09.03.02 – «Информационные системы и технологии», направленность 09.03.02.07 «Информационно-управляющие системы» (Протокол № 903 от « 05» апреля 2021 г.)

Лесосибирск-2021

УДК 004.42 (075.8)
ББК 32.973
К43

Рецензенты:

Н.И. Пак, д-р пед. наук, профессор (Красноярский государственный педагогический институт им.В.П.Астафьева);

Т.А. Степанова, доцент, канд. пед. наук (Красноярский государственный педагогический институт им. В.П. Астафьева)

Е.В. Киргизова
К43 Технологии програмирования: от теории к практике**:** учеб. пособие / Е.В. Киргизова. – Красноярск: Сибирский федеральный университет, 2021. – 124 с.

ISBN 978-5-7638-4471-9

Пособие предлагает глубокое изложение основ современных технологий и методов программирования, соответствующее уровню знаний, необходимому для практической работы будущих специалистов в области информационных технологий. В пособии приведены общие сведения о языке C# и платформе .NET. Рассмотрены основные типы данных и преобразования между ними. Изучены основные операторы языка программирования, в том числе оператор обработки исключительных ситуаций. Изложен синтаксис описания методов и способы передачи параметров между основной подпрограммой и методом. Объяснены основные концепции объектно-ориентированного программирования: инкапсуляция, наследование, полиморфизм. Данное пособие предусматривает последовательность изучения учебного материала, которое ведется в форме, доступной пониманию студентов.

Предназначено для студентов вузов, обучающихся по направлению подготовки 09.03.02 «Информационные системы и технологии».

ISBN 978-5-7638-4471-9

УДК 004.42 (075.8)
ББК 32.973

ВВЕДЕНИЕ

Учебное пособие разработано в соответствии с требованиями, предъявляемыми Федеральным государственным образовательным стандартом высшего профессионального образования (3++), и предназначено для студентов вузов, обучающихся по направлению 09.03.02 «Информационные системы и технологии» и изучающих дисциплину «Технологии программирования».

Создание программной системы – весьма трудоёмкая задача, особенно в наше время, когда объём кода программного обеспечения превышает сотни тысяч операторов. Будущий специалист в области разработки программного обеспечения должен иметь представление о методах анализа, проектирования, реализации и тестирования программных систем, а также ориентироваться в существующих подходах и технологиях.

Структурно пособие состоит из трех глав и списка использованных источников. В первой главе описываются этапы развития технологии программирования и проблемы, возникающие при разработке сложных программных систем, рассматриваются подходы и этапы разработки программного обеспечения.

Вторая глава посвящена принципам разработки программных систем и приемам обеспечения технологичности программного обеспечения. Рассматриваются наиболее распространенные приемы и методы, используемые в таких процессах, а также возникающие в них проблемы. Даются рекомендации по организации этих процессов и по решению конкретных возникающих в них задачах.

Третья глава посвящена объектно-ориентированному программированию на языке высокого уровня C#. Описаны основные возможности платформы .NET Framework. Показана структура программ на языке C#, возможности отладки и запуска консольных приложений. Рассмотрены основные типы данных и преобразования между ними. Изучены основные операторы языка программирования, в том числе, оператор обработки исключительных ситуаций. Рассмотрены синтаксис описания методов и способы передачи параметров между основной подпрограммой и методом. Объяснены основные концепции объектно-ориентированного программирования: инкапсуляция, наследование, полиморфизм. Показано, как работать со ссылочными типами данных, обсуждена работа сборщика мусора. Изучены предопределенные платформой классы. Объяснено, что такое индексаторы и атрибуты. Отдельный параграф посвящен работе с событиями и делегатами, свойствами и перегруженными операторами.

ГЛАВА 1. ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ: ПОНЯТИЯ И ПОДХОДЫ

Программирование – сравнительно молодая и быстро развивающаяся отрасль науки и техники. Опыт ведения реальных разработок и совершенствования имеющихся программных и технических средств постоянно переосмысливается, в результате чего появляются новые методы, методологии и технологии, которые, в свою очередь, служат основой более современных средств разработки программного обеспечения. Исследовать процессы создания новых технологий и определять их основные тенденции целесообразно, сопоставляя эти технологии с уровнем развития программирования и особенностями имеющихся в распоряжении программистов программных и аппаратных средств.

1.1 Программа. Процесс обработки данных. Программное средство

Цель программирования – описание ***процесса обработки данных*** (ПОД). ***Данные*** – это представление фактов и идей в формализованном виде, пригодном для передачи и переработки в некотором ПОД. ***Информация*** – это смысл, который придаётся данным при их переработке. ***Обработка данных*** – систематическое выполнение некоторой последовательности действий с данными. Данные представляются и хранятся на носителях. Совокупность носителей данных, используемых при некоторой обработке данных, называется ***информационной средой***. Набор данных, содержащихся в какой-либо момент в информационной среде, называют ***состоянием этой среды***. Следовательно, ***ПОД можно описать как последовательность сменяющих друг друга состояний некоторой информационной среды***. Чтобы описать ПОД, необходимо определить последовательность состояний заданной информационной среды. Для того чтобы по заданному описанию ПОД порождался на ЭВМ автоматически, необходимо формализовать это описание. Такое описание называют ***программой***. Программа должна быть понятна и человеку, поэтому программы составляются на удобном, формализованном языке программирования.

Программа переводится на язык соответствующего компьютера с помощью другой программы-транслятора.

Подготовительная работа перед составлением программы:

– уточнение постановки;

– выбор метода решения;

– выяснение специфики области применения;

– выяснение общей организации программы и т.д.

Результаты подготовительной работы оформляются в виде некоторой документации. Это значительно упрощает понимание человеком программы.

Программное средство (ПС) – программа или логически связанная совокупность программ, размещённая на носителе и снабжённая

документацией.

Документация ПС позволяет понять:

− какие функции выполняет та или иная программа ПС;
− как подготовить исходные данные;
− какие программы содержит ПС;
− как запустить на выполнение;
− что означают полученные результаты;
− как её можно модифицировать.

1.2 Понятие правильности программы

Считается, что продуктом ТП является ПС, содержащее программы, выполняющие требуемые функции. Под программой часто понимают правильную программу, т.е. программу, не содержащую ошибок. Однако понятие ошибки в программе трактуется многими программистами неоднозначно. ***По мнению Майерса в программе имеется ошибка, если она не выполняет того, что разумно ожидать от неё пользователю.*** При этом разумность определяет документация. Поэтому понятие ошибки не формализовано. А так как в программном средстве программа логически связана с программной документацией, правильно говорить не об ошибке в программе, а об ошибке в ПС. Частным случаем ошибки может быть несогласованность документации с ПС. Так как задание на ПС неформально, то нельзя доказать правильность программы формальными методами. Тестирование не показывает правильность ПС, а лишь иллюстрирует наличие в нём ошибки.

1.3 Надежность программного средства

Альтернативой правильности является надёжность. ***Надёжность ПС – способность ПС безотказно выполнять определённые функции при заданных условиях в течение заданного времени и с достаточно большой вероятностью.*** Под отказом понимают наличие в ПС ошибки. Причём нужно отметить, что надёжное ПС не исключает наличия в нём ошибки. Важно, чтобы при практическом применении ПС эти ошибки проявлялись достаточно редко. Убедиться, что ПС обладает следующими свойствами, можно тестированием или практическим применением. Практически мы разрабатываем лишь надёжные, а не правильные ПС. Программные средства обладают различной степенью надёжности. Некоторые ошибки выражают лишь неудобства, а некоторые могут вызывать катастрофические последствия. Поэтому для оценки надёжности иногда используют дополнительный показатель, который учитывает стоимость или вред каждого отказа.

1.4 Технология прграммирования: основные этапы развития

Технология программирования – это совокупность методов и средств, используемых в процессе разработки программного обеспечения. Технология программирования представляет собой набор технологических инструкций, включающий:

– указание последовательности выполнения технологических операций;

– перечисление условий, при которых выполняется та или иная операция;

– описания самих операций, в которых для каждой операции определены исходные данные, результаты, а также инструкции, нормативы, стандарты, критерии и методы оценки и т.п. (рис. 1.1).

Кроме набора операций и их последовательности, технология программирования также определяет способ описания проектируемой системы, используемой на конкретном этапе разработки.

Выделяют технологии программирования, используемые на конкретных этапах разработки или для решения отдельных задач этих этапов, и технологии, охватывающие несколько этапов или весь процесс разработки. В основе первых, как правило, лежит ограниченно применимый метод, позволяющий решить конкретную задачу, в основе вторых – базовый метод или подход, определяющий совокупность методов, используемых на разных этапах разработки, или методологию.

Рис. 1.1 Структура описания технологической операции

Исторически в развитии программирования можно выделить четыре этапа принципиально отличающихся методологий (рис. 1.2).

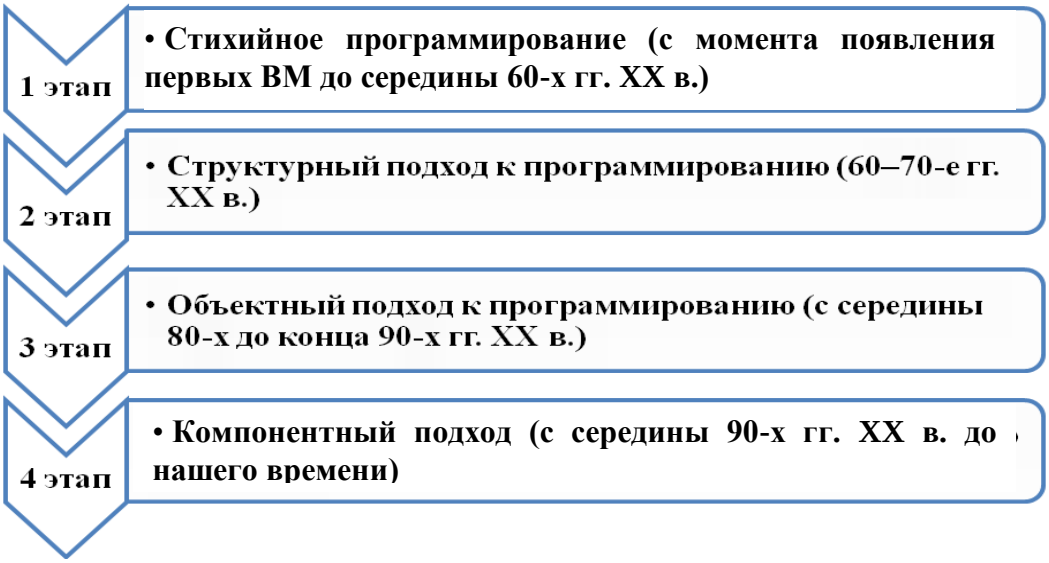


Рис. 1.2 Этапы развития программирования

Первый этап – «стихийное» программирование

Этот этап охватывает период от момента появления первых вычислительных машин до середины 60-х гг. XX в. В этот период практически отсутствовали сформулированные технологии и программирование фактически было искусством. Первые программы имели простейшую структуру. Они состояли из собственно программы на машинном языке и обрабатываемых ею данных (рис. 1.3).

Рис. 1.3 Структура первых программ

Сложность программ в машинных кодах ограничивалась способностью программиста одновременно мысленно отслеживать последовательность выполняемых операций и местонахождение данных при программировании.

Появление ассемблеров позволило вместо двоичных или 16-ричных кодов использовать символические имена данных и мнемоники кодов операций. В результате программы стали более «читаемыми».

Создание языков программирования высокого уровня, таких как FORTRAN и ALGOL, существенно упростило программирование вычислений, снизив уровень детализации операций. Это, в свою очередь, позволило увеличить сложность программ.

Революционным было появление в языках средств, дающих возможность оперировать подпрограммами (идея написания подпрограмм возникла гораздо раньше, но отсутствие средств поддержки в первых языковых средствах значительно снижало эффективность их применения). Подпрограммы можно было сохранять и использовать в других программах. В результате были

созданы огромные библиотеки расчетных и служебных подпрограмм, которые по мере надобности вызывали из разрабатываемой программы.

Типичная программа того времени состояла из основной программы, области *глобальных данных* и набора подпрограмм (в основном библиотечных), выполняющих обработку всех данных или их части (рис. 1.4).

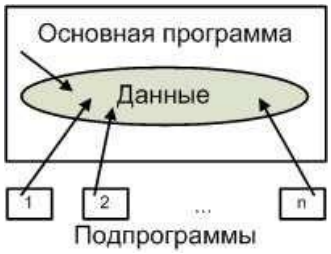


Рис. 1.4 Архитектура программы с глобальной областью данных

Слабым местом такой архитектуры было то, что при увеличении количества подпрограмм возрастала вероятность искажения части глобальных данных какой-либо подпрограммой. Например, подпрограмма поиска корней уравнения на заданном интервале по методу деления отрезка пополам меняет величину интервала. Если при выходе из подпрограммы не предусмотреть восстановления первоначального интервала, то в глобальной области окажется неверное значение интервала. Чтобы сократить количество таких ошибок, было предложено в подпрограммах размещать *локальные данные* (рис. 1.5).

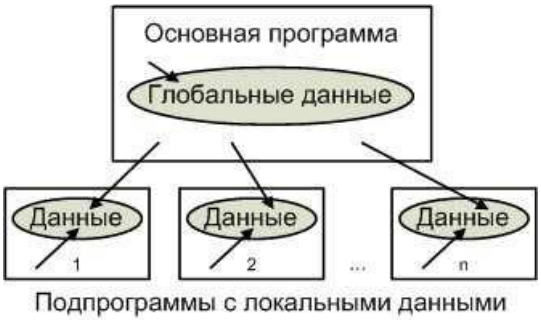


Рис. 1.5 Архитектура программы, использующей подпрограммы с локальными данными

Сложность разрабатываемого программного обеспечения при использовании подпрограмм с локальными данными по-прежнему ограничивалась возможностью программиста отслеживать процессы обработки данных, но уже на новом уровне. Однако появление средств поддержки подпрограмм позволило осуществлять разработку программного обеспечения нескольким программистам параллельно.

Вначале 60-х гг. XX в. разразился «кризис программирования». Он выражался в том, что фирмы, взявшиеся за разработку сложного программного обеспечения, такого как операционные системы, срывали все сроки завершения

проектов. Проект устаревал раньше, чем был готов к внедрению, увеличивалась его стоимость, и в результате многие проекты так никогда и не были завершены.

Объективно все это было вызвано несовершенством технологии программирования. Прежде всего стихийно использовалась разработка «снизу вверх» – подход, при котором вначале проектировали и реализовывали сравнительно простые подпрограммы, из которых затем пытались построить сложную программу. В отсутствие четких моделей описания подпрограмм и методов их проектирования создание каждой подпрограммы превращалось в непростую задачу, интерфейсы подпрограмм получались сложными, и при сборке программного продукта выявлялось большое количество ошибок согласования. Исправление таких ошибок, как правило, требовало серьезного изменения уже разработанных подпрограмм, что еще более осложняло ситуацию, так как при этом в программу часто вносились новые ошибки, которые также необходимо было исправлять. В конечном счете процесс тестирования и отладки программ занимал более 80 % времени разработки, если вообще когда-нибудь заканчивался. На повестке дня самым серьезным образом стоял вопрос разработки технологии создания сложных программных продуктов, снижающей вероятность ошибок проектирования.

Анализ причин возникновения большинства ошибок позволил сформулировать новый подход к программированию, который был назван структурным.

Второй этап – структурный подход к программированию (60–70-е гг. XX в.)

Структурный подход к программированию представляет собой совокупность рекомендуемых технологических приемов, охватывающих выполнение всех этапов разработки программного обеспечения. В основе структурного подхода лежит *декомпозиция* (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших (до 40–50 операторов) подпрограмм. С появлением других принципов декомпозиции (объектного, логического и т.д.) данный способ получил название *процедурной* декомпозиции.

В отличие от используемого ранее процедурного подхода к декомпозиции структурный подход требовал представления задачи в виде иерархии подзадач простейшей структуры. Проектирование, таким образом, осуществлялось «сверху вниз» и подразумевало реализацию общей идеи, обеспечивая проработку интерфейсов подпрограмм. Одновременно вводились ограничения на конструкции алгоритмов, рекомендовались формальные модели их описания, а также специальный метод проектирования алгоритмов – метод пошаговой детализации.

Поддержка принципов структурного программирования была заложена в основу так называемых *процедурных* языков программирования. Как правило, они включали основные «структурные» операторы передачи управления, поддерживали вложение подпрограмм, локализацию и ограничение области

«видимости» данных. Среди наиболее известных языков этой группы стоит назвать PL/1, ALGOL-68, Pascal, C.

Одновременно со структурным программированием появилось огромное количество языков, базирующихся на других концепциях, но большинство из них не выдержало конкуренции. Какие-то языки были просто забыты, идеи других были использованы в следующих версиях развиваемых языков.

Дальнейший рост сложности и размеров разрабатываемого программного обеспечения потребовал развития *структурирования данных*. Как следствие этого в языках появляется возможность определения пользовательских типов данных. Одновременно усилилось стремление разграничить доступ к глобальным данным программы, чтобы уменьшить количество ошибок, возникающих при работе с глобальными данными. В результате появилась и начала развиваться технология модульного программирования.

Модульное программирование предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные в отдельно компилируемые *модули* (библиотеки подпрограмм), например модуль графических ресурсов, модуль подпрограмм вывода на принтер (рис. 1.6).

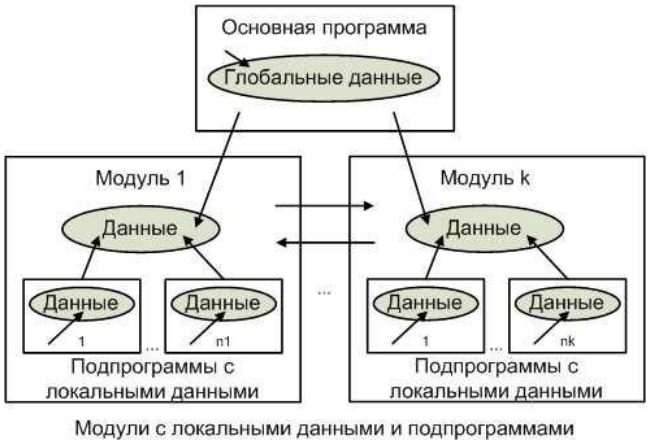


Рис. 1.6 Архитектура программы, состоящей из модулей

Связи между модулями при использовании данной технологии осуществляются через специальный интерфейс, в то время как доступ к реализации модуля (телам подпрограмм и некоторым «внутренним» переменным) запрещен. Эту технологию поддерживают современные версии языков Pascal и C (C++), языки Ада и Modula.

Использование модульного программирования существенно упростило разработку программного обеспечения несколькими программистами. Теперь каждый из них мог разрабатывать свои модули независимо, обеспечивая взаимодействие модулей через специально оговоренные межмодульные интерфейсы. Кроме того, модули в дальнейшем без изменений можно было использовать в других разработках, что повысило производительность труда программистов.

Практика показала, что структурный подход в сочетании с модульным программированием позволяет получать достаточно надежные программы,

размер которых не превышает 100 000 операторов. Узким местом модульного программирования служит то, что ошибка в интерфейсе при вызове подпрограммы выявляется только при выполнении программы (из-за раздельной компиляции модулей обнаружить эти ошибки раньше невозможно). При увеличении размера программы обычно возрастает сложность межмодульных интерфейсов, и с некоторого момента предусмотреть взаимовлияние отдельных частей программы становится практически невозможно. Для разработки программного обеспечения большого объема было предложено использовать объектный подход.

Третий этап – объектный подход к программированию (с середины 80-х до конца 90-х гг. XX в.)

Объектно-ориентированное программирование – технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности *объектов*, каждый из которых является экземпляром определенного типа (*класса*), а классы образуют иерархию с *наследованием* свойств. Взаимодействие программных объектов в такой системе осуществляется путем передачи *сообщений*.

Основным достоинством объектно-ориентированного программирования по сравнению с модульным программированием является «более естественная» декомпозиция программного обеспечения, которая существенно облегчает его разработку. Это приводит к более полной локализации данных и интегрированию их с подпрограммами обработки, что позволяет вести практически независимую разработку отдельных частей (объектов) программы. Кроме этого, объектный подход предлагает новые способы организации программ, основанные на механизмах наследования, полиморфизма, композиции, наполнения. Эти механизмы дают возможность конструировать сложные объекты из сравнительно простых. В результате существенно увеличивается показатель повторного использования кодов и появляется возможность создания библиотек классов для различных применений (рис. 1.7).

Бурное развитие технологий программирования, основанных на объектном подходе, позволило решить многие проблемы. Так были созданы среды, поддерживающие визуальное программирование, например Delphi, C++ Builder, Visual C++ и т.д. При использовании визуальной среды у программиста появляется возможность проектировать некоторую часть, например интерфейсы будущего продукта, с применением визуальных средств добавления и настройки специальных библиотечных компонентов. Результат визуального проектирования – заготовка будущей программы, в которую уже внесены соответствующие коды.

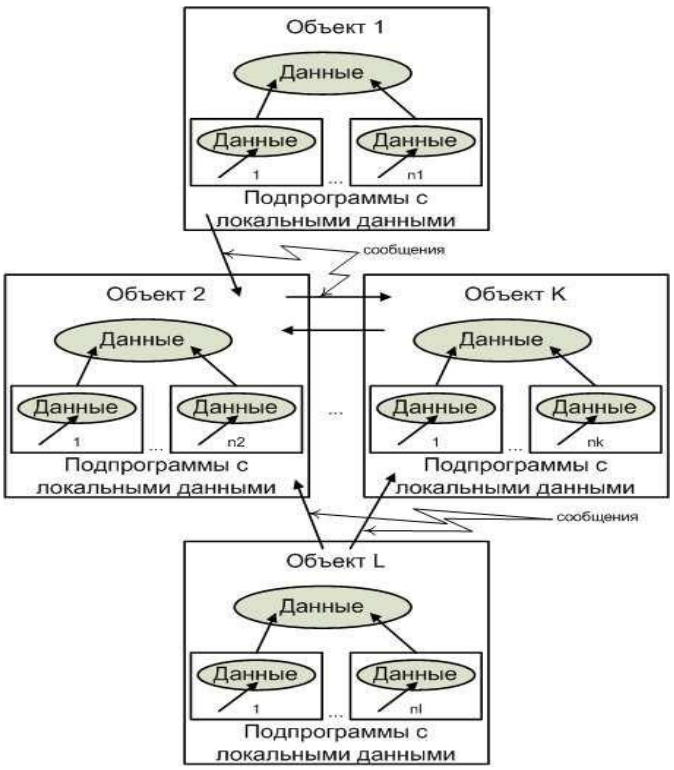


Рис. 1.7 Архитектура программ при объектно-ориентированном программировании

Можно дать обобщающее определение: *объект ООП* – это совокупность переменных состояния и связанных с ними методов (операций). Упомянутые методы устанавливают, как объект взаимодействует с окружающим миром.

Под *методами объекта* понимают процедуры и функции, объявление которых включено в описание объекта и которые выполняют действия. Возможность управлять состояниями объекта посредством вызова методов в итоге и определяет поведение объекта. Эту совокупность методов часто называют интерфейсом объекта.

Инкапсуляция – это механизм, который объединяет данные и методы, манипулирующие этими данными, и защищает и то и другое от внешнего вмешательства или неправильного использования. Когда методы и данные объединяются таким способом, создается объект.

Применяя инкапсуляцию, мы защищаем данные, принадлежащие объекту, от возможных ошибок, которые могут возникнуть при прямом доступе к этим данным. Кроме того, применение этого принципа очень часто помогает локализовать возможные ошибки в коде программы. А это намного упрощает процесс поиска и исправления этих ошибок. Можно сказать, что инкапсуляция подразумевает под собой скрытие данных, что позволяет защитить эти данные. Однако применение инкапсуляции ведет к снижению эффективности доступа к элементам объекта. Это обусловлено необходимостью вызова методов для изменения внутренних элементов (переменных) объекта. Но при современном уровне развития вычислительной техники эти потери в эффективности не играют существенной роли.

Наследование – это процесс, посредством которого один объект может наследовать свойства другого объекта и добавлять к ним черты, характерные только для него. В итоге создаётся иерархия объектных типов, где поля данных и методов «предков» автоматически являются и полями данных и методов «потомков».

Смысл и универсальность наследования заключается в том, что не надо каждый раз заново («с нуля») описывать новый объект, а можно указать «родителя» (базовый класс) и описать отличительные особенности нового класса. В результате новый объект будет обладать всеми свойствами родительского класса плюс своими собственными отличительными особенностями.

Полиморфизм – это свойство, которое позволяет одно и то же имя использовать для решения нескольких технически разных задач. Полиморфизм подразумевает такое определение методов в иерархии типов, при котором метод с одним именем может применяться к различным родственным объектам. В общем смысле концепцией полиморфизма служит идея «один интерфейс – множество методов». Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование одного интерфейса для единого класса действий. Выбор конкретного действия в зависимости от ситуации возлагается на компилятор.

Четвертый этап – компонентный подход (с середины 90-х XX в. до нашего времени)

Компонентный подход предполагает построение программного обеспечения из отдельных компонентов физически отдельно существующих частей программного обеспечения, которые взаимодействуют между собой через стандартизованные двоичные интерфейсы. В отличие от обычных объектов объекты-компоненты можно собрать в динамически вызываемые библиотеки или исполняемые файлы, распространять в двоичном виде (без исходных текстов) и использовать в любом языке программирования, поддерживающем соответствующую технологию. Это позволяет программистам создавать продукты, хотя бы частично состоящие из повторно использованных частей, т.е. применять технологию, хорошо зарекомендовавшую себя в области проектирования аппаратуры.

Компонентный подход лежит в основе технологий, разработанных на базе COM (Component Object Model – компонентная модель объектов), и технологии создания распределённых приложений CORBA (Common Object Request Broker Architecture – общая архитектура с посредником обработки запросов объектов). Эти технологии используют сходные принципы и различаются лишь особенностями их реализации.

Технология COM фирмы Microsoft является развитием технологии OLE I (Object Linking and Embedding – связывание и внедрение объектов), которая использовалась в ранних версиях Windows для создания составных документов. Технология COM определяет общую парадигму взаимодействия программ любых типов: библиотек, приложений, операционной системы, т.е. позволяет

одной части программного обеспечения использовать функции (службы), предоставляемые другой, независимо от того, функционируют ли эти части в пределах одного процесса, в разных процессах на одном компьютере или на разных компьютерах (рис. 1.8). Модификация COM, обеспечивающая передачу вызовов между компьютерами, называется DCOM (Distributed COM – распределённая COM).

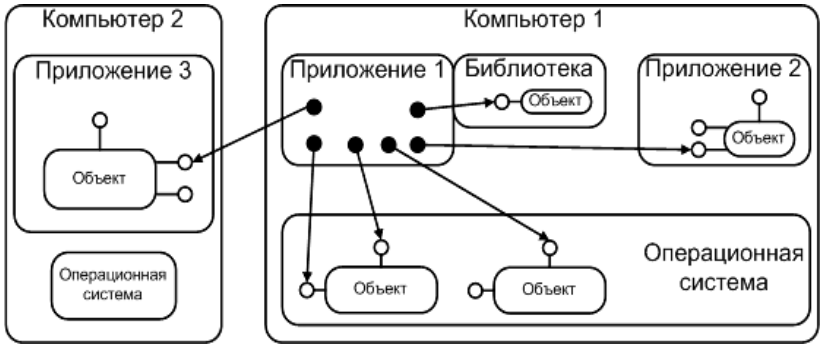


Рис. 1.8 Взаимодействие программных компонентов различных типов

По технологии COM приложение предоставляет свои службы, используя специальные объекты – *объекты* COM, которые являются экземплярами *классов* COM. Объект COM так же, как обычный объект, включает поля и методы, но в отличие от обычных объектов каждый объект COM может реализовывать несколько интерфейсов, обеспечивающих доступ к его полям и функциям. Это достигается за счет организации отдельной таблицы адресов методов для каждого интерфейса (по типу таблиц виртуальных методов). При этом интерфейс обычно объединяет несколько однотипных функций. Кроме того, классы COM поддерживают *наследование интерфейсов*, но не поддерживают *наследования реализации*, т.е. не наследуют код методов, хотя при необходимости объект класса-потомка может вызвать метод родителя.

Каждый интерфейс имеет имя, начинающееся с символа «I», и глобальный уникальный идентификатор IID (Interface IDentifier). Любой объект COM обязательно реализует интерфейс ILJnknown (на схемах этот интерфейс всегда располагают сверху). Использование этого интерфейса позволяет получить доступ к остальным интерфейсам объекта.

Объект всегда функционирует в составе *сервера* – динамической библиотеки или исполняемого файла, которые обеспечивают функционирование объекта. Различают три типа серверов:

– внутренний сервер – реализуется динамическими библиотеками, которые подключаются к приложению-клиенту и работают в одном с ними адресном пространстве, – наиболее эффективный сервер, кроме того, он не требует специальных средств;

– локальный сервер – создается отдельным процессом (модулем, exe), который работает на одном компьютере с клиентом;

– удаленный сервер – создается процессом, который работает на другом

компьютере. Например, Microsoft Word является локальным сервером. Он включает множество объектов, которые могут использоваться другими приложениями.

Для обращения к службам клиент должен получить указатель на соответствующий интерфейс. Перед первым обращением к объекту клиент посылает запрос к библиотеке COM, хранящей информацию обо всех зарегистрированных в системе классах COM объектов, и передает ей имя класса, идентификатор интерфейса и тип сервера. Библиотека запускает необходимый сервер, создает требуемые объекты и возвращает указатели на объекты и интерфейсы. Получив указатели, клиент может вызывать необходимые функции объекта.

Взаимодействие клиента и сервера обеспечивается базовыми механизмами COM или DCOM, поэтому клиенту безразлично местонахождение объекта. При использовании локальных и удаленных серверов в адресном пространстве клиента создается *proxy-объект* – заместитель объекта COM, а в адресном пространстве сервера COM – заглушка, соответствующая клиенту. Получив задание от клиента, заместитель упаковывает его параметры и, используя службы операционной системы, передает вызов заглушке. Заглушка распаковывает задание и передает его объекту COM. Результат возвращается клиенту в обратном порядке.

На базе технологии COM и её распределённой версии DCOM были разработаны компонентные технологии, решающие различные задачи разработки программного обеспечения.

OLE-automation или просто Automation (автоматизация), – технология создания программируемых приложений, обеспечивающая программируемый доступ к внутренним службам этих приложений.

ActiveX – технология, построенная на базе OLE-automation, предназначена для создания программного обеспечения как сосредоточенного на одном компьютере, так и распределённого в сети. Предполагает использование визуального программирования для создания компонентов – элементов управления ActiveX. Полученные таким образом элементы управления можно устанавливать на компьютер дистанционно с удалённого сервера, причём устанавливаемый код зависит от используемой операционной системы. Это позволяет применять элементы управления ActiveX в клиентских частях приложений Интернет. Технология CORBA, разработанная группой компаний OMC (Object Management Group – группа внедрения объектной технологии программирования), реализует подход, аналогичный COM, на базе объектов и интерфейсов CORBA. Программное ядро CORBA реализовано для всех основных аппаратных и программных платформ, и потому эту технологию можно использовать для создания распределённого программного обеспечения в гетерогенной (разнородной) вычислительной среде. Организация взаимодействия между объектами клиента и сервера в CORBA осуществляется с помощью специального посредника, названного VisiBroker, и другого специализированного программного обеспечения.

Отличительной особенностью современного этапа развития технологии программирования, кроме изменения подхода, является создание и внедрение автоматизированных технологий разработки и сопровождения программного обеспечения, которые были названы CASE-технологиями (Computer-Aided Software/System Engineering – разработка программного обеспечения/программных систем с использованием компьютерной поддержки). Без средств автоматизации разработка достаточно сложного программного обеспечения на настоящий момент становится трудно осуществимой. На сегодня существуют CASE-технологии, поддерживающие как структурный, так и объектный (в том числе и компонентный) подходы к программированию.

Появление нового подхода не означает, что отныне всё программное обеспечение будет создаваться из программных компонентов, но анализ существующих проблем разработки сложного программного обеспечения показывает, что он будет применяться достаточно широко.

Вопросы к первой главе:

1. Что такое технологии программирования?
2. Что таклое программное средство?
3. Что такое надежность программного средства?
4. Охарактеризуйте основные этапы развития технологии программирования.

ГЛАВА 2. ПРИНЦИПЫ РАЗРАБОТКИ ПРОГРАММНЫХ СИСТЕМ И ПРИЕМЫ ОБЕСПЕЧЕНИЯ ТЕХНОЛОГИЧНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Проблемы разработки сложных программных систем

Большинство современных программных систем объективно очень сложны. Эта сложность обусловливается многими причинами, главной из которых является логическая сложность решаемых ими задач.

Пока вычислительных установок было мало и их возможности были ограничены, ЭВМ применяли в очень узких областях науки и техники, причём, в первую очередь, там, где решаемые задачи были хорошо детерминированы и требовали значительных вычислений. В наше время, когда созданы мощные компьютерные сети, появилась возможность переложить на них решение сложных ресурсоёмких задач, о компьютеризации которых раньше никто и не думал. Сейчас в процесс компьютеризации вовлекаются совершенно новые предметные области, а для уже освоенных областей усложняются сложившиеся постановки задач.

Дополнительными факторами, увеличивающими сложность разработки программных систем, являются:

– сложность формального определения требований к программным системам;

– отсутствие удовлетворительных средств описания поведения дискретных систем с большим числом состояний при недетерминированной последовательности входных воздействий;

– коллективная разработка;

– необходимость увеличения степени повторяемости кодов.

Сложность определения требований к программным системам обусловливается двумя факторами. Во-первых, при определении требований необходимо учесть большое количество различных факторов. Во-вторых, разработчики программных систем не являются специалистами в автоматизируемых предметных областях, а специалисты в предметной области, как правило, не могут сформулировать проблему в нужном ракурсе.

Отсутствие удовлетворительных средств формального описания поведения дискретных систем. В процессе создания программных систем используют языки сравнительно низкого уровня. Это приводит к ранней детализации операций в процессе создания программного обеспечения и увеличивает объём описаний разрабатываемых продуктов, который, как правило, превышает сотни тысяч операторов языка программирования. Средств же, позволяющих детально описывать поведение сложных дискретных систем на более высоком уровне, чем универсальный язык программирования, не существует.

Коллективная разработка. Из-за больших объёмов проектов разработка программного обеспечения ведётся коллективом специалистов. Работая в

коллективе, отдельные специалисты должны взаимодействовать друг с другом, обеспечивая целостность проекта, что при отсутствии удовлетворительных средств описания поведения сложных систем, упоминавшемся выше, достаточно сложно. Причём чем больше коллектив разработчиков, тем сложнее организовать процесс работы.

Необходимость увеличения степени повторяемости кодов. На сложность разрабатываемого программного продукта влияет и то, что для увеличения производительности труда компании стремятся к созданию библиотек компонентов, которые можно было бы использовать в дальнейших разработках. Однако в этом случае компоненты приходится делать более универсальными, что в конечном итоге увеличивает сложность разработки.

Вместе взятые, эти факторы существенно увеличивают сложность процесса разработки. Однако очевидно, что все они напрямую связаны со сложностью объекта разработки – программной системы.

2.2 Основные подходы к разработке ПС

Разработка ПС имеет ряд специфических особенностей:

1. Разработка носит творческий характер. На каждом этапе нужно принимать обдуманные решения, производить какой-либо выбор, а не руководствоваться какой-либо последовательностью механических действий. То есть разработка ПС близка разработке или проектированию сложных устройств. Творческий характер присутствует на всех этапах разработки ПС.
2. Особенность разрабатываемого продукта. ПС представляет собой совокупность некоторых текстов (статических объектов). Смысл же (семантика) этих объектов выражается процессами обработки данных и действиями пользователей по запуску этих процессов (т.е. является динамическим). Это определяет специфический двойственный характер программных средств.
3. Продукт разработки (или ПС) при своём использовании не расходуется и не расходует используемых ресурсов.

Блочно-иерархический подход к созданию сложных систем

Подавляющее большинство сложных систем как в природе, так и в технике имеет иерархическую внутреннюю структуру. Это объясняется тем, что обычно связи элементов сложных систем различны как по типу, так и по силе, что и позволяет рассматривать эти системы как некоторую совокупность взаимозависимых подсистем. Внутренние связи элементов таких подсистем сильнее, чем связи между подсистемами. Например, компьютер состоит из процессора, памяти и внешних устройств, а Солнечная система включает Солнце и планеты, вращающиеся вокруг него.

В свою очередь, используя то же различие связей, можно каждую подсистему разделить на подсистемы и т.д. до самого нижнего «элементарного» уровня, причём выбор уровня, компоненты которого следует

считать элементарными, остаётся за исследователем. На элементарном уровне система, как правило, состоит из немногих типов подсистем, по-разному скомбинированных и организованных. Иерархии такого типа получили название «целое-часть».

Поведение системы в целом обычно оказывается сложнее поведения отдельных частей, причём из-за более сильных внутренних связей особенности системы в основном обусловлены отношениями между её частями, а не частями как таковыми.

В природе существует ещё один вид иерархии – иерархия «простое-сложное», или иерархия развития (усложнения) систем в процессе эволюции. В этой иерархии любая функционирующая система является результатом развития более простой системы. Именно данный вид иерархии реализуется механизмом наследования объектно-ориентированного программирования.

Будучи в значительной степени отражением природных и технических систем, программные системы обычно иерархические, т.е. обладают описанными выше свойствами. На этих свойствах иерархических систем строится блочно-иерархический подход к их исследованию или созданию. Этот подход предполагает сначала создавать части таких объектов (блоки, модули), а затем собирать из них сам объект.

Процесс разбиения сложного объекта на сравнительно независимые части получил название декомпозиции. При декомпозиции учитывают, что связи между отдельными частями должны быть слабее, чем связи элементов внутри частей. Кроме того, чтобы из полученных частей можно было собрать разрабатываемый объект, в процессе декомпозиции необходимо определить все виды связей частей между собой.

При создании очень сложных объектов процесс декомпозиции выполняется многократно: каждый блок, в свою очередь, декомпозируют на части, пока не получают блоки, которые сравнительно легко разработать. Данный метод разработки получил название пошаговой детализации.

Существенно и то, что в процессе декомпозиции стараются выделить аналогичные блоки, которые можно было бы разрабатывать на общей основе. Таким образом, как упоминалось выше, обеспечивают увеличение степени повторяемости кодов и, соответственно, снижение стоимости разработки.

Результат декомпозиции обычно представляют в виде схемы иерархии, на нижнем уровне которой располагают сравнительно простые блоки, а на верхнем – объект, подлежащий разработке. На каждом иерархическом уровне описание блоков выполняют с определённой степенью детализации, абстрагируясь от несущественных деталей. Следовательно, для каждого уровня используют свои формы документации и свои модели, отражающие сущность процессов, выполняемых каждым блоком. Так, для объекта в целом, как правило, удаётся сформулировать лишь самые общие требования, а блоки нижнего уровня должны быть специфицированы так, чтобы из них действительно можно было собрать работающий объект. Другими словами, чем больше блок, тем более абстрактным должно быть его описание (рис. 2.1).

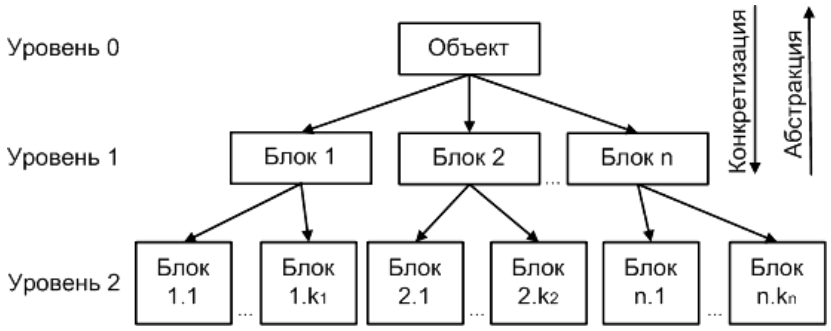


Рис. 2.1 Соотношение абстрактного и конкретного в описании блоков при блочно-иерархическом подходе

При соблюдении этого принципа разработчик сохраняет возможность осмысления проекта и, следовательно, может принимать наиболее правильные решения на каждом этапе, что называют локальной оптимизацией (в отличие от глобальной оптимизации характеристик объектов, которая для действительно сложных объектов не всегда возможна).

Итак, в основе блочно-иерархического подхода лежат декомпозиция и иерархическое упорядочение. Важную роль играют также следующие принципы:

– формализация – строгость методического подхода;

– повторяемость – необходимость выделения одинаковых блоков для удешевления и ускорения разработки;

– локальная оптимизация – оптимизация в пределах уровня иерархии;

– непротиворечивость – контроль согласованности элементов между собой;

– полнота – контроль на присутствие лишних элементов. Совокупность языков моделей, постановок задач, методов описаний некоторого иерархического уровня принято называть уровнем проектирования.

Каждый объект в процессе проектирования, как правило, приходится рассматривать с нескольких сторон. Различные взгляды на объект проектирования принято называть аспектами проектирования.

Помимо того, что использование блочно-иерархического подхода делает возможным создание сложных систем, он также:

– упрощает проверку работоспособности как системы в целом, так и отдельных блоков;

– обеспечивает возможность модернизации систем, например замены ненадёжных блоков с сохранением их интерфейсов.

Необходимо отметить, что использование блочно-иерархического подхода применительно к программным системам стало возможным только после конкретизации общих положений подхода и внесения некоторых изменений в процесс проектирования. При этом структурный подход учитывает только свойства иерархии «целое-часть», а объектный – использует ещё и свойства иерархии «простое-сложное».

2.3 Жизненный цикл ПС

В технологиях программирования *под жизненным циклом ПС понимают весь период его разработки и эксплуатации, который начинается от момента возникновения замысла ПС и кончается с прекращением всех видов его использования*. Жизненный цикл – это сложный процесс, он организован по-разному для различных ПС и на него влияет специфика коллектива разработчиков. В настоящее время выделяют пять основных подходов к организации создания и использования ПС.

1. *Водопадный подход***.** При таком подходе разработка ПС состоит из цепочки этапов. На каждом этапе этого подхода разрабатываются документы, которые используются на последующих этапах. В исходном документе – требования к ПС, в конце цепочки – программы, из которых состоит ПС.

2. *Исследовательское программирование***.** Второй подход предполагает как можно болеебыстрое создание либо реализацию рабочих версий ПС. После экспериментального применения разработанных программ проводится их модификация. Этот процесс итеративно повторяется. Такой подход был характерен для начальных этапов развития программирования и ВТ. В настоящее время такой подход применяется для разработки ПС, пользователи или заказчики которых не могут чётко сформулировать свои требования.

3. *Прототипирование.* Этот подход моделирует фазу исследовательского программирования вплоть до создания рабочих версий ПС. Эти рабочие версии используются для экспериментов с ними и для определения требований к ПС. В дальнейшем идёт разработка ПС по выработанным требованиям в рамках какого-либо другого подхода.

4. *Формальные преобразования.* Заключаются в разработке формальных спецификаций (либо требований) к ПС и дальнейшему превращению их в программы путём корректных преобразований. На этом подходе базируется компьютерная технология разработки ПС или CASE-технология.

5. *Сборочное программирование* предполагает, что ПС конструируется на основе уже имеющихся модулей. Планируется некоторое хранилище (библиотека) таких модулей, а сами они называются повторно используемыми. Процесс разработки ПС при таком подходе состоит, скорее, из сборки программ, нежели из программирования.

Состав процессов жизненного цикла регламентируется международным стандартом ISO/IEC 12207:1995 «Information Technologe – Software Life Cycle Processes» («Информационные технологии – Процессы жизненного цикла программного обеспечения»). ISO (International Organization for Standardization) – Международная организация по стандартизации. IEC (International Electrotechnical Commission) – Международная комиссия по электротехнике.

Этот стандарт описывает структуру жизненного цикла программного обеспечения и его процессы. Процесс жизненного цикла определяется как совокупность взаимосвязанных действий, преобразующих некоторые входные

данные в выходные.

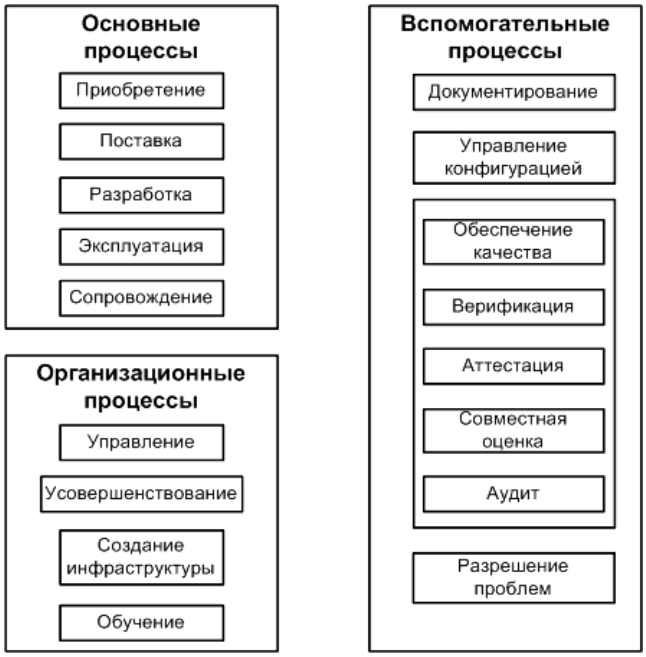


Рис. 2.2 Структура процессов жизненного цикла программного обеспечения

На рис. 2.2. представлены процессы жизненного цикла по указанному стандарту. Каждый процесс характеризуется определёнными задачами и методами их решения, а также исходными данными и результатами.

Процесс разработки в соответствии со стандартом предусматривает действия и задачи, выполняемые разработчиком, и охватывает работы по созданию программного обеспечения и его компонентов в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, а также подготовку материалов, необходимых для проверки работоспособности и соответствия качества программных продуктов, материалов, необходимых для обучения персонала, и т.д.

По стандарту процесс разработки включает следующие действия:

– подготовительную работу – выбор модели жизненного цикла, стандартов, методов и средств разработки, а также составление плана работ;

– анализ требований к системе – определение её функциональных возможностей, пользовательских требований, требований к надёжности и безопасности, требований к внешним интерфейсам и т.д.;

– проектирование архитектуры системы – определение состава необходимого оборудования, программного обеспечения и операций, выполняемых обслуживающим персоналом;

– анализ требований к программному обеспечению – определение функциональных возможностей, включая характеристики производительности, среды функционирования компонентов, внешних интерфейсов, спецификаций надёжности и безопасности, эргономических требований, требований к используемым данным, установке, приёмке, пользовательской документации, эксплуатации и сопровождению;

– проектирование архитектуры программного обеспечения – определение структуры программного обеспечения, документирование интерфейсов его компонентов, разработку предварительной версии пользовательской документации, а также требований к тестам и плана интеграции;

– детальное проектирование программного обеспечения – подробное описание компонентов программного обеспечения и интерфейсов между ними, обновление пользовательской документации, разработка и документирование требований к тестам и плана тестирования компонентов программного обеспечения, обновление плана интеграции компонентов;

– кодирование и тестирование программного обеспечения – разработку и документирование каждого компонента, а также совокупности тестовых процедур и данных для их тестирования, тестирование компонентов, обновление пользовательской документации, обновление плана интеграции программного обеспечения;

– интеграцию программного обеспечения – сборку программных компонентов в соответствии с планом интеграции и тестирование программного обеспечения на соответствие квалификационным требованиям, представляющих собой набор критериев или условий, которые необходимо выполнить, чтобы квалифицировать программный продукт как соответствующий своим спецификациям и готовый к использованию в заданных условиях эксплуатации;

– квалификационное тестирование программного обеспечения – тестирование программного обеспечения в присутствии заказчика для демонстрации его соответствия требованиям и готовности к эксплуатации; при этом проверяются также готовность и полнота технической и пользовательской документации;

– интеграцию системы – сборку всех компонентов системы, включая программное обеспечение и оборудование;

– квалификационное тестирование системы – тестирование системы на соответствие требованиям к ней и проверку оформления и полноты документации;

– установку программного обеспечения – установку программного обеспечения на оборудовании заказчика и проверку его работоспособности;

– приёмку программного обеспечения – оценку результатов квалификационного тестирования программного обеспечения и системы в целом и документирование результатов оценки совместно с заказчиком, окончательную передачу программного обеспечения заказчику.

Указанные действия можно сгруппировать, условно выделив следующие основные этапы разработки программного обеспечения (в скобках указаны соответствующие стадии разработки по ГОСТ 19.102–77 «Стадии разработки»):

– постановка задачи (стадия «Техническое задание»);

– анализ требований и разработка спецификаций (стадия «Эскизный проект»);

– проектирование (стадия «Технический проект»);

– реализация (стадия «Рабочий проект»).

Традиционно разработка также включала этап сопровождения (началу этого этапа соответствует стадия «Внедрение» по ГОСТу). Однако по международному стандарту в соответствии с изменениями, произошедшими в индустрии разработки программного обеспечения, этот процесс теперь рассматривается отдельно.

Условность выделения этапов связана с тем, что на любом этапе возможно принятие решений, которые потребуют пересмотра решений, принятых ранее.

Постановка задачи. В процессе постановки задачи чётко формулируют назначение программного обеспечения и определяют основные требования к нему. Каждое требование представляет собой описание необходимого или желаемого свойства программного обеспечения. Различают функциональные требования, определяющие функции, которые должно выполнять разрабатываемое программное обеспечение, и эксплуатационные требования, определяющие особенности его функционирования.

Требования к программному обеспечению, имеющему прототипы, обычно определяют по аналогии, учитывая структуру и характеристики уже существующего программного обеспечения. Для формулирования требований к программному обеспечению, не имеющему аналогов, иногда необходимо провести специальные исследования, называемые предпроектными. В процессе таких исследований определяют разрешимость задачи, возможно, разрабатывают методы её решения (если они новые) и устанавливают наиболее существенные характеристики разрабатываемого программного обеспечения. Для выполнения предпроектных исследований, как правило, заключают договор на выполнение научно-исследовательских работ. В любом случае этап постановки задачи заканчивается разработкой технического задания, фиксирующего принципиальные требования, и принятием основных проектных решений.

Анализ требований и определение спецификаций. Спецификациями называют точное формализованное описание функций и ограничений разрабатываемого программного обеспечения. Соответственно, различают функциональные и эксплуатационные спецификации. Совокупность спецификаций представляет собой общую логическую модель проектируемого программного обеспечения.

Для получения спецификаций выполняют анализ требований технического задания, формулируют содержательную постановку задачи, выбирают математический аппарат формализации, строят модель предметной области, определяют подзадачи и выбирают или разрабатывают методы их решения. Часть спецификаций может быть определена в процессе предпроектных исследований и, соответственно, зафиксирована в техническом задании.

На этом этапе также целесообразно сформировать тесты для поиска ошибок в проектируемом программном обеспечении, обязательно указав

ожидаемые результаты.

Проектирование. Основной задачей этого этапа является определение подробных спецификаций разрабатываемого программного обеспечения. Процесс проектирования сложного программного обеспечения обычно включает:

– проектирование общей структуры – определение основных компонентов и их взаимосвязей;

– декомпозицию компонентов и построение структурных иерархий в соответствии с рекомендациями блочно-иерархического подхода;

– проектирование компонентов.

Результатом проектирования является детальная модель разрабатываемого программного обеспечения вместе со спецификациями его компонентов всех уровней. Тип модели зависит от выбранного подхода (структурный, объектный или компонентный) и конкретной технологии проектирования. Однако в любом случае процесс проектирования охватывает как проектирование программ (подпрограмм) и определение взаимосвязей между ними, так и проектирование данных, с которыми взаимодействуют эти программы или подпрограммы.

Принято различать также два аспекта проектирования:

– логическое проектирование, которое включает те проектные операции, которые непосредственно не зависят от имеющихся технических и программных средств, составляющих среду функционирования будущего программного продукта;

– физическое проектирование – привязку к конкретным техническим и программным средствам среды функционирования, т.е. учёт ограничений, определённых в спецификациях.

Реализация. Реализация представляет собой процесс поэтапного написания кодов программы на выбранном языке программирования (кодирование), их тестирование и отладку.

Сопровождение. Сопровождение – это процесс создания и внедрения новых версий программного продукта. Причинами выпуска новых версий могут служить:

– необходимость исправления ошибок, выявленных в процессе эксплуатации предыдущих версий;

– необходимость совершенствования предыдущих версий, например улучшения интерфейса, расширения состава выполняемых функций или повышения его производительности;

– изменение среды функционирования, например, появление новых технических средств и/или программных продуктов, с которыми взаимодействует сопровождаемое программное обеспечение.

На этом этапе в программный продукт вносят необходимые изменения, которые так же, как в остальных случаях, могут потребовать пересмотра проектных решений, принятых на любом предыдущем этапе. С изменением модели жизненного цикла программного обеспечения роль этого этапа

существенно возросла, так как продукты теперь создаются итерационно: сначала выпускается сравнительно простая версия, затем следующая с большими возможностями, затем следующая и т.д. Именно это и послужило причиной выделения этапа сопровождения в отдельный процесс жизненного цикла в соответствии со стандартом ISO/IEC 12207.

2.4 Эволюция моделей жизненного цикла программного обеспечения

На протяжении последних тридцати лет в программировании сменились три модели жизненного цикла программного обеспечения: каскадная, модель с промежуточным контролем и спиральная.

Каскадная модель. Первоначально (1970 – 1985 гг.) была предложена и использовалась каскадная схема разработки программного обеспечения (рис. 2.3), которая предполагала, что переход на следующую стадию осуществляется после того, как полностью будут завершены проектные операции предыдущей стадии и получены все исходные данные для следующей стадии.

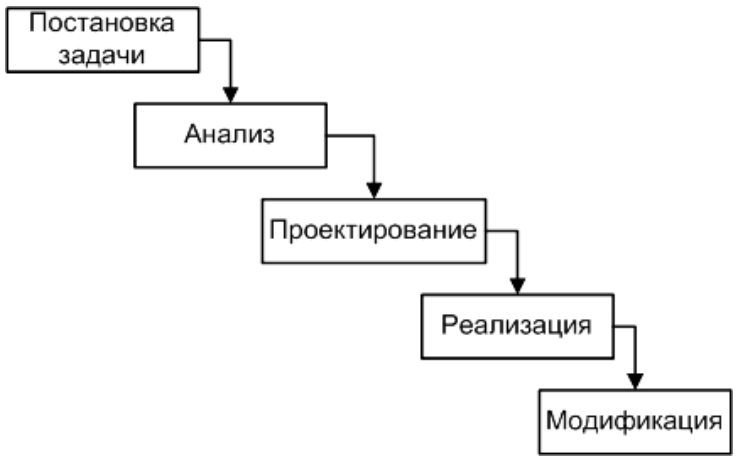


Рис. 2.3 Каскадная схема разработки программного обеспечения

Достоинствами такой схемы являются:

– получение в конце каждой стадии законченного набора проектной документации, отвечающего требованиям полноты и согласованности;

– простота планирования процесса разработки.

Именно такую схему и используют обычно при блочно-иерархическом подходе к разработке сложных технических объектов, обеспечивая очень высокие параметры эффективности разработки. Однако данная схема оказалась применимой только к созданию систем, для которых в самом начале разработки удавалось точно и полно сформулировать все требования. Это уменьшало вероятность возникновения в процессе разработки проблем, связанных с принятием неудачного решения на предыдущих стадиях. На практике такие разработки встречаются крайне редко.

В целом необходимость возвратов на предыдущие стадии обусловлена следующими причинами:

– неточные спецификации, уточнение которых в процессе разработки

может привести к необходимости пересмотра уже принятых решений;

– изменение требований заказчика непосредственно в процессе разработки;

– быстрое моральное устаревание используемых технических и программных средств;

– отсутствие удовлетворительных средств описания разработки на стадиях постановки задачи, анализа и проектирования.

Отказ от уточнения (изменения) спецификаций приведёт к тому, что законченный продукт не будет удовлетворять потребности пользователей. При отказе от учёта смены оборудования и программной среды пользователь получит морально устаревший продукт. А отказ от пересмотра неудачных проектных решений приводит к ухудшению структуры программного продукта и, соответственно, усложнит, растянет по времени и удорожит процесс его создания. Реальный процесс разработки, таким образом, носит итерационный характер.

Модель с промежуточным контролем. Схема, поддерживающая итерационный характер процесса разработки, была названа схемой с промежуточным контролем (рис. 2.4). Контроль, который выполняется по данной схеме после завершения каждого этапа, позволяет при необходимости вернуться на любой уровень и внести необходимые изменения.

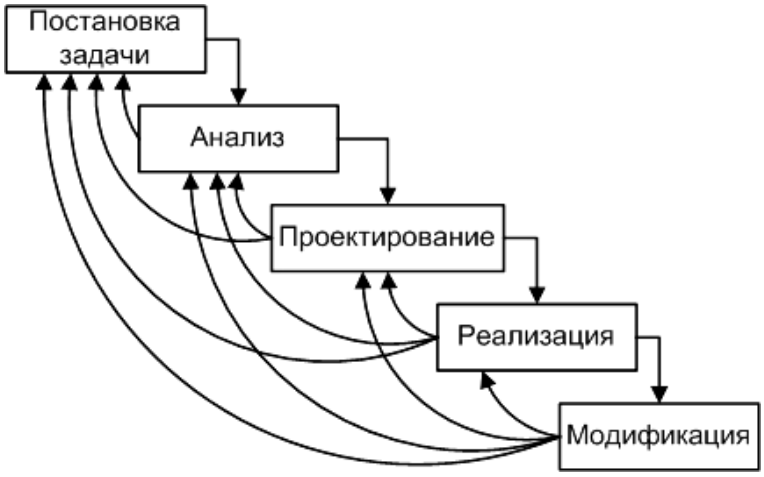


Рис. 2.4 Схема разработки программного обеспечения с промежуточным контролем

Основная опасность использования такой схемы связана с тем, что разработка никогда не будет завершена, постоянно находясь в состоянии уточнения и усовершенствования.

Спиральная модель. Для преодоления перечисленных проблем в середине 80-х гг. XX в. была предложена спиральная схема (рис. 2.5). В соответствии с данной схемой программное обеспечение создаётся не сразу, а итерационно с использованием метода прототипирования, базирующегося на

создании прототипов. Именно появление прототипирования привело к тому, что процесс модификации программного обеспечения перестал восприниматься как «необходимое зло», а стал восприниматься как отдельный важный процесс.

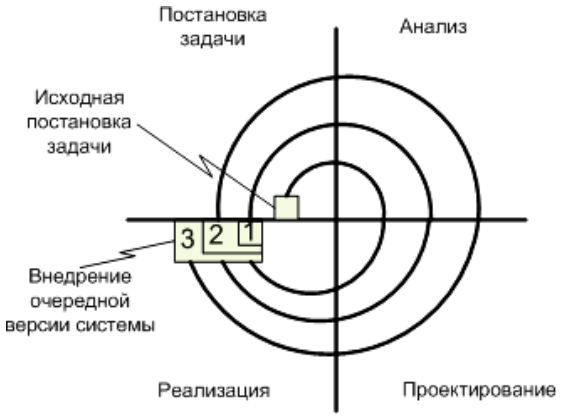


Рис. 2.5 Спиральная или итерационная схема разработки программного обеспечения

Прототипом называют действующий программный продукт, реализующий отдельные функции и внешние интерфейсы разрабатываемого программного обеспечения.

На первой итерации, как правило, специфицируют, проектируют, реализуют и тестируют интерфейс пользователя. На второй – добавляют некоторый ограниченный набор функций. На последующих этапах этот набор расширяют, наращивая возможности данного продукта.

Основным достоинством данной схемы является то, что, начиная с некоторой итерации, на которой обеспечена определённая функциональная полнота, продукт можно предоставлять пользователю, что позволяет:

– сократить время до появления первых версий программного продукта;

– заинтересовать большое количество пользователей, обеспечивая быстрое продвижение следующих версий продукта на рынке;

– ускорить формирование и уточнение спецификаций за счёт появления практики использования продукта;

– уменьшить вероятность морального устаревания системы за время разработки.

Основной проблемой использования спиральной схемы является определение моментов перехода на следующие стадии. Для её решения обычно ограничивают сроки прохождения каждой стадии, основываясь на экспертных оценках.

2.5 Оценка качества процессов создания программного обеспечения

Текущий период на рынке программного обеспечения характеризуется переходом от штучного ремесленного производства программных продуктов к их промышленному созданию. Соответственно, возросли требования к качеству разрабатываемого программного обеспечения, что требует совершенствования

процессов их разработки. На настоящий момент существует несколько стандартов, связанных с оценкой качества этих процессов, которое обеспечивает организация- разработчик. К наиболее известным относят:

– международные стандарты серии ISO 9000 (ISO 9000 – ISO 9004);

– CMM (Capability Maturity Model) – модель зрелости (совершенствования) процессов создания программного обеспечения, предложенная SEI (Software Engineering Institute – Институт программирования при университете Карнеги–Меллон);

– рабочая версия международного стандарта ISO/IEC 15504: Information Technology – Software Process Assessment; эта версия более известна под названием SPICE (Software Process Improvement and Capability dEtermination – определение возможностей и улучшение процесса создания программного обеспечения).

Единый стандарт оценки программных процессов SPICE предполагает обеспечение постоянного улучшения процессов разработки программного обеспечения и может быть применен не только к организации в целом, но и к отдельно взятым процессам. Стандарт позволяет проводить оценку проектирования программного обеспечения и при этом выявлять возможности улучшения процесса. В некоторых случаях стандарт имеет преимущество перед группой стандартов ISO 9000, поскольку предоставляет более полный набор средств по обеспечению качества и улучшению процессов.

Оба направления зарекомендовали себя как достаточно жизнеспособные, но имеющие ряд недостатков. Общие принципы управления процессами, изложенные в требованиях стандартов семейства ISO 9000, дают возможность выбрать наиболее подходящий метод обеспечения и оценки качества процесса проектирования конкретного программного продукта и способствуют развитию новых методов управления и оценки. В то же время SPICE даёт возможность существенно сократить время на «отслеживание» и оценку процессов проектирования за счёт конкретных методик, но не позволяет рассматривать процесс проектирования программного продукта и качество самого продукта как систему.

Существуют и разнообразные методы оценки программных средств, которые должны обеспечивать требования различных групп потребителей. Следует отметить, что в данном случае круг потребителей такого рода продукции несколько расширен, что связано со специфическими особенностями программного обеспечения (ПО), поскольку к «классическим» группам потребителей (государство, организация, конкретный пользователь) необходимо добавить составляющую внешней среды (например, при использовании во время функционирования ПО локальных, глобальных сетей и т.п.) и потребителей, обслуживающих ПО.

Для определения групп потребительских свойств могут быть использованы как отечественные, так и международные нормативные документы. Например, российские стандарты предлагают использовать следующие группы и комплексные показатели качества:

– показатели надёжности программных средств (ПС) (устойчивость функционирования, работоспособность);

– показатели сопровождения (структурность, простота конструкции, наглядность, повторяемость);

– показатели удобства применения (лёгкость освоения, доступность эксплуатационных программных документов, удобство эксплуатации и обслуживания);

– показатели эффективности (уровень автоматизации, временная эффективность, ресурсоёмкость);

– показатели универсальности (гибкость, мобильность, модифицируемость);

– показатели корректности (полнота реализации, согласованность, логическая корректность, проверенность).

Согласно ГОСТ Р ИСО 9126 следует обеспечивать следующие качественные характеристики программных средств:

– функциональные возможности (Functionality) – набор атрибутов, относящихся к сути набора функций и их конкретным свойствам. Функциями являются те, которые реализуют установленные или предполагаемые потребности;

– надёжность (Reliability) – набор атрибутов, относящихся к способности программного обеспечения сохранять свой уровень качества функционирования при установленных условиях за установленный период времени;

– практичность (Usability) – набор атрибутов, относящихся к объему работ, требуемых для использования и индивидуальной оценки такого использования определенным или предполагаемым кругом пользователей;

– эффективность (Efficiences) – набор атрибутов, относящихся к соотношению между уровнем качества функционирования программного обеспечения и объемом используемых ресурсов при установленных условиях;

– сопровождаемость (Maintainability) – набор атрибутов, относящихся к объему работ, требуемых для проведения конкретных изменений (модификаций);

– мобильность (Portability) – набор атрибутов, относящихся к способности программного обеспечения быть перенесенным из одного окружения в другое.

2.6 Приёмы обеспечения технологичности программного обеспечения

В условиях индустриального подхода к разработке и сопровождению программного обеспечения особый вес приобретают технологические характеристики разрабатываемых программ. Для обеспечения необходимых технологических свойств применяют специальные технологические приёмы и следуют определённым методикам, сформулированным всем предыдущим опытом создания программного обеспечения. К таким приёмам и методикам

относят правила декомпозиции, методы проектирования, программирования и контроля качества, которые под общим названием «структурный подход к программированию» были сформулированы ещё в 60-х гг. XX в. В его основу были положены следующие основные концепции:

– нисходящая разработка;

– модульное программирование;

– структурное программирование;

– сквозной структурный контроль.

Под технологичностью понимают качество проекта программного продукта, от которого зависят трудовые и материальные затраты на его реализацию и последующие модификации.

Хороший проект сравнительно быстро и легко кодируется, тестируется, отлаживается и модифицируется.

Технологичность программного обеспечения определяется проработанностью его моделей, уровнем независимости модулей, стилем программирования и степенью повторного использования кодов.

Чем лучше проработана модель разрабатываемого программного обеспечения, тем чётче определены подзадачи и структуры данных, хранящие входную, промежуточную и выходную информацию, тем проще их проектирование и реализация и меньше вероятность ошибок, для исправления которых потребуется существенно изменять программу.

Чем выше независимость модулей, тем их легче понять, реализовать, модифицировать, а также находить в них ошибки и исправлять их.

Стиль программирования, под которым понимают стиль оформления программ и их «структурность», также существенно влияет на читаемость программного кода и количество ошибок программирования.

Увеличение степени повторного использования кодов предполагает как использование ранее разработанных библиотек подпрограмм или классов, так и унификацию кодов текущей разработки. Причём для данного критерия ситуация не так однозначна, как в предыдущих случаях: если степень повторного использования кодов повышается искусственно (например, путём разработки «суперуниверсальных» процедур), то технологичность проекта может существенно снизиться.

Как следует из определения, высокая технологичность проекта особенно важна, если разрабатывается программный продукт, рассчитанный на многолетнее интенсивное использование, или необходимо обеспечить повышенные требования к его качеству.

При проектировании достаточно сложного программного обеспечения после определения его общей структуры выполняют декомпозицию компонентов в соответствии с выбранным подходом до получения элементов, которые, по мнению проектировщика, в дальнейшей декомпозиции не нуждаются.

Как упоминалось раньше, в настоящее время используют два способа декомпозиции разрабатываемого программного обеспечения, связанных с

соответствующим подходом:

– процедурный (или структурный – по названию подхода);

– объектный.

Результатом процедурной декомпозиции является иерархия подпрограмм (процедур), в которой функции, связанные с принятием решения, реализуются подпрограммами верхних уровней, а непосредственно обработка – подпрограммами нижних уровней. Это согласуется с принципом вертикального управления, который был сформулирован вместе с другими рекомендациями структурного подхода к программированию. Он также ограничивает возможные варианты передачи управления, требуя, чтобы любая подпрограмма возвращала управление той подпрограмме, которая её вызвала.

Результатом объектной декомпозиции является совокупность объектов, которые затем реализуют как переменные некоторых специально разрабатываемых типов (классов), представляющих собой совокупность полей данных и методов, работающих с этими полями.

Таким образом, при любом способе декомпозиции получают набор связанных с соответствующими данными подпрограмм, которые в процессе реализации организуют в модули.

Модули. Модулем называют автономно компилируемую программную единицу. Термин «модуль» традиционно используется в двух смыслах. Первоначально, когда размер программ был сравнительно невелик и все подпрограммы компилировались отдельно, под модулем понималась подпрограмма, т.е. последовательность связанных фрагментов программы, обращение к которой выполняется по имени. Со временем, когда размер программ значительно вырос и появилась возможность создавать библиотеки ресурсов: констант, переменных, описаний типов, классов и подпрограмм, термин «модуль» стал использоваться и в смысле автономно компилируемого набора программных ресурсов. Данные модуль может получать и/или возвращать через общие области памяти или параметры.

Первоначально к модулям (ещё понимаемым как подпрограммы) предъявлялись следующие требования:

– отдельная компиляция;

– одна точка входа;

– одна точка выхода;

– соответствие принципу вертикального управления;

– возможность вызова других модулей;

– небольшой размер (до 50 – 60 операторов языка);

– независимость от истории вызовов;

– выполнение одной функции.

Требования одной точки входа, одной точки выхода, независимости от истории вызовов и соответствия принципу вертикального управления были вызваны тем, что в то время из-за серьёзных ограничений на объём оперативной памяти программисты были вынуждены разрабатывать программы с максимально возможной повторяемостью кодов. В результате

подпрограммы, имеющие несколько точек входа и выхода, не только были обычным явлением, но и считались высоким классом программирования. Следствием же было то, что программы было очень сложно не только модифицировать, но и понять, а иногда и просто полностью отладить.

Со временем, когда основные требования структурного подхода стали поддерживаться языками программирования и под модулем стали понимать отдельно компилируемую библиотеку ресурсов, требование независимости модулей стало основным.

Практика показала, что чем выше степень независимости модулей, тем:

– легче разобраться в отдельном модуле и всей программе и, соответственно, тестировать, отлаживать и модифицировать её;

– меньше вероятность появления новых ошибок при исправлении старых или внесении изменений в программу, т.е. вероятность появления «волнового» эффекта;

– проще организовать разработку программного обеспечения группой программистов и легче его сопровождать.

Таким образом, уменьшение зависимости модулей улучшает технологичность проекта.

Степень независимости модулей (как подпрограмм, так и библиотек) оценивают двумя критериями: сцеплением и связностью.

Сцепление модулей. Сцепление является мерой взаимозависимости модулей, которая определяет, насколько хорошо модули отделены друг от друга. Модули независимы, если каждый из них не содержит о другом никакой информации. Чем больше информации о других модулях хранит модуль, тем больше он с ними сцеплен.

Различают пять типов сцепления модулей:

– по данным;

– по образцу;

– по управлению;

– по общей области данных;

– по содержимому.

Сцепление по данным предполагает, что модули обмениваются данными, представленными скалярными значениями. При небольшом количестве передаваемых параметров этот тип обеспечивает наилучшие технологические характеристики программного обеспечения.

Сцепление по образцу предполагает, что модули обмениваются данными, объединёнными в структуры. Этот тип также обеспечивает неплохие характеристики, но они хуже, чем у предыдущего типа, так как конкретные передаваемые данные «спрятаны» в структуры и потому уменьшается «прозрачность» связи между модулями. Кроме того, при изменении структуры передаваемых данных необходимо модифицировать все использующие её модули.

При сцеплении по управлению один модуль посылает другому некоторый информационный объект (флаг), предназначенный для управления внутренней

логикой модуля. Таким способом часто выполняют настройку режимов работы программного обеспечения. Подобные настройки также снижают наглядность взаимодействия модулей и потому обеспечивают ещё худшие характеристики технологичности разрабатываемого программного обеспечения по сравнению с предыдущими типами связей.

Сцепление по общей области данных предполагает, что модули работают с общей областью данных. Этот тип сцепления считается недопустимым, поскольку:

– программы, использующие данный тип сцепления, очень сложны для понимания при сопровождении программного обеспечения;

– ошибка одного модуля, приводящая к изменению общих данных, может проявиться при выполнении другого модуля, что существенно усложняет локализацию ошибок;

– при ссылке к данным в общей области модули используют конкретные имена, что уменьшает гибкость разрабатываемого программного обеспечения.

Следует иметь в виду, что «подпрограммы с памятью», действия которых зависят от истории вызовов, используют сцепление по общей области, что делает их работу в общем случае непредсказуемой. Именно этот вариант используют статические переменные C и C++.

В случае сцепления по содержимому один модуль содержит обращения к внутренним компонентам другого (передаёт управление внутрь, читает и/или изменяет внутренние данные или сами коды), что полностью противоречит блочно-иерархическому подходу. Отдельный модуль в этом случае уже не является блоком («черным ящиком»): его содержимое должно учитываться в процессе разработки другого модуля. Современные универсальные языки процедурного программирования, например Pascal, данного типа сцепления в явном виде не поддерживают, но для языков низкого уровня, например Ассемблера, такой вид сцепления остаётся возможным.

В табл. 2.1 приведены характеристики различных типов сцепления по экспертным оценкам. Допустимыми считают первые три типа сцепления, так как использование остальных приводит к резкому ухудшению технологичности программ.

Как правило, модули сцепляются между собой несколькими способами. Учитывая это, качество программного обеспечения принято определять по типу сцепления с худшими характеристиками. Так, если использовано сцепление по данным и сцепление по управлению, то определяющим считают сцепление по управлению.

Таблица 2.1

Тип сцепления	Сцепление, балл	Устойчивость к ошибкам других модулей	Наглядность (понятность)	Возможность изменения	Вероятность повторного использования
По данным	1	Хорошая	Хорошая	Хорошая	Большая
По образцу	3	Средняя	Хорошая	Средняя	Средняя
По управлению	4	Средняя	Плохая	Плохая	Малая

По общей области	6	Плохая	Плохая	Средняя	Малая
По содержимому	10	Плохая	Плохая	Плохая	Малая

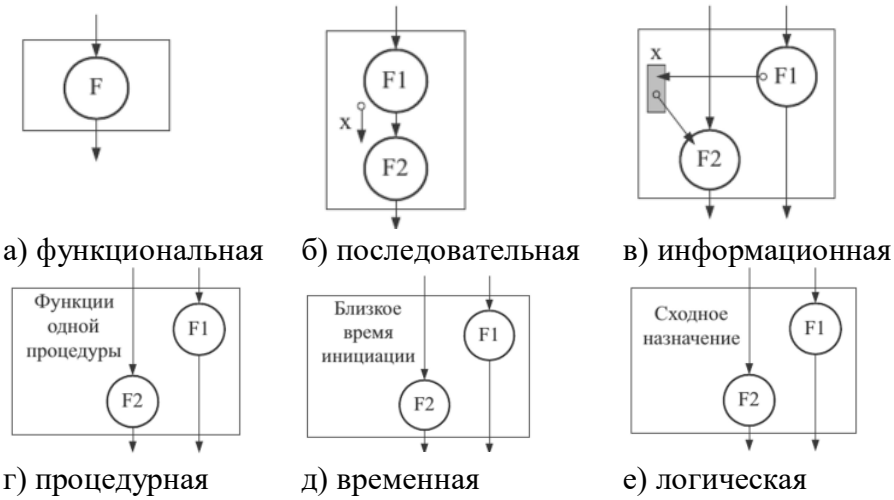
В некоторых случаях сцепление модулей можно уменьшить, удалив необязательные связи и структурировав необходимые связи. Примером может служить объектно-ориентированное программирование, в котором вместо большого количества параметров метод неявно получает адрес области (структуры), в которой расположены поля объекта, и явно дополнительные параметры. В результате модули оказываются сцепленными по образцу.

Связность модулей. Связность – мера прочности соединения функциональных и информационных объектов внутри одного модуля. Если сцепление характеризует качество отделения модулей, то связность характеризует степень взаимосвязи элементов, реализуемых одним модулем. Размещение сильно связанных элементов в одном модуле уменьшает межмодульные связи и, соответственно, взаимовлияние модулей. В то же время помещение сильно связанных элементов в разные модули не только усиливает межмодульные связи, но и усложняет понимание их взаимодействия. Объединение слабо связанных элементов также уменьшает технологичность модулей, так как такими элементами сложнее мысленно манипулировать.

Различают следующие виды связности (в порядке убывания уровня):

– функциональную;
– последовательную;
– информационную (коммуникативную);
– процедурную;
– временную;
– логическую;
– случайную.

При функциональной связности все объекты модуля предназначены для выполнения одной функции (рис. 2.6 *а*): операции, объединяемые для выполнения одной функции, или данные, связанные с одной функцией. Модуль, элементы которого связаны функционально, имеет чётко определённую цель, при его вызове выполняется одна задача, например подпрограмма поиска минимального элемента массива. Такой модуль имеет максимальную связность, следствием которой являются его хорошие технологические качества: простота тестирования, модификации и сопровождения. Именно с этим связано одно из требований структурной декомпозиции «один модуль – одна связь между модулями-библиотеками ресурсов». Например, если при проектировании текстового редактора предполагается функция редактирования, то лучше организовать модуль библиотеку функций редактирования, чем поместить часть функций в один модуль, а часть – в другой.



а) функциональная б) последовательная в) информационная

г) процедурная д) временная е) логическая

Рис. 2.6 Связность модулей

При последовательной связности функций выход одной функции служит исходными данными для другой функции (рис. 2.6, *б*). Как правило, такой модуль имеет одну точку входа, т.е. реализует одну подпрограмму, выполняющую две функции. Считают, что данные, используемые последовательными функциями, также связаны последовательно. Модуль с последовательной связностью функций можно разбить на два или более модулей как с последовательной, так и с функциональной связностью. Такой модуль выполняет несколько функций, и, следовательно, его технологичность хуже: сложнее организовать тестирование, а при выполнении модификации мысленно приходится разделять функции модуля.

Информационно связанными считают функции, обрабатывающие одни и те же данные (рис. 2.6, *в*). При использовании структурных языков программирования раздельное выполнение функций можно осуществить, только если каждая функция реализуется своей подпрограммой.

Несмотря на объединение нескольких функций, информационно связанный модуль имеет неплохие показатели технологичности. Это объясняется тем, что все функции, работающие с некоторыми данными, собраны в одно место, что позволяет при изменении формата данных корректировать лишь один модуль. Информационно связанными также считают данные, которые обрабатываются одной функцией.

Процедурно связаны функции или данные, которые являются частями одного процесса (рис. 2.6, *г*). Обычно модули с процедурной связностью функций получают, если в модуле объединены функции альтернативных частей программы. При процедурной связности отдельные элементы модуля связаны крайне слабо, так как реализуемые ими действия связаны лишь общим процессом, следовательно, технологичность данного вида связи ниже, чем предыдущего.

Временная связность функций подразумевает, что эти функции выполняются параллельно или в течение некоторого периода времени (рис. 2.6, *д*). Временная связность данных означает, что они используются в некотором временном интервале. Например, временную связность имеют функции, выполняемые при инициализации некоторого процесса. Отличительной

особенностью временной связности является то, что действия, реализуемые такими функциями, обычно могут выполняться в любом порядке. Содержание модуля с временной связностью функций имеет тенденцию меняться: в него могут включаться новые действия и/или исключаться старые. Большая вероятность модификации функции ещё больше уменьшает показатели технологичности модулей данного вида по сравнению с предыдущим.

Логическая связь базируется на объединении данных или функций в одну логическую группу (рис. 2.6, *е*). В качестве примера можно привести функции обработки текстовой информации или данные одного и того же типа. Модуль с логической связностью функций часто реализует альтернативные варианты одной операции, например сложение целых чисел и сложение вещественных чисел. Из такого модуля всегда будет вызываться одна какая-либо его часть, при этом вызывающий и вызываемый модули будут связаны по управлению. Понять логику работы модулей, содержащих логически связанные компоненты, как правило, сложнее, чем модулей, использующих временную связность, следовательно, их показатели технологичности ещё ниже.

В том случае, если связь между элементами мала или отсутствует, считают, что они имеют случайную связность. Модуль, элементы которого связаны случайно, имеет самые низкие показатели технологичности, так как элементы, объединённые в нём, вообще не связаны.

Таблица 2.2

Вид связности	Сцепление, балл	Наглядность (понятность)	Возможность изменения	Сопровождаемость
Функциональная	10	Хорошая	Хорошая	Хорошая
Последовательная	9	Хорошая	Хорошая	Хорошая
Информационная	8	Средняя	Средняя	Средняя
Процедурная	5	Средняя	Средняя	Плохая
Временная	3	Средняя	Средняя	Плохая
Логическая	1	Плохая	Плохая	Плохая
Случайная	0	Плохая	Плохая	Плохая

В трёх предпоследних случаях связь между несколькими подпрограммами в модуле обусловлена внешними причинами, а в последнем – вообще отсутствует. Это соответствующим образом проецируется на технологические характеристики модулей. В табл. 2.2 представлены характеристики различных видов связности по экспертным оценкам.

Анализ данных табл. 2.2 показывает, что на практике целесообразно использовать функциональную, последовательную и информационную связности.

Как правило, при хорошо продуманной декомпозиции модули верхних уровней иерархии имеют функциональную или последовательную связность функций и данных. Для модулей обслуживания данных характерна информационная связность функций. Данные таких модулей могут быть

связаны по-разному. Так, модули, содержащие описание классов при объектно-ориентированном подходе, характеризуются информационной связностью методов и функциональной связностью данных. Получение в процессе декомпозиции модулей с другими видами связности, скорее всего, означает недостаточно продуманное проектирование. Исключением являются лишь библиотеки ресурсов.

Библиотеки ресурсов. Различают библиотеки ресурсов двух типов: библиотеки подпрограмм и библиотеки классов.

Библиотеки подпрограмм реализуют функции, близкие по назначению, например библиотека графического вывода информации. Связность подпрограмм между собой в такой библиотеке – логическая, а связность самих подпрограмм функциональная, так как каждая из них обычно реализует одну функцию.

Библиотеки классов реализуют близкие по назначению классы. Связность элементов класса информационная, связность классов между собой может быть функциональной для родственных или ассоциированных классов и логической для остальных.

В качестве средства улучшения технологических характеристик библиотек ресурсов в настоящее время широко используют разделение тела модуля на интерфейсную часть и область реализации.

Интерфейсная часть в данном случае содержит совокупность объявлений ресурсов (заголовков подпрограмм, имен переменных, типов, классов и т.п.), которые данная библиотека предоставляет другим модулям. Ресурсы, объявление которых в интерфейсной части отсутствует, извне не доступны. Область реализации содержит тела подпрограмм и, возможно, внутренние ресурсы (подпрограммы, переменные, типы), используемые этими подпрограммами. При такой организации любые изменения реализации библиотеки, не затрагивающие её интерфейс, не требуют пересмотра модулей, связанных с библиотекой, что улучшает технологические характеристики модулей-библиотек. Кроме того, подобные библиотеки, как правило, хорошо отлажены и продуманы, так как часто используются разными программами.

Нисходящая и восходящая разработка программного обеспечения

При проектировании, реализации и тестировании компонентов структурной иерархии, полученной при декомпозиции, применяют два подхода:

– восходящий;

– нисходящий.

В литературе встречается ещё один подход, получивший название «расширение ядра». Он предполагает, что в первую очередь проектируют и разрабатывают некоторую основу – ядро программного обеспечения, например структуры данных и процедуры, связанные с ними. В дальнейшем ядро наращивают, комбинируя восходящий и нисходящий методы. На практике данный подход в зависимости от уровня ядра практически сводится либо к

нисходящему, либо к восходящему подходу.

Восходящий подход. При использовании восходящего подхода сначала проектируют и реализуют компоненты нижнего уровня, затем предыдущего и т.д. По мере завершения тестирования и отладки компонентов осуществляют их сборку, причём компоненты нижнего уровня при таком подходе часто помещают в библиотеки компонентов.

Для тестирования и отладки компонентов проектируют и реализуют специальные тестирующие программы. Подход имеет следующие недостатки:

– увеличение вероятности несогласованности компонентов вследствие неполноты спецификаций;

– наличие издержек на проектирование и реализацию тестирующих программ, которые нельзя преобразовать в компоненты;

– позднее проектирование интерфейса, а соответственно невозможность продемонстрировать его заказчику для уточнения спецификаций и т.д.

Исторически восходящий подход появился раньше, что связано с особенностью мышления программистов, которые в процессе обучения привыкают при написании небольших программ сначала детализировать компоненты нижних уровней (подпрограммы, классы). Это позволяет им лучше осознавать процессы верхних уровней. При промышленном изготовлении программного обеспечения восходящий подход в настоящее время практически не используют.

Нисходящий подход. Нисходящий подход предполагает, что проектирование и последующая реализация компонентов выполняются «сверху-вниз», т.е. вначале проектируют компоненты верхних уровней иерархии, затем следующих и так далее до самых нижних уровней. В той же последовательности выполняют и реализацию компонентов. При этом в процессе программирования компоненты нижних, ещё не реализованных уровней заменяют специально разработанными отладочными модулями – «заглушками», что позволяет тестировать и отлаживать уже реализованную часть.

При использовании нисходящего подхода применяют иерархический, операционный и комбинированный методы определения последовательности проектирования и реализации компонентов.

Иерархический метод предполагает выполнение разработки строго по уровням. Исключения допускаются при наличии зависимости по данным, т.е. если обнаруживается, что некоторый модуль использует результаты другого, то его рекомендуется программировать после этого модуля. Основной проблемой данного метода является большое количество достаточно сложных заглушек. Кроме того, при использовании данного метода основная масса модулей разрабатывается и реализуется в конце работы над проектом, что затрудняет распределение человеческих ресурсов.

Операционный метод связывает последовательность выполнения при запуске программы. Применение метода усложняется тем, что порядок выполнения модулей может зависеть от данных. Кроме того, модули вывода

результатов, несмотря на то, что они вызываются последними, должны разрабатываться одними из первых, чтобы не проектировать сложную заглушку, обеспечивающую вывод результатов при тестировании. С точки зрения распределения человеческих ресурсов сложным является начало работ, пока не закончены все модули, находящиеся на так называемом критическом пути.

Комбинированный метод учитывает такие факторы, влияющие на последовательность разработки:

– достижимость модуля – наличие всех модулей в цепочке вызова данного модуля;

– зависимость по данным – модули, формирующие некоторые данные, должны создаваться раньше обрабатывающих;

– обеспечение возможности выдачи результатов – модули вывода результатов должны создаваться раньше обрабатывающих;

– готовность вспомогательных модулей – вспомогательные модули, например модули закрытия файлов, завершения программы, должны создаваться раньше обрабатывающих;

– наличие необходимых ресурсов.

Кроме того, при прочих равных условиях сложные модули должны разрабатываться прежде простых, так как при их проектировании могут выявиться неточности в спецификациях, а чем раньше это произойдет, тем лучше.

Нисходящий подход допускает нарушение нисходящей последовательности разработки компонентов в специально оговоренных слчаях. Так, если некоторый компонент нижнего уровня используется многими компонентами более высоких уровней, то его рекомендуют проектировать и разрабатывать раньше, чем вызывающие его компоненты. И, наконец, в первую очередь проектируют и реализуют компоненты, обеспечивающие обработку правильных данных, оставляя компоненты обработки неправильных данных напоследок.

Нисходящий подход обычно используют и при объектно-ориентированном программировании. В соответствии с рекомендациями подхода вначале проектируют и реализуют пользовательский интерфейс программного обеспечения, затем разрабатывают классы некоторых базовых объектов предметной области, а уже потом, используя эти объекты, проектируют и реализуют остальные компоненты. Нисходящий подход обеспечивает:

– максимально полное определение спецификаций проектируемого компонента и согласованность компонентов между собой;

– раннее определение интерфейса пользователя, демонстрация которого заказчику позволяет уточнить требования к создаваемому программному обеспечению:

– возможность нисходящего тестирования и комплексной отладки.

Вопросы ко второй главе:

1. Охарактеризуйте основные проблемы разработки сложных программных систем.
2. Перечислите и охарактеризуйте основные подходы к разработке ПС.
3. Что такое жизненный цикл ПС?
4. Перечислите и охарактеризуйте основные этапы разработки программного обеспечения.
5. Охарактеризуйте модели жизненного цикла программного обеспечения.
6. Что такое внешнее описание ПС?
7. Что такое сопровождение ПС?
8. Что такое качество ПС?
9. Перечислите и охарактеризуйте приёмы обеспечения технологичности программного обеспечения.

ГЛАВА 3. ЯЗЫК ПРОГРАММИРОВАНИЯ C#

3.1 Обзор платформы MS.NET

Microsoft®.NET Framework – платформа для создания windows приложений и распределенных web-приложений. .NET Framework содержит классы, интерфейсы и типы данных, которые облегчают и оптимизируют процесс разработки, а также обеспечивают доступ к функциям системы. Для упрощения взаимодействия между языками большинство типов платформы .NET Framework являются CLS-совместимыми, и поэтому их можно использовать в любом языке программирования, компилятор которого соответствует спецификации CLS.

Инструменты платформы .NET Framework представляют собой основу для создания элементов управления, компонентов и приложений .NET. .NET Framework содержит инструменты, предназначенные для следующих задач:

– представление базовых типов данных и исключений;

– инкапсуляция структур данных;

– операции ввода-вывода;

– доступ к сведениям о загруженных типах;

– вызов проверок безопасности .NET Framework;

– доступ к данным, предоставление графического пользовательского интерфейса на стороне клиента и управляемого сервером графического пользовательского интерфейса на стороне клиента.

.NET Framework предлагает расширенный набор интерфейсов, а также абстрактных и конкретных (неабстрактных) классов. Можно использовать существующие конкретные классы, кроме того, во многих случаях на их основе можно создавать собственные производные классы.

Приложению .NET через предопределенные классы в .NET Framework доступны все сервисы Windows: IIS, WMI, Component services, Message Queuing. Перечислим основные компоненты платформы .NET Framework:

– Common Language Runtime (CLR) – общеязыковая среда выполнения – упрощает разработку приложений, поддерживает различные языки, обеспечивает безопасное выполнение кода, контролирует ресурсы.

– Библиотека классов – содержит большое количество классов с возможностью расширения под нужды разработчика.

– ADO.NET – обеспечивает поддержку работы с базами данных и XML.

– ASP.NET – инструмент для разработки web-приложений.

– XML Web-сервисы – программируемые web компоненты, которые доступны различным приложениям.

Платформа MS.NET предоставляет следующие возможности:

– Поддержка всех web стандартов, таких как HTML, XML, SOAP, XSLT, XML Path Language и других. Создание и поддержка XML Web сервисов.

– Одинаковая среда для всех языков программирования.

– Легкое использование при разработке, все классы собраны в

иерархические пространства имен, поддержка общей системы типов.

– Легко расширяемые классы – иерархия классов не скрыта от разработчика. Разработчик может разрабатывать свои собственные классы на основе имеющихся.

Общеязыковая среда выполнения

Общеязыковая среда выполнения упрощает разработку приложений, обеспечивает правильное и безопасное выполнение, поддерживает несколько языков программирования и следит за ресурсами. Эта среда также называется управляемой средой, она управляет памятью, потоками, организует удаленное взаимодействие. В среде CLR автоматически работает ряд сервисов:

– Загрузчик классов – управляет метаданными, загружает и располагает классы в памяти.

– Перевод из промежуточного языка MSIL в машинные коды во время выполнении (Just-in-time).

– Сборщик мусора – следит за неиспользуемыми объектами. При выходе из области видимости объектов и недостатке памяти сборщик очищает их.

– Ядро отладки – позволяет отлаживать и трассировать код.

– Проверка типов – не допускает опасных преобразований типов.

– Обработка исключений – структурированная система классов исключений, интегрированная с Windows.

– Поддержка COM-объектов.

– Поддержка базовых классов интегрированных в среду.

– Служба безопасности.

Библиотека классов .NET Framework увеличивает мощность среды выполнения и обеспечивает высокоуровневые сервисы, необходимые при программировании. Библиотека классов разбита на иерархию пространств имен. Представим некоторые из них:

Пространство имен System содержит фундаментальные классы и определяет наиболее часто используемые типы данных, события, интерфейсы, атрибуты и исключения. Также классы преобразования данных, манипуляций с параметрами System.Collections содержат списки, хэш-таблицы и другие виды группировки данных.

ADO.NET – следующее поколение технологии ADO, обеспечивает расширенную поддержку отсоединенной программной модели БД, поддерживает работу с XML. Классы для работы с ADO.NET находятся в System.Data, с XML – в пространстве имен System.XML.

ASP.NET – программная структура, основанная на среде выполнения, работающей на сервере, которая позволяет создавать мощные Web-приложения. Классы расположены в System.Web. Обеспечивается поддержка XML Web сервисов, необходимых для распределенной разработки.

Пространство имен System.Windows.Forms используется для создания windows интерфейса. В нем содержится большое количество функций, ранее доступных только в API.

Пространство имен System.Drawing обеспечивает доступ к графике GDI+.

Языки MS.NET Framework

MS.NET Framework поддерживает множество языков программирования, всего порядка двадцати, рассмотрим некоторые из них:

C# разработан для платформы .NET, это первый современный объектно-ориентированный язык из семейства C и C++. Его основные свойства – это классы, интерфейсы, делегаты, пространства имен, свойства, индексаторы и т.д. Нет необходимости в заголовочном файле.

Расширение управляемого C++ – минимальное расширение C++. Обеспечен доступ к возможностям платформы .NET Framework, включая сборщик мусора, наследование только от одного класса и т.д.

Visual Basic .NET улучшен по сравнению с предыдущей версией, добавлены наследование, конструкторы, полиморфизм, перегрузка конструкторов и т.д.

JScript.NET полностью переписан для поддержки среды .NET, включает поддержку классов, наследования, типов и компиляции.

Visual J#.NET – инструмент разработки для Java-программистов, желающих создавать приложения и сервисы в среде .NET. Имеется возможность автоматически обновлять существующие проекты Visual J++ 6.0 в решения Visual Studio .NET.

Сторонние языки – некоторые языки программирования также поддерживаются платформой .NET: APL, COBOL, Pascal, Oberon, Perl, Python и SmallTalk.

Вопросы к п. 3.1

1. Что такое платформа MS.NET? Каковы её преимущества?
2. Перечислите основные понятия платформы .NET.
3. Охарактеризуйте компоненты MS.NET Framework.
4. Что такое общеязыковая среда выполнения? Какие ее основные сервисы?
5. Перечислите основные классы библиотеки .NET Framework.

3.2 Основы языка C#

Рассмотрим основную структуру программ на языке C#. Рассмотрим класс Console для выполнения простых операций ввода и вывода.

Структура программы на C#

При изучении любого языка первая программа, которую обычно пишут, – «Hello, World». Рассмотрим, как она выглядит в C#. using System;

```
class Hello
{
        public static void Main()
        {
                Console.WriteLine(“Hello, World”);
```

```
    }
}
```

При выполнении приведенного кода на экране появится надпись «Hello, World».

В C# приложение – это коллекция одного или нескольких классов, структур и других типов. Класс определяется как набор данных и методов, работающих с ними.

При рассмотрении кода приложения «Hello, World» видим, что есть единственный класс, названный Hello. После имени класса открываем фигурные скобки ({). Всё, что находится до соответствующей закрывающейся скобки (}), является частью класса.

Можно распределить один класс приложения C# на несколько файлов.

Также, с другой стороны, можно поместить несколько классов в один файл.

Каждое приложение должно начинаться с какого-то оператора. В C# при запуске приложения начинает выполняться метод Main. Несмотря на то, что в приложении может быть много классов, точка входа всегда одна. Может быть несколько методов Main, но запускаться будет только один, он выбирается при компиляции. Синтаксис Main важен, если проект создан в Visual Studio, то он сгенерируется автоматически. При завершении метода Main приложение заканчивает работу.

Как часть MS.NET Framework C# содержит много различных классов, которые упорядочены в пространствах имен. Пространство имен (namespace) – это набор связанных классов. Пространство имен также может содержать вложенные пространства имен. Основное пространство имен – это System. К объектам пространств имен можно обращаться при помощи полного имени, используя префиксы.

Например, пространство System содержит класс Console, который выполняет несколько методов, включая WriteLine System.Console.WriteLine(“Hello, World”);

Директива using позволяет обращаться к членам пространства имен напрямую без использования полного имени.

using System; Console.WriteLine(“Hello, World”);

Основные операции ввода/вывода

Рассмотрим класс Console и его методы ввода и вывода. Класс Console обеспечивает приложению C# доступ к стандартным потокам ввода, вывода и ошибок. Стандартный поток ввода – клавиатура. Стандартный поток вывода и ошибок – экран. Эти потоки могут быть перенаправлены из/в файлы.

Write и WriteLine методы вывода информации на консоль.

Они перегружаемы, т.е. могут выводить как строки, так и числа.

```
Console.WriteLine(99);
Console.WriteLine("Hello, World");
Console.WriteLine("The sum of {0} and {1} is {2}", 100, 130, 100+130);
```

Можно использовать левое и правое выравнивания, задавать ширину вывода.

```
Console.WriteLine("\"Левое выравн. в поле ширины 10: {0, -10}\"", 99);
Console.WriteLine("\"Правое выравн. в поле ширины 10: {0,10}\"", 99);
```

Будет выведено

“Левое выравн. в поле ширины 10: 99 ”

“Правое выравн. в поле ширины 10: 99”

Есть настройки для вывода числовых форматов:

```
Console.WriteLine("Валюта - {0:C} {1:C4}", 88.8, -888.8);
Console.WriteLine("Целое число - {0:D5}", 88);
Console.WriteLine("Экспонента - {0:E}", 888.8);
Console.WriteLine("Фиксированная точка - {0:F3}", 888.8888);
Console.WriteLine("Общий формат- {0:G}", 888.8888);
Console.WriteLine("Числовой формат - {0:N}", 8888888.8);
Console.WriteLine("Шестнадцатиричный - {0:X4}", 88);
```

Соответственно отобразят:

Валюта - $88.80 ($888.8000)

Целое число - 00088

Экспонента - 8.888000E+002

Фиксированная точка - 888.889

Общий формат - 888.8888

Числовой формат - 8,888,888.80

Шестнадцатеричный формат – 0058

Рекомендации по оформлению кода на языке C#

Основная рекомендация – это комментирование кода. Комментирование программы позволяет разработчикам, не участвовавшим при написании, разобраться, как работает приложение. Нужно использовать полные и значимые имена для именования компонент. Хорошие комментарии не объясняют что написано в программе, а отвечают на вопрос, почему именно так. Если в вашей организации есть стандарты комментирования, придерживайтесь их.

C# поддерживает несколько вариантов комментирования кода: однострочные(//), многострочные комментарии(/* */) и XML-документация(///). В XML-документации можно использовать различные предопределенные теги. Для генерации XML файла с комментариями используется дополнительный параметр компиляции. Для компиляции кода из командной строки:

```
csc myprogram.cs /doc:mycomments.xml
```

Надежная программа на C# обязана обрабатывать непредвиденные ситуации. Независимо от того, сколько различных ошибок предусмотрено, всегда существует вероятность того, что что-то может пойти не так. Когда происходит ошибка при выполнении приложения, операционная система генерирует исключение. Конструкцией try-catch можно перехватывать эти

исключения. Если при выполнении программы в блоке try произошло исключение, управление передаётся блоку catch. Если в программе не обрабатывается исключение, то оно вызовет окно операционной системы с предложением отладить программу при помощи Just-in-Time Debugging.

Перед запуском приложения C# необходимо его откомпилировать. Компилятор преобразует исходный код в машинные коды. Компилятор C# можно вызывать как из командной строки, так и из Visual Studio. Пример вызова компилятора из командной строки:

csc Hello.cs /debug+ /out:Greet.exe

(Заметим, что csc – это файл, путь к нему нужно прописывать полностью.)

Если компилятор находит ошибки, он сообщает строку ошибки и номер символа. Если ошибок нет и приложение откомпилировалось, то его можно запускать как при помощи Visual Studio, так и в командной строке по имени файла с расширением exe.

Вопросы к п. 3.2

1. Откуда начинается выполнение приложения C#?
2. Когда приложение заканчивает работу?
3. Сколько классов может содержать приложение C#?
4. Сколько методов Main может содержать приложение?
5. Как прочитать данные, введенные пользователем с клавиатуры?
6. В каком пространстве имен находится класс Console?
7. Что произойдёт при необработанном в приложении исключении?

Задания для самостоятельной работы

Задания на создание C# программ, компилирование и отладку, использование отладчика Visual Studio, обработку исключительных ситуаций.

1. Написать программу, которая спрашивает имя пользователя и затем приветствует пользователя по имени. (Создать консольное приложение.)

– Откомпилировать и запустить программу с помощью Visual Studio.

– Откомпилировать и запустить программу с командной строки.

– В среде Visual Studio использовать пошаговую отладку. Посмотреть, как изменяется текущее значение переменной.

2. Написать программу, которой на вход подается два целых числа, на выходе – результат деления одного числа на другое. Предусмотреть обработку исключительной ситуации, возникающей при делении числа на ноль.

3.3 Использование структурных переменных

Все приложения работают с теми или иными данными. Разработчику C# необходимо понимать, как хранятся и обрабатываются данные в приложении. Обычно данные хранятся в переменных, которые необходимо объявить до их использования. При объявлении переменной под неё резервируется некоторое

количество памяти в зависимости от типа переменной и объявляется имя. После объявления переменной можно присваивать ей значения.

Рассмотрим, как использовать структурные переменные в C#, как именовать переменные в соответствии со стандартами, а также как присваивать значения и переводить существующие переменные из одного типа в другой.

Общая система типов (Common Type System)

Каждая переменная имеет тип данных, который определяет какие значения хранятся в ней. C# – язык безопасных типов, т.е. компилятор гарантирует, что значение хранящееся в переменной будет всегда соответствующего типа.

CTS (Common Type System) – интегрированная часть общеязыковой среды выполнения. Эта модель определяет правила, которыми руководствуется среда выполнения при объявлении, использовании и управления типами. CTS обеспечивает структуру, необходимую для многоязыковой интеграции, безопасности типов и высокой эффективности выполнения кода.

Рассмотрим два типа переменных: структурные и ссылочные. Структурные типы данных напрямую содержат данные. Каждая переменная содержит свою копию данных, т.е. при операции над одной переменной невозможно изменить другую. Структурные типы данных включают в себя встроенные и пользовательские типы. Разница между ними в C# минимальна, так как они используются одинаково. Все структурные типы напрямую содержат данные и не могут быть null.

Ссылочные типы данных содержат ссылки на данные. Две переменные могут указывать на один и тот же объект, т.е. при изменении одной ссылочной переменной можно изменить другую.

Все базовые типы данных содержатся в пространстве имен System. Все типы наследуются от System.Object. Все структурные типы наследуются от System.ValueType.

Встроенные типы объявляются при помощи зарезервированных слов, кроме того, можно объявить при помощи типа структуры из пространства имен System:

sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

Тип данных определяет внутреннее представление данных, множество значений, которые может принимать объект, а также допустимые действия, которые можно применять над объектом.

В языке C# есть следующие примитивные типы данных:

bool: хранит значение true или false (логические литералы). Представлен системным типом System.Boolean

```
bool alive = true;
bool isDead = false;
```

byte: хранит целое число от 0 до 255 и занимает 1 байт. Представлен системным типом System.Byte

```
byte bit1 = 1;
byte bit2 = 102;
```

sbyte: хранит целое число от -128 до 127 и занимает 1 байт. Представлен системным типом System.SByte

```
sbyte bit1 = -101;
sbyte bit2 = 102;
```

short: хранит целое число от -32768 до 32767 и занимает 2 байта. Представлен системным типом System.Int16

```
short n1 = 1;
short n2 = 102;
```

ushort: хранит целое число от 0 до 65535 и занимает 2 байта. Представлен системным типом System.UInt16

```
ushort n1 = 1;
ushort n2 = 102;
```

int: хранит целое число от -2147483648 до 2147483647 и занимает 4 байта. Представлен системным типом System.Int32. Все целочисленные литералы по умолчанию представляют значения типа int:

```
int a = 10;
int b = 0b101;  // бинарная форма b =5
int c = 0xFF;   // шестнадцатеричная форма c = 255
```

uint: хранит целое число от 0 до 4294967295 и занимает 4 байта. Представлен системным типом System.UInt32

```
uint a = 10;
uint b = 0b101;
uint c = 0xFF;
```

long: хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт. Представлен системным типом System.Int64

```
long a = -10;
long b = 0b101;
long c = 0xFF;
```

ulong: хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт. Представлен системным типом System.UInt64

```
ulong a = 10;
```

ulong b = 0b101;
ulong c = 0xFF;

float: хранит число с плавающей точкой от -3.4*1038 до 3.4*1038 и занимает 4 байта. Представлен системным типом System.Single

double: хранит число с плавающей точкой от ±5.0*10-324 до ±1.7*10308 и занимает 8 байта. Представлен системным типом System.Double

decimal: хранит десятичное дробное число. Если употребляется без десятичной запятой, имеет значение от ±1.0*10-28 до ±7.9228*1028, может хранить 28 знаков после запятой и занимает 16 байт. Представлен системным типом System.Decimal

char: хранит одиночный символ в кодировке Unicode и занимает 2 байта. Представлен системным типом System.Char. Этому типу соответствуют символьные литералы:

char a = 'A';
char b = '\x5A';
char c = '\u0420';

Правила именования переменных

При именовании переменных необходимо соблюдать следующие правила (если не соблюдать, то получим ошибку при компиляции):

– можно использовать только буквы, нижнее подчеркивание и цифры,

– имя переменной начинается только с буквы или подчеркивания,

– после первого символа можно использовать и цифры,

– нельзя использовать зарезервированные слова.

Рекомендации по именованию:

– избегать имена только из заглавных букв,

– избегать начинать имя с подчеркивания,

– избегать аббревиатур,

– именование PascalCasing – каждое слово начинается с большой буквы, используется для классов, методов, свойств, перечислений, интерфейсов, констант, пространств имен,

– именование camelCasing – каждое слово, кроме первого, начинается с большой буквы, используется для переменных, полей и параметров.

Использование встроенных типов данных

Для использования переменной необходимо выбрать ей имя, назначить тип и присвоить начальное значение.

Переменные, которые объявлены в методах, называются локальными. В C# нельзя использовать неинициализированные переменные, в этом случае выдаётся ошибка при компиляции. Переменной можно присвоить значение соответствующего типа.

Добавление значения переменной осуществляется очень просто:

int itemCount;
itemCount = 2;

itemCount = itemCount + 40;

Приведем сокращенные способы записи арифметический действий:

```
var += expression;          // var = var + expression
var -= expression;          // var = var - expression
var *= expression;          // var = var * expression
var /= expression;          // var = var / expression
var %= expression;          // var = var % expression
```

Выражения (expression) состоят из операций и операндов. Существуют следующие общие операции:

Операции присваивания –присваивают значение правого операнда левому: = *= /= %= += -= <<= >>= &= ^= |= .

Как и во многих других языках программирования, в C# имеется базовая операция присваивания =, которая присвоивает значение правого операнда левому операнду:

```
int number = 23;
```

Здесь переменной number присваивается число 23. Переменная number представляет левый операнд, которому присваивается значение правого операнда, то есть числа 23.

Также можно выполнять множественно присвоение сразу нескольким переменным одновременно:

```
int a, b, c;
a = b = c = 34;
```

Стоит отметить, что операции присвоения имеют низкий приоритет. И вначале будет вычисляться значение правого операнда, только потом будет идти присвоение этого значения левому операнду. Например:

```
int a, b, c;
a = b = c = 34 * 2 / 4; // 17
```

Сначала будет вычисляться выражение 34 * 2 / 4, затем полученное значение будет присвоено переменным.

Кроме базовой операции присвоения в C# есть еще ряд операций:

+=: присваивание после сложения. Присваивает левому операнду сумму левого и правого операндов: выражение A += B равнозначно выражению A = A + B;

-=: присваивание после вычитания. Присваивает левому операнду разность левого и правого операндов: A -= B эквивалентно A = A - B;

*=: присваивание после умножения. Присваивает левому операнду произведение левого и правого операндов: A *= B эквивалентно A = A * B;

/=: присваивание после деления. Присваивает левому операнду частное левого и правого операндов: A /= B эквивалентно A = A / B;

%=: присваивание после деления по модулю. Присваивает левому операнду остаток от целочисленного деления левого операнда на правый: A %= B эквивалентно A = A % B;

&=: присваивание после поразрядной конъюнкции. Присваивает левому операнду результат поразрядной конъюнкции его битового представления с битовым представлением правого операнда: A &= B эквивалентно A = A & B;

|=: присваивание после поразрядной дизъюнкции. Присваивает левому операнду результат поразрядной дизъюнкции его битового представления с битовым представлением правого операнда: A |= B эквивалентно A = A | B;

^=: присваивание после операции исключающего ИЛИ. Присваивает левому операнду результат операции исключающего ИЛИ его битового представления с битовым представлением правого операнда: A ^= B эквивалентно A = A ^ B;

<<=: присваивание после сдвига разрядов влево. Присваивает левому операнду результат сдвига его битового представления влево на определенное количество разрядов, равное значению правого операнда: A <<= B эквивалентно A = A << B;

>>=: присваивание после сдвига разрядов вправо. Присваивает левому операнду результат сдвига его битового представления вправо на определенное количество разрядов, равное значению правого операнда: A >>= B эквивалентно A = A >> B.

Применение операций присвоения:

```
int a = 10;
a += 10;       // 20
a -= 4;        // 16
a *= 2;        // 32
a /= 8;        // 4
a <<= 4;       // 64
a >>= 2;       // 16
```

Операции сравнения – сравнивают два значения: == != .

Логические – производят побитовые операции над операндами:

<< >> & ^ | .

Условные – выполняют одно выражение из двух в зависимости от булевого условия: && || ?: .

Операция инкремента:

Инкремент бывает префиксным: ++x - сначала значение переменной x увеличивается на 1, а потом ее значение возвращается в качестве результата операции.

И также существует постфиксный инкремент: x++ - сначала значение переменной x возвращается в качестве результата операции, а затем к нему прибавляется 1.

Операция декремента или уменьшения значения на единицу. Также существует префиксная форма декремента (--x) и постфиксная (x--).

Арифметические – выполняют стандартные арифметические операции:

+ (сложения), – (вычитания), * (умножения), / (деления), % (операция получение остатка от целочисленного деления двух чисел).

При выполнении сразу нескольких арифметических операций следует учитывать порядок их выполнения. Приоритет операций от наивысшего к низшему:

Инкремент, декремент.

Умножение, деление, получение остатка.

Сложение, вычитание.

Для изменения порядка следования операций применяются скобки.

Пользовательские типы данных

В приложениях есть возможность создавать свои типы данных: структуры и перечисления.

Перечисления полезны, когда переменные могут принимать значения только из определенного набора. Каждому значению в перечислении соответствует свой номер. По умолчанию нумерация элементов начинается с нуля.

```
enum Color { Red, Green, Blue }
```

Цвет Red будет иметь значение 0, Green – 1, а Blue будет 2.

Пример использования переменной перечислимого типа:

```
Color colorPalette;             // объявление переменной
colorPalette = Color.Red; // установка значения
```

или

```
colorPalette = (Color)0;      // явное преобразование из int
```

Структуры можно использовать для создания объектов, ведущих себя как встроенные типы. Они группируют данные различного типа.

```
public struct Employee
{
        public string firstName;
        public int age;
}
```

Для доступа к элементам структур применяетсяследующая конструкция:

```
Employee companyEmployee;                       // объявление переменной
companyEmployee.firstName = “Joe”;          // присваивание
companyEmployee.age = 23;                       //   значений
```

Преобразование типов

Сужающие и расширяющие преобразования

Преобразования могут сужающие (narrowing) и расширяющие (widening). Расширяющие преобразования расширяют размер объекта в памяти. Например:

```
byte a = 4;             // 0000100
ushort b = a;   // 000000000000100
```

В данном случае переменной типа ushort присваивается значение типа byte. Тип byte занимает 1 байт (8 бит), и значение переменной a в двоичном виде можно представить как

00000100

Значение типа ushort занимает 2 байта (16 бит). И при присвоении переменной b значение переменной a расширяется до 2 байт

0000000000000100

То есть значение, которое занимает 8 бит, расширяется до 16 бит.

Сужающие преобразования, наоборот, сужают значение до типа меньшей разядности. Во втором листинге статьи мы как раз имели дело с сужающими преобразованиями:

```
ushort a = 4;
    byte b = (byte) a;
```

Здесь переменной b, которая занимает 8 бит, присваивается значение ushort, которое занимает 16 бит. То есть из 0000000000000100 получаем 00000100. Таким образом, значение сужается с 16 бит (2 байт) до 8 бит (1 байт).

Явные и неявные преобразования

Неявные преобразования

В случае с расширяющими преобразованиями компилятор за нас выполнял все преобразования данных, то есть преобразования были неявными (implicit conversion). Такие преобразования не вызывают каких-то затруднений. Тем не менее стоит сказать пару слов об общей механике подобных преобразований.

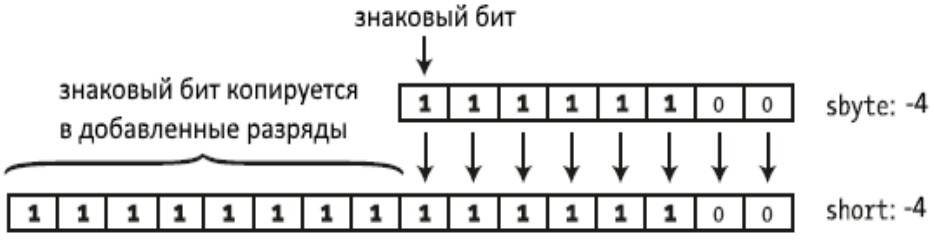
Если производится преобразование от беззнакового типа меньшей разрядности к беззнаковому типу большой разрядности, то добавляются дополнительные биты, которые имеют значени 0. Это называется дополнение нулями или zero extension.

```
byte a = 4;          // 0000100
    ushort b = a;   // 000000000000100
```

Если производится преобразование к знаковому типу, то битовое представление дополняется нулями, если число положительное, и единицами, если число отрицательное. Последний разряд числа содержит знаковый бит - 0 для положительных и 1 для отрицательных чисел. При расширении в добавленные разряды компируется знаковый бит.

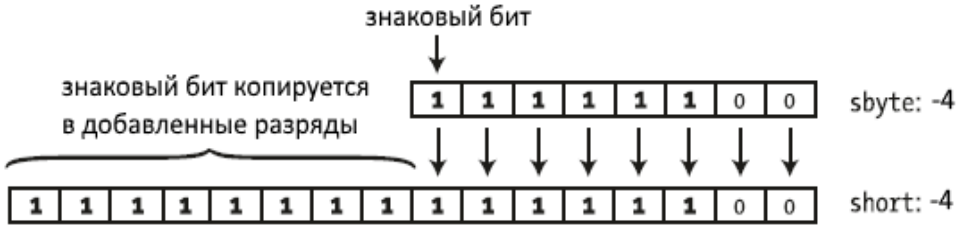
Рассмотрим преобразование положительного числа:

```
sbyte a = 4;          // 0000100
    short b = a;    // 000000000000100
```



Преобразование отрицательного числа:

```
sbyte a = -4;          // 1111100
short b = a;    // 111111111111100
```



Явные преобразования

При явных преобразованиях (explicit conversion) мы сами должны применить операцию преобразования (операция ()). Суть операции преобразования типов состоит в том, что перед значением указывается в скобках тип, к которому надо привести данное значение:

```
int a = 4;
int b = 6;
byte c = (byte)(a+b);
```

Расширяющие преобразования от типа с меньшей разрядностью к типу с большей разрядностью компилятор подводит неявно. Это могут быть следующие цепочки преобразований:

byte -> short -> int -> long -> decimal

int -> double

short -> float -> double

char -> int

Все преобразования можно описать в табл. 3.1:

Таблица 3.1

Тип	**В какие типы безопасно преобразуется**
byte	short, ushort, int, uint, long, ulong, float, double, decimal
sbyte	short, int, long, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

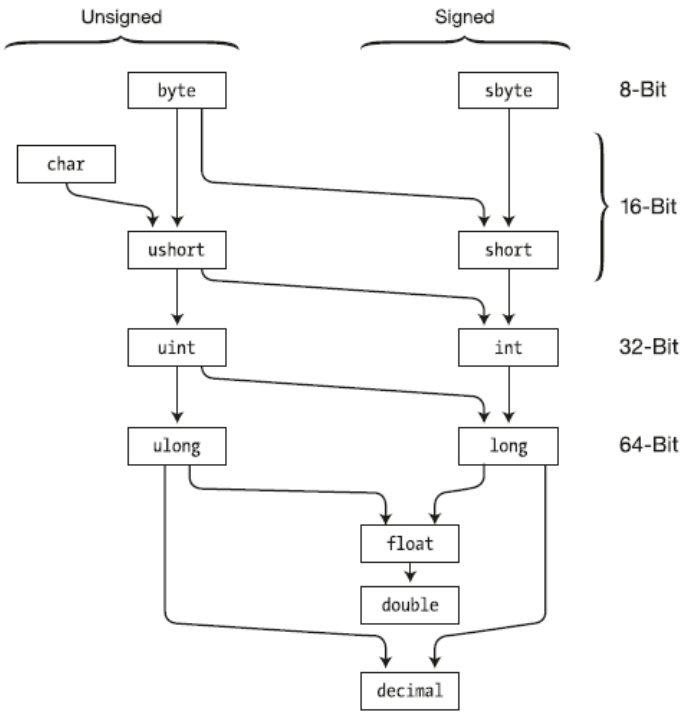


Рис. 3.1 Расширяющие преобразования от типа с меньшей разрядностью к типу с большей разрядностью

В остальных случаях следует использовать явные преобразования типов.

Также следует отметить, что несмотря на то, что и double, и decimal могут хранить дробные данные, а decimal имеет большую разрядность, чем double, все равно значение double нужно явно приводить к типу decimal:

```
double a = 4.0;
decimal b = (decimal)a;
```

Потеря данных и ключевое слово checked

Рассмотрим другую ситуацию, что будет, например, в следующем случае:

```
int a = 33;
int b = 600;
byte c = (byte)(a+b);
```

Результатом будет число 121, так число 633 не попадает в допустимый диапазон для типа byte, и старшие биты будут усекаться. В итоге получится число 121. Поэтому при преобразованиях надо это учитывать. И мы в данном случае можем либо взять такие числа a и b, которые в сумме дадут число не больше 255, либо мы можем выбрать вместо byte другой тип данных, например int.

Однако ситуации могут быть разные. Мы можем точно не знать, какие значения будут иметь числа a и b. И чтобы избежать подобных ситуаций, в c# имеется ключевое слово checked:

```
try
{
    int a = 33;
    int b = 600;
    byte c = checked((byte)(a + b));
    Console.WriteLine(c);
```

```
}
catch (OverflowException ex)
{
   Console.WriteLine(ex.Message);
}
```

При использовании ключевого слова checked приложение выбрасывает исключение о переполнении. Поэтому для его обработки в данном случае используется конструкция try...catch.

Пример неявного преобразования:

```
using System;
class Test;
{
        static void Main()
        {
                int intValue = 123;
                long longValue = intValue;
                Console.WriteLine("(long) {0} = {1}", intValue, longValue)
        }
}
```

Пример явного преобразования:

```
using System;
class Test;
{
        static void Main()
        {
                long longValue = Int64.MaxValue;
                int intValue = (int) longValue;
                Console.WriteLine("(int) {0} = {1}", longValue, intValue)
        }
}
```

Вторая программа на экране напечатает (int) 9223372036854775807 = -1, так как произойдет переполнение переменной типа int. Есть возможность отслеживать такие ошибки путем размещения данного кода внутри блока checked. Checked используется для явной проверки переполнения при выполнении арифметических операций и преобразований с данными целого типа. Пример:

```
using System;
class Test
{
static void Main()
{
        long longValue = Int64.MaxValue;
        checked{
```

```
            try{
            int intValue = (int) longValue;
            Console.WriteLine("(int) {0} = {1}", longValue, intValue);
            }
            catch (Exception e){
            Console.WriteLine("Ошибка приведения типов");
            }
        }
}
}
```

Вопросы к п. 3.3

1. Что такое общеязыковая система типов?
2. Может ли структурная переменная иметь значение null?
3. Можно ли не инициализировать переменные в C#? Почему?
4. Можно ли потерять данные в результате неявного преобразования?

Задания для самостоятельной работы

1. Создать перечислимый тип данных, отображающий виды банковского счета (текущий и сберегательный). Создать переменную типа перечисления, присвоить ей значение и вывести это значение на печать.

2. Создать структуру данных, которая хранит информацию о банковском счете – его номер, тип и баланс. Создать переменную такого типа, заполнить структуру значениями и напечатать результат.

3.4 Операторы и исключения

Операторы в C#

Программа состоит из последовательности операторов. При выполнении они запускаются по очереди. При разработке приложения необходимо группировать операторы, в C# они группируются в блоки при помощи фигурных скобок. Каждый блок задает свою область видимости, в которой существуют локальные переменные

```
{
// блок операторов
}
```

Операторы, используемые в программах, можно разделить на операторы присваивания, арифметические операторы, логические операторы, условные операторы, оператор инкремента и декремента, операторы цикла и операторы перехода. Операторы присваивания, арифметические, логические, инкремента и декремента были рассмотрены в предыдущей главе. Рассмотрим остальные операторы.

Условные операторы

Операторы if и switch – операторы выбора, они вычисляют значение условия и на основе этого выбора выполняют операторы.

Неявное преобразование из int в bool содержит потенциальный путь к ошибкам, поэтому в C# такое действие запрещено. В условии обязательно должно быть булево выражение.

```
int x; ...
if (x) ...      // Должно быть x != 0 в C#
if (x = 0) ... // Должно быть x == 0 в C#
```

Можно использовать каскадную последовательность if, для чего предусмотрена конструкция else if.

Оператор switch – механизм для обработки сложных условий. Он состоит из нескольких блоков case, каждый из которых определяется своей константой. Для группировки констант пишем каждую из них в собственном case, а выполняющийся оператор – после всей группы. Операторы выполняются до оператора break.

```
enum MonthName { January, February, ..., December }
MonthName current;
int monthDays; ...
switch (current) {
case MonthName.February :
        monthDays = 28;
        break;
case MonthName.April :
case MonthName.June :
case MonthName.September :
case MonthName.November :
        monthDays = 30;
        break;
default :
        monthDays = 31;
        break;
}
```

Если константы повторяются, то получим ошибку при компиляции:

```
switch (trumps) {
          case Suit.Clubs :
          case Suit.Clubs : // Ошибка: повтор метки
          ...
          default :
          default : // Ошибка: повтор метки
          ...
}
```

Операторы цикла

Операторы цикла выполняют операции до тех пор, пока условие верно.

Перечислим операторы цикла: while, do - while, for и foreach.

Цикл while – простейший оператор цикла. Заметим, что C# не поддерживает неявное преобразование в bool, поэтому обязательно надо использовать условие.

Отличие цикла do – while в том, что условие проверяется после выполнения операции.

Цикл for включает в себя первоначальную инициализацию счетчика, условие выполнения и обновление счетчика. Однако в нем может отсутствовать любая часть for – инициализация, условие и обновление. Например,

```
for (;;) {
        Console.WriteLine("Help ");
        ...
}
```

Цикл foreach – цикл для выборки всех элементов коллекции. Необходимо выбрать имя и тип переменной, которая будет идти по коллекции. Это переменная в теле цикла будет доступна только для чтения.

```
foreach(int number in numbers)
Console.WriteLine(number);
```

Операторы перехода

Операторы goto, break и continue – операторы перехода. Они используются для передачи управления из одной части программы в другую.

goto – простейший оператор, управление при его вызове передается в точку программы, отмеченную меткой. Метка определяется двоеточием после идентификатора.

```
if(number % 2 == 0) goto Even;
Console.WriteLine(“odd”);
goto End;
Even:
Console.WiteLine(“even”);
End:;
```

Операторы break и continue используются в циклах, первый из них предназначен для выхода из цикла, второй – для перехода к следующей итерации.

Обработка исключений

Традиционно обработка ошибок производится в теле программы, что затрудняет понимание логики самого приложения и запутывает код. Сообщения об ошибках представляют собой числа, которые не несут никакой

смысловой нагрузки. Более того, один и тот же код ошибки может использоваться в разных функциях и означать совершенно разные ошибки.

В C# предусмотрен специальный механизм для генерирования и обработки исключительных ситуаций. Все исключения – это объекты, унаследованные от класса Exception. Названия классов раскрывают вид исключения. Объекты могут содержать дополнительную информацию об ошибке, например FileNotFoundException может хранить имя файла.

Для обработки исключительной ситуации необходимо потенциально опасный код заключить в блок **try-catch-finally.** В блоке try содержится код, который может сгенерировать исключительную ситуацию, например деление на ноль или отсутствие запрашиваемого файла на диске. Если возникла исключительная ситуация, выполнение кода в блоке try приостанавливается и начинается поиск блока catch, соответствующего по типу исключительной ситуации. Блок finally выполняется обязательно, вне зависимости от того, произошло исключение или нет. Он используется в двух случаях: для избегания повтора операторов и для освобождения ресурсов.

Есть несколько вариантов блока catch:

– общий catch { ... }, что равносильно catch (System.Exception) { ... } – обработка ошибки любого типа;

– по виду исключения, например catch (OutOfMemoryException caught) { ... }, где caught – объект класса исключения OutOfMemoryException.

После блока try может присутствовать несколько блоков catch. В этом случае у каждого блока catch необходимо явно указывать, ошибку какого типа он обрабатывает. Нельзя чтобы в следующем блоке catch был указан тип данных, наследник от класса, указанном в предыдущем блоке catch. Нельзя чтобы было два блока catch, предназначенных для обработки ошибки одного и того же класса. Блок catch для ошибки типа System.Exception может быть только последним.

Приведем несколько примеров.

```
try {
        Console.WriteLine("Enter first number");
        int i = int.Parse(Console.ReadLine());
        Console.WriteLine("Enter second number");
        int j = int.Parse(Console.ReadLine());
        int k = i / j;
}
catch (OverflowException caught)
{
        Console.WriteLine(caught);
}
catch(DivideByZeroException caught)
{
        Console.WriteLine(caught);
}
```

```
finally
{
        Console.WriteLine(“Блок try-catch завершен”);
}
```

Приведенный выше код не содержал никаких ошибок.

```
try {
...
}
catch { ... }
catch (OutOfMemoryException caught) { ... } // ошибка – общий блок catch
```

следует перед блоком с конкретной ошибкой.

В следующем примере один и тот же класс используется два раза:

```
catch (OutOfMemoryException caught) { ... }
catch (OutOfMemoryException caught) { ... } // Error
```

В следующем примере сначала идет блок catch, который обрабатывает ошибку класса Exception, в следующем блоке – ошибку класса OutOfMemoryException, который является наследником от класса Exception.

```
catch (Exception caught) { ... }
catch (OutOfMemoryException caught) { ... } // Error
```

Оператор throw генерирует исключение вручную. Можно генерировать только объекты класса исключения, то есть для этого надо создать свой собственный класс, наследник от класса Exception.

```
if (minute<1|| minute>=60){
          string fault=minute + “is not a valid minute”;
         throw new InvalidTimeException(fault);
…
}
```

В приведенном примере предполагается, что класс InvalidTimeException описан где-то раньше, этот класс является наследником от класса Exception, его конструктор содержит один строковый параметр.

Можно использовать оператор throw внутри блока catch:

```
catch (IOException caught) {
        ...
        throw new FileNotFoundException(filename);
}
```

По умолчанию проверка переполнения стека арифметическими операциями в C# выключена. Для включения проверки переполнения можно создать блок checked или при компиляции указать опцию: csc /checked+ example.cs, а для отключения:

csc /checked- example.cs. Блок unchecked явно выключает проверку.

Рекомендации по обработке исключений:

– Включать строки описания

```
string description = String.Format("{0}({1}): newline in string constant", filename,
```

linenumber);
throw new System.Exception(description).

– Генерировать исключения более специфичного вида, например класс FileNotFoundException, лучше чем более общий класс IOException.

– Не позволять исключениям выходить из метода Main, то есть всегда должна существовать такая конструкция

```
static void Main( )
{
        try {
                ...
        }
         catch (Exception caught) {
          ...
         }
}
```

Вопросы к п. 3.4

1. Перечислите основные группы операторов, выделив их характерные особенности.

2. Что такое исключение? Какие классы для обработки исключительных ситуаций вы знаете?

3. Как организовать обработку исключительной ситуации?

Задания для самостоятельной работы

Задания на операторы if, switch, for, while, foreach, обработку исключений.

1. Напишите консольную программу, в которую пользователь вводит с клавиатуры два числа. Программа сранивает два введенных числа и выводит на консоль результат сравнения (два числа равны, первое число больше второго или первое число меньше второго).

2. Написать программу, которая читает с экрана число от 1 до 365 (номер дня в году), переводит этот число в месяц и день месяца. Например, число 40 соответствует 9 февраля (високосный год не учитывать).

3. Добавить к задаче из предыдущего задания проверку числа, введенного пользователем. Если число меньше 1 или больше 365, программа должна вырабатывать исключение и выдавать на экран сообщение.

4. Изменить программу из заданий 1 и 2 так, чтобы она учитывала год (високосный или нет). Год вводится с экрана. (Год високосный, если он делится на четыре без остатка, но если он делится на 100 без остатка, это не високосный год. Однако, если он делится без остатка на 400, это високосный год.)

5. В банке в зависимости от суммы вклада начисляемый процент по вкладу может отличаться. Напишите консольную программу, в которую пользователь вводит сумму вклада. Если сумма вклада меньше 100, то начисляется 5 %. Если сумма вклада от 100 до 200, то начисляется 7 %. Если

сумма вклада больше 200, то начисляется 10 %. В конце программа должна выводить сумму вклада с начисленными процентами.

Для получения вводимого с клавиатуры числа используйте выражение Convert.ToDouble(Console.ReadLine()).

6. За каждый месяц банк начисляет к сумме вклада 7 % от суммы. Напишите консольную программу, в которую пользователь вводит сумму вклада и количество месяцев. Банк вычисляет конечную сумму вклада с учетом начисления процентов за каждый месяц.

Для вычисления суммы с учетом процентов используйте цикл for.

Для ввода суммы вклада используйте выражение Convert.ToDecimal(Console.ReadLine()) (сумма вклада будет представлять тип decimal).

7. Напишите программу, в которую пользователь вводит два числа и выводит результат их умножения. При этом программа должны запрашивать у пользователя ввод чисел, пока оба вводимых числа не окажутся в диапазоне от 0 до 10. Если введенные числа окажутся больше 10 или меньше 0, то программа должна предупредить пользователя о том, что введенные числа недопустимы, и повторно запросить у пользователя ввод двух чисел. Если введенные числа принадлежат диапазону от 0 до 10, то программа выводит результат умножения.

Для организации ввода чисел используйте бесконечный цикл while и оператор break.

3.5 Методы и параметры

При разработке большинства приложений их разделяют на функциональные модули, так как маленькие разделы кода удобнее для понимания, разработки и отладки. Кроме того, это позволяет несколько раз использовать одни и те же участки кода для других приложений.

В C# приложение состоит из классов, которые содержат именованные блоки кода, называемые методами. Метод – это член класса, который может осуществлять действия или вычислять значения.

Использование методов

В C# программа состоит из классов, содержащих методы. Метод может выполнять действие или вычислять значение.

Метод – это набор операторов, которые выполняются вместе. В C# все методы принадлежат какому-либо классу. Для создания метода необходимо задать его имя, определить список параметров и тело метода.

Для вызова метода используется имя метода: если у метода есть параметры, то необходимо их определить. Для вызова метода другого класса необходимо, чтобы он был объявлен как public, вызов осуществляется по имени класса и метода.

using System;

```
class NestExample
{
        static void Method1
        {
                Console.WriteLine(“Method1”);
         }
         static void Method2
        {
                 Method1();
                 Console.WriteLine(“Method2”);
                 Method1();
        }
        static void Main()
       {
       Method2();
       Method1();
       }
}
```

В результате получим:

```
Method1
Method2
Method1
Method1
```

Оператор return останавливает выполнение метода и передаёт управление вызвавшему данный метод оператору. Если метод не void, то необходимо вернуть значение соответствующего типа.

Каждый метод имеет набор своих локальных переменных, они видны только в нем и при завершении работы метода уничтожаются. Для того чтобы переменные были видны в нескольких методах класса, необходимо объявить их полями вне метода, но внутри класса.

Для не void методов необходимо возвращать значение, каждый «путь выполнения» метода должен заканчиваться оператором return. Для void-методов оператор return не обязателен.

Использование параметров

В C# существуют три варианта передачи параметров: по значению, по ссылке и выходные параметры. При передаче параметров по значению изменение значения параметра в методе не влияет на значение в вызвавшем методе.

```
static void AddOne(int x)
{
        x++;
}
```

```
static void Main()
{
        int k = 6;
        AddOne(k);
        Console.WriteLine(k); // Выведет на экран 6, не 7
}
```

При передаче по ссылке передается ссылка на переменную, поэтому все действия, производимые с параметром, оказывают влияние на значение переменной в вызывающем методе. Для каждого параметра по ссылке необходимо указывать ключевое слово ref. До вызова метода необходимо обязательно инициализировать переменную.

```
static void AddOne(ref int x)
{
        x++;
}
static void Main()
{
        int k = 6;
        AddOne(ref k);
        Console.WriteLine(k); // Выведет на экран 7
}
```

Выходные параметры похожи на параметры по ссылке, единственное отличие их состоит в том, что их можно не инициализировать до вызова метода. При передаче выходного параметра перед ним указывается ключевое слово out.

```
static void OutDemo(out int p)
{
        // …
}
static void Main()
{
        int k;
        OutDemo(out k);
        Console.WriteLine(k);
}
```

C# позволяет использовать механизм передачи списка параметров изменяемой длины. Для этого используется ключевое слово params. Правило использования: допустим только один список параметров, который должен быть массивом конкретного типа и размещаться последним в общем списке параметров.

```
static long AddList(int k, params long[] v)
```

```
{
        long total;
        long i;
        for(i = 0, total = 0; i < v.Length; i++)
                total += v[i];
        return total*k;
}
```

При вызове можно использовать два пути: вызывать как список отдельных элементов или как массив.

```
static void Main( )
{
        long x;
        x = AddList(63, 21, 84); // Список
        x = AddList(new long[ ]{ 63, 21, 84 }); // Массив
}
```

Для выбора вида параметров необходимо учитывать два аспекта: механизм передачи и эффективность. По эффективности передача по значению лучше, чем передачи по ссылке.

Когда метод вызывает себя, то это называется рекурсией. Если это происходит при участии другого метода, то это будет неявной рекурсией. Рекурсия часто используется для упрощения логики программы.

Перегрузка методов

Имя метода не может совпадать с именем любого другого элемента класса, но может совпадать с другим методом – это называется перегрузка метода. Перегружаемые методы – это методы с одинаковыми именами в одном классе.

```
class OverloadingExample
{
        static int Add(int a, intb)
        {
                return a + b;
        }
        static int Add(int a, intb, int c)
        {
                return a + b + c;
        }
         static void Main()
        {
                Console.WriteLine(Add(1,2) + Add(1,2,3));
        }
}
```

Нельзя использовать одно имя для метода и переменной, константы или

перечисления.

```
class BadMethodNames
{
        static int k;
       static void k( )
       {
            // ...
       }
}
```

Компилятор использует сигнатуру метода для различия методов в классе. В сигнатуру входят следующие элементы: имя метода, типы параметров, модификаторы параметров. В сигнатуру не входят имена параметров и возвращаемый тип метода.

Следующие три метода имеют различную сигнатуру

```
static int LastErrorCode( ) { }
static int LastErrorCode(int n) {  }
static int LastErrorCode(int n, int p) {  }
```

Следующие три метода имеют одинаковую сигнатуру

```
static int LastErrorCode(int n) { }
static string LastErrorCode(int n) { }
 static int LastErrorCode(int x) { }
```

Вопросы к п. 3.5

1. Объясните, что такое методы и почему они важны.
2. Опишите три возможных пути передачи параметров и соответствующие ключевые слова C#.
3. Когда создаются и уничтожаются локальные переменные?
4. Что входит в сигнатуру метода?

Задания для самостоятельной работы

Задания на методы с параметрами и без, различные механизмы передачи параметров.

1. Написать метод, возвращающий наибольшее из двух чисел. Входные параметры метода – два целых числа. Протестировать метод.
2. Написать метод, который меняет местами значения двух передаваемых параметров. Параметры передавать по ссылке. Протестировать метод.
3. Написать метод вычисления факториала числа, результат вычислений передавать в выходном параметре. Если метод отработал успешно, то вернуть значение true; если в процессе вычисления возникло переполнение, то вернуть значение false. Для отслеживания переполнения значения использовать блок checked.
4. Написать рекурсивный метод вычисления факториала числа.
5. Написать метод, который вычисляет НОД двух натуральных чисел (алгоритм Евклида). Написать метод с тем же именем, который вычисляет НОД

трех натуральных чисел.

3.6 Массивы и коллекции

Массивы хороший инструмент группировки данных. Однако массивы хранят фиксированное количество объектов, а иногда заранее не неизвестно, сколько потребуется объектов. И в этом случае намного удобнее применять коллекции. Еще один плюс коллекций состоит в том, что некоторые из них реализуют стандартные структуры данных, например, стек, очередь, словарь, которые могут пригодиться для решения различных специальных задач.

Большая часть классов коллекций содержится в пространствах имен System.Collections (простые необобщенные классы коллекций), System.Collections.Generic (обобщенные или типизированные классы коллекций) и System.Collections.Specialized (специальные классы коллекций). Также для обеспечения параллельного выполнения задач и многопоточного доступа применяются классы коллекций из пространства имен System.Collections.Concurrent.

Массивы

Синтаксис массивов в C#

```
type[ ]  name;      // правильно
type     name[ ];   // неправильно в C#
type[4]name;        // также неправильно C#
```

Для объявления двумерных массивов используются пустые индексы через запятую.

```
int[,] grid;
```

Для доступа к элементам массива используются индексы в квадратных скобках. Нумерация начинается с нуля. Если размерность массива больше одного, то индексы перечисляются через запятую.

```
long[] row;                    int[,] grid;
…                              …
row[3];                        grid[1,2];
```

В C# индексы массива автоматически проверяются на удовлетворение размерности, при выходе за интервал выдаётся исключение IndexOutOfRangeException. Для проверки размерности можно использовать свойство массива Length и метод GetLength.

Сравним массивы с коллекциями. Коллекции гибче массивов, могут хранить различные элементы и имеют переменную длину. Но в связи с этим работа с коллекциями медленнее.

```
ArrayList flexible = new ArrayList( );
 flexible.Add("one"); // Добавили строку...
flexible.Add(99);      // Добавили int
```

Создание коллекции только для чтения

```
ArrayList flexible = new ArrayList( ); ...
ArrayList noWrite = ArrayList.ReadOnly(flexible);
noWrite[0] = 42;      // Исключение при выполнении
```

Объявление массива не создает его, так как массив – это ссылочный тип. При объявлении массива можно не знать его размерность, но при создании знать размер необходимо. Память для массива выделяется последовательно.

```
long[] row = new long[4];
int[,] grid = new int[2,3];
```

Можно использовать список инициализации при создании массива. Необходимо объявить все элементы, в качестве элементов можно использовать выражения. Те же правила справедливы для многомерных массивов: необходимо определить все строки и в каждой строке все элементы.

```
int[,] data = new int[2,3]{
        {2,3,4},
         {5,6,7}  }
```

Размерность массива можно задавать как константами, так и вычисляемыми значениями. Единственное ограничение при вычисляемой длине, нельзя использовать список инициализации.

```
string s = Console.ReadLine();
int size = int.Parse(s);
long[] row = new long[size];
```

При копировании переменной массива не создается новый массив, создается новая ссылка на тот же массив.

```
long [] row = new long[4];
long [] copy = row;
row[0]++; // изменит copy[0]
```

Рассмотрим некоторые свойства и методы класса System.Array.

– Свойство Rank – размерность массива.

– Свойство Length – общая длина массива, для многомерных массивов количество всех ячеек.

– System.Array – класс, от которого неявно наследуют все массивы в C#.

– Sort – сортировка массива, поддержка интерфейса IComparable.

– Clear – очистка массива, все элементы устанавливаются в NULL.

– Clone – создаёт копию массива, копирует все элементы. Этот метод не следит за значениями в элементах. Если там ссылки на объекты, то они просто скопируются, новых объектов создано не будет.

– GetLength – по номеру размерности получаем длину массива в этой размерности.

– IndexOf – ищет значение в массиве, если нашел, возвращает индекс первого вхождения, иначе -1.

При возвращении массива из метода его размеры не задаются, скобки

остаются пустыми. Если задать, получим ошибку при компиляции. То же с многомерными массивами.

```
static int[,] CreateArray( ) {
        string s1 = System.Console.ReadLine( );
        int rows = int.Parse(s1);
        string s2 = System.Console.ReadLine( );
        int cols = int.Parse(s2);
        return new int[rows,cols];
}
```

При передаче массива в метод в качестве параметра новый массив не создается, передается ссылка на тот же массив. Все действия с массивом в методе сохранятся для вызывающего метода. Если нужно избежать этого, необходимо передавать копию массива при помощи метода Array.Copy

При вызове приложения из командной строки можно использовать строку параметров, которые разделяются пробелом. Например:

C:\> pkzip –add –rec –path=relative c:\code *.cs Тогда, если pkzip написан на C#, получим массив

```
string[ ] args = {
"-add",
"-rec",
"-path=relative", "c:\\code",
"*.cs"
};
```

Этот массив получим при запуске метода Main.

```
class  PKZip
{
        static void Main(string[] args)
        {
        }
}
```

Для прохода по массиву можно использовать цикл foreach. Тогда не нужен счетчик, проверка длины массива и обращение к элементу.

Две следующие конструкции эквивалентны:

```
for (int i = 0; i < args.Length; i++) {
        System.Console.WriteLine(args[i]);
}
```

и

```
foreach (string arg in args) {
        System.Console.WriteLine(arg);
}
```

Также foreach можно использовать и для многомерных массивов:

```
int[,] numbers = { {0,1,2}, {3,4,5} };
foreach (int number in numbers) {
        System.Console.WriteLine(number);
}
```

Списки – List<T>

Класс List<T> является самым простым из классов коллекций. Его можно использовать практически так же, как массив, ссылаясь на существующий в коллекции List<T> элемент с применением обычной для массивов системы записи с квадратными скобками и индексом элемента. Можно добавить элемент к концу коллекции List<T>, воспользовавшись имеющимся в ее классе методом Add, которому предоставляется добавляемый элемент. Размер коллекции увеличивается List<T> автоматически.

```
List<int> list = new List<int>();
list.Add(3);
list.Add(1);
 list[0]=list[1]+3;
```

Указывать размер коллекции List<T> при ее создании не обязательно. Коллекция может изменять свои размеры по мере добавления (или удаления) элементов. Но надо иметь в виду, что на физическое добавление элементов уходит время процессора и при необходимости нужно указать начальный размер. Но если он будет превышен, то в силу необходимости коллекция List<T> просто расширится. Если первую строку примера изменить на

```
List<int> list = new List<int>(2);
```

то результат будет один и тот же, но последний вариант отработает быстрее. В этом случае команда Add – это просто инициализация очередного элемента в конце списка.

Для удаления из коллекции List<T> указанного элемента можно воспользоваться методом Remove. Элементы коллекции List<T> автоматически перестроятся, закрывая образовавшееся пустое место. С помощью метода RemoveAt можно также удалить элемент, указав его позицию в коллекции List<T>. Можно вставить элемент в середину коллекции List<T>, воспользовавшись для этого методом Insert. При этом размер коллекции List<T> также изменится автоматически. Размер коллекции, т.е. количество инициализированных элементов, можно получить через свойство (только на чтение) Count, а емкость коллекции – через свойство (чтение и запись) Capacity.

Приведем ряд методов класса List<T>:

– Add(T) – добавляет объект в конец списка List<T>.

– AddRange(IEnumerable<T>) – добавляет элементы указанной коллекции в конец списка List<T>.

– Clear()– удаляет все элементы из коллекции List<T>.

– Contains(T) – определяет, входит ли элемент в коллекцию List<T>.

– ConvertAll<TOutput>(Converter<T, TOutput>) – преобразует элементы текущего списка List<T> в другой тип и возвращает список преобразованных

элементов.

– Exists(Predicate<T>) – определяет, содержит ли List<T> элементы, удовлетворяющие условиям указанного предиката.

– Find(Predicate<T>) – выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает первое найденное вхождение в пределах всего списка List<T>.

– FindAll(Predicate<T>) – извлекает все элементы, удовлетворяющие условиям указанного предиката.

– FindIndex(Predicate<T>) – выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает отсчитываемый от нуля индекс первого найденного вхождения в пределах всего списка List<T>.

– IndexOf(T) – осуществляет поиск указанного объекта и возвращает отсчитываемый от нуля индекс первого вхождения, найденного в пределах всего списка List<T>.

– Insert(Int32, T) – вставляет элемент в коллекцию List<T> по указанному индексу.

– Remove(T) – удаляет первое вхождение указанного объекта из коллекции List<T>.

– RemoveAll(Predicate<T>) – удаляет все элементы, удовлетворяющие условиям указанного предиката.

– RemoveAt(Int32) – удаляет элемент списка List<T> с указанным индексом.

– Sort() – сортирует элементы во всем списке List<T> с помощью функции сравнения по умолчанию.

– ToArray() – копирует элементы списка List<T> в новый массив. Многие методы коллекции List<T> содержатся и в других коллекциях.

Двухсвязные списки – LinkedList<T>

Класс коллекций LinkedList<T> реализует двусвязный список. В каждом элементе списка содержится значение элемента со ссылкой на следующий элемент списка (свойство Next) и его предыдущий элемент (свойство Previous).

В классе LinkedList<T> записи, присущие массивам, не поддерживаются. Вставка элементов осуществляется отличным от List<T> способом. Можно воспользоваться методом AddFirst для вставки элемента в начало списка с перемещением предыдущего первого элемента дальше по списку и установки в качестве значения его свойства Previous ссылки на новый элемент. Аналогично этому для вставки элемента в конец списка можно воспользоваться методом AddLast. Для вставки элемента перед указанным элементом списка или после него можно применить методы AddBefore и AddAfter.

Первый элемент коллекции LinkedList<T> можно найти, запросив значение свойства First, а свойство Last даст ссылку на последний элемент списка. Для последовательного обхода элементов связного списка можно приступить к этой операции с одного конца и пошагово применять ссылки из

свойства Next или Previous, пока не будет найден элемент, у которого это свойство имеет значение null. Конечно же, лучше воспользоваться инструкцией foreach, которая выполнит последовательный обход элементов вперед по списку LinkedList<T>-объекта, автоматически остановившись в конце.

Удаление элемента из коллекции LinkedList<T> осуществляется с помощью методов Remove, RemoveFirst и RemoveLast.

Преимущество связного списка проявляется в том, что операция вставки элемента в середину выполняется очень быстро. Это происходит за счет того, что только ссылки Next (следующий) предыдущего элемента и Previous (предыдущий) следующего элемента должны быть изменены так, чтобы указывать на вставляемый элемент. В классе List<T> при вставке нового элемента все последующие должны быть сдвинуты.

Естественно, у связных списков есть и свои недостатки. Так, например, все элементы таких списков доступны лишь друг за другом. Поэтому для нахождения элемента, который расположен в середине или в конце списка, требуется довольно много времени. Связный список не может просто хранить элементы внутри себя. Вместе с каждым из них ему необходимо иметь информацию о следующем и предыдущем элементах. Поэтому LinkedList<T> содержит элементы типа LinkedListNode<T>. С помощью класса LinkedListNode<T> появляется возможность обратиться к предыдущему и последующему элементам списка. Класс LinkedListNode<T> определяет свойства List, Next, Previous и Value. Свойство List возвращает объект LinkedList<T>, ассоциированный с узлом. Свойства Next и Previous предназначены для итераций по списку и для доступа к следующему и предыдущему элементам. Свойство Value типа T возвращает элемент, соответствующий узлу.

Если в простом списке List<T> каждый элемент представляет объект типа T, то в LinkedList<T> каждый узел помимо T содержит также объект класса LinkedListNode<T>. Этот класс имеет следующие свойства:

– Value – значение узла, представленное типом T.

– Next – ссылка на следующий элемент типа LinkedListNode<T> в списке. Если следующий элемент отсутствует, то имеет значение null.

– Previous – ссылка на предыдущий элемент типа LinkedListNode<T> в списке. Если предыдущий элемент отсутствует, то имеет значение null.

– Используя методы класса LinkedList<T>, можно обращаться к различным элементам как в конце, так и в начале списка:

– AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode): вставляет узел newNode в список после узла node.

– AddAfter(LinkedListNode<T> node, T value): вставляет в список новый узел со значением value после узла node.

– AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode): вставляет в список узел newNode перед узлом node.

– AddBefore(LinkedListNode<T> node, T value): вставляет в список новый узел со значением value перед узлом node.

– AddFirst(LinkedListNode<T> node): вставляет новый узел в начало списка.

– AddFirst(T value): вставляет новый узел со значением value в начало списка.

– AddLast(LinkedListNode<T> node): вставляет новый узел в конец списка.

– AddLast(T value): вставляет новый узел со значением value в конец списка.

– RemoveFirst(): удаляет первый узел из списка. После этого новым первым узлом становится узел, следующий за удаленным.

– RemoveLast(): удаляет последний узел из списка.

Словари – Dictionary<TKey, TValue>

Массив и объекты типа List<T> предоставляют способ отображения на элемент целочисленного индекса. Целочисленный индекс указывается с помощью квадратных скобок (например, [4]), и извлекается элемент по индексу 4, будучи фактически пятым. Но иногда может понадобиться реализация отображения, при котором используется другой, нецелочисленный тип, например string, double или DateTime. В других языках программирования такая организация хранения данных часто называется ассоциативным массивом. Эта функциональная возможность реализуется в классе Dictionary<TKey, TValue> путем внутреннего обслуживания двух массивов, один из которых предназначен для ключей, от которых выполняется отображение на одно из отображаемых значений. Когда в коллекцию Dictionary<TKey, TValue> вставляется пара «ключ–значение», класс автоматически отслеживает принадлежность ключа к значению, позволяя быстро и легко извлекать значение, связанное с указанным ключом. В конструкции класса Dictionary<TKey, TValue> имеется ряд важных особенностей.

В коллекции Dictionary<TKey, TValue> не могут содержаться продублированные ключи. Если для добавления уже имеющегося в массиве ключа вызывается метод Add, выдается исключение. Но для добавления пары «ключ–значение» можно воспользоваться системой записи с использованием квадратных скобок, не опасаясь при этом выдачи исключения, даже если ключ уже был добавлен: любое значение с таким же самым ключом будет переписано новым значением. Протестировать наличие в коллекции Dictionary<TKey, TValue> конкретного ключа можно с помощью метода ContainsKey.

По внутреннему устройству коллекция Dictionary<TKey, TValue> является разряженной структурой данных, работающей наиболее эффективно, когда в ее распоряжении имеется довольно большой объем памяти. По мере вставки элементов размер коллекции Dictionary<TKey, TValue> в памяти может очень быстро увеличиваться.

Когда для последовательного обхода элементов коллекции Dictionary<TKey, TValue> используется инструкция foreach, возвращается

элемент KeyValue-Pair<TKey, TValue>. Это структура, содержащая копию элементов ключа и значения, находящихся в коллекции Dictionary<TKey, TValue>, и доступ к каждому элементу можно получить через свойства Key и Value. Эти элементы доступны только для чтения, и их нельзя использовать для изменения данных в коллекции Dictionary<TKey, TValue>.

Отметим, что есть схожая необобщенная коллекция – Hashtable, имеющая такой же функционал. Однако эта коллекция проигрывает в скорости при работе с однотипными объектами и используется, как и все необобщенные коллекции, для группировки различных объектов.

Относительно производительности рассмотренных здесь коллекций заметим, что добавление нового объекта (Add) быстрее делает List<T>, медленнее – Dictionary<TKey, TValue>. На поиск элемента уходит примерно одинаковое время, однако поиск по ключу существенно быстрее в Dictionary<TKey, TValue>. Удаление объекта медленнее делается в классе List<T>.

Вопросы к п. 3.6

1. В чем отличия коллекций от массивов?
2. Перечислите основные свойства и методы класса System.Array.
3. Приведите примеры описания массивов и коллекций.
4. Как передавать и возвращать массивы и коллекции из методов.
5. Объясните принцип работы цикла foreach.
6. Скажите о плюсах и минусах использования двусвязных списков.
7. Приведите практические примеры эффективного использования рассмотренных коллекций.

Задания для самостоятельной работы

Задания на массивы, передачу массива в качестве аргумента методу Main

1. Написать программу, которая вычисляет число гласных и согласных букв в файле. Имя файла передавать как аргумент в функцию Main. Содержимое текстового файла заносится в массив символов. Количество гласных и согласных букв определяется проходом по массиву. Предусмотреть метод, входным параметром которого является массив символов. Метод вычисляет количество гласных и согласных букв.

2. Написать программу, реализующую умножение двух матриц, заданных в виде двумерного массива. В программе предусмотреть два метода: метод печати матрицы, метод умножения матриц (на вход две матрицы, возвращаемое значение – матрица).

3. Написать программу, вычисляющую среднюю температуру за год. Создать двумерный рандомный массив temperature[12,30], в котором будет храниться температура для каждого дня месяца (предполагается, что в каждом месяце 30 дней). Сгенерировать значения температур случайным образом. Для каждого месяца распечатать среднюю температуру. Для этого написать метод, который по массиву temperature [12,30] для каждого месяца вычисляет

среднюю температуру в нем, и в качестве результата возвращает массив средних температур. Полученный массив средних температур отсортировать по возрастанию.

3.7 Основы объектно-ориентированного программирования

C# – объектно-ориентированный язык. В этом пункте изучим терминологию и концепцию ООП.

Классы и объекты

Ключевым словом в ООП является «класс». Все языки программирования могут обращаться с общими данными и методами. Эта возможность помогает избежать дублирования. Главная концепция программирования – не писать один и тот же код дважды. Программы без дублирования лучше и понятнее, так как содержат меньше кода. ООП переводит эту концепцию на новый уровень, он позволяет описывать классы (множества объектов), которые делают общими и структуру, и поведение. Классы не ограничиваются описанием конкретных объектов, они также могут описывать и абстрактные вещи.

Объект – это конкретный представитель класса. Его определяет три характеристики: уникальность, поведение и состояние. *Уникальность* – это характеристика, определяющая отличие одного объекта от другого. *Поведение* определяет то, чем объект может быть полезен, что он может делать. Поведение объекта классифицирует его. Объекты разных классов отличаются своим поведением. *Состояние* описывает внутреннюю работу объекта, то, что обеспечивает его поведение. Хорошо спроектированный объект оставляет своё состояние недоступным. Нас не интересует, как он это делает, нам важно то, что он умеет это делать.

Сравним структуры и классы. В C# структуры могут иметь методы, но желательно избегать этого. Однако в некоторых структурах необходимы операторы. Операторы – стилизованные методы, но они не добавляют поведение, а обеспечивают более краткий синтаксис.

Структурные типы – нижний уровень программы, это элементы, из которых строятся более сложные элементы. Переменные структурных типов свободно копируются и используются как поля и атрибуты объектов.

Ссылочные типы – верхний уровень программы, они состоят из мелких элементов. Ссылочные типы в основном не могут быть скопированы.

Абстракция – это тактика очистки объекта от всех несущественных деталей, остается только существенная минимальная форма. Абстракция – важный принцип программирования. Хорошо спроектированный класс содержит минимальный набор методов, полностью описывающий его поведение.

Инкапсуляция данных

Традиционное процедурное программирование содержит много данных и много процедур. Любая функция имеет доступ к любым данным. Когда

программы становятся большими, это создаёт много проблем – при небольших изменениях в коде необходимо следить за всей программой. Другая проблема – это хранение данных отдельно от функций. В ООП эта проблема решается за счет объединения данных и методов, которые работают с этими данными как одно целое.

Данные и функции объединены в одну сущность, эта сущность ограничивает капсулу. Получаем две области: снаружи этой капсулы и внутри неё. Элементы, которые доступны извне, называются public, те, которые доступны только внутри класса, – private. C# не ограничивает области видимости, любой элемент может быть как public, так и private.

Две причины использования инкапсуляции – это контроль использования и минимизация воздействий при изменениях. Можно использовать инкапсуляцию данных и определить поведение для того, чтобы с объектом работали по заданным правилам. Вторая причина вытекает из первой, если данные закрыты от внешнего использования, то их изменение не влияет на использование объекта извне.

Большинство данных внутри объектов описывает информацию об индивидуальности объекта. Данные внутри объекта обычно private и доступны только из методов класса.

Иногда необязательно хранить информацию внутри каждого объекта. То есть может быть информация, одинаковая для всех объектов данного класса. Для этого используются статичные поля, которые принадлежат не конкретному объекту, а всему классу.

Статичные методы инкапсулируют статичные данные. Статичные методы существуют на уровне класса, в них нельзя использовать оператор this, но можно обращаться к полям класса, если получить объект класса как параметр.

```
class Time
{
        public static void Reset(Time t)
        {
                t.hour = 0;   // ОК
                t.minute = 0; // ОК
                hour = 0;      // ошибка при компиляции
                minute = 0;     // ошибка при компиляции
        }
        private int hour, minute;
}
```

Теперь вернемся к программе Hello world, рассмотрим её со стороны объектно-ориентированного программирования. Ответим на два вопроса: как при выполнении вызывается класс и почему метод Main статичный?

Если в файле два класса с методом Main, то точка входа определяется при компиляции.

```
// TwoEntries.cs
```

```
using System;
class EntranceOne
{
        public static void Main( )
        {
                Console.Write("EntranceOne.Main( )");
        }
}
class EntranceTwo
{
        public static void Main( )
        {
                Console.Write("EntranceTwo.Main( )");
        }
}
// Конец файла

c:\> csc /main:EntranceOne TwoEntries.
cs c:\> twoentries.exe
EntranceOne.Main( )
c:\> csc /main:EntranceTwo TwoEntries.cs
c:\> twoentries.exe
EntranceTwo.Main( ) c:\>
```

Если в файле нет классов с методом Main, то из него нельзя скомпилировать запускаемый файл, только библиотеку dll.

```
// NoEntrance.cs
using System;
class NoEntrance
{
        public static void NotMain( )
        {
                Console.Write("NoEntrance.NotMain( )");
        }
}
// Конец файла
c:\> csc /target:library NoEntrance.cs
c:\> dir
...
NoEntrance.dll
...
```

Почему метод Main должен быть статичным? Так как для вызова нестатичного метода необходимо создать объект, а при вызове Main программа только начинает работу, и никаких объектов ещё нет.

Для определения простых классов необходимо выполнить следующую

последовательность действий: обозначить ключевым словом class начало класса, определить поля как в структурах, определить методы внутри класса, установить модификаторы доступа для всех полей и методов класса. Модификатор public означает, что «доступ неограничен», private – «доступ ограничен типом, которому принадлежит». Если пропустить модификатор, то по умолчанию поле или метод будет private.

```
class BankAccount
{
        public void Deposit(decimal amount)
        {
                balance += amount;
         }
         private decimal balance;
}
```

При объявлении объекта класса объект не создаётся, необходимо использовать оператор new, при этом все поля инициализируются нулями. При использовании объекта без создания будет ошибка при компиляции.

```
Time now = new Time();    // пример объявления объекта класса
```

Ключевое слово this неявно указывает на объект вызвавший метод. Например, в следующем примере полю name нельзя присвоить значение, компилятор будет считать его параметром.

```
class BankAccount
{
        public void SetName(string name)
        {
                name = name;
         }
         private string name;
}
```

Здесь необходимо было использовать this.name = name;

В C# можно выделить пять различных видов типов:

class, struct, interface, enum, delegate.

Любой из них можно использовать в классе. То есть в классе могут содержаться другие классы. Вложенные классы должны помечаться модификатором доступа, использование вложенных классов переводит имена из глобальной области видимости и пространства имен.

Public вложенные классы не имеют ограничений по использованию, полное имя класса можно использовать в любом месте программы. Private вложенный класс виден только из класса, содержащего его. Класс без модификатора по умолчанию является private.

Наследование и полиморфизм

Наследование – это связи на уровне классов. Новый класс может наследовать существующий класс. Наследование – это мощная связь, так как

наследуемый класс наследует все (не private) элементы базового класса. От базового класса может наследоваться любое количество классов. Изменение базового класса, автоматически изменяет все классы-потомки.

Классы-потомки могут быть одновременно и базовыми классами для других классов. Группа классов, связанных наследованием, формирует структуру, называемую иерархией классов. При движении по иерархии вверх переходим к более общим классам. При движении вниз – к более специализированным классам.

Простое наследование – это случай, когда у класса есть только один прямой базовый класс.

Множественное наследование, когда у класса есть несколько прямых базовых классов. Множественное наследование создаёт предпосылки к ошибочному использованию наследования. Поэтому C#, как и большинство современных языков программирования, запрещает множественное наследование. Напомним, что наследование, особенно множественное, позволяет рассматривать один объект с разных точек зрения.

Полиморфизм с литературной точки зрения означает много форм или много обликов. Это концепция, по которой один и тот же метод, определенный в базовом классе, может быть по-разному определен в разных классах потомках. Появляется новая проблема: как работать методу у объекта базового класса? Есть возможность не определять метод в базовом классе, т.е. оставить тело метода пустым. Такие методы называют *операциями.*

В типичной иерархии классов операции объявляются в базовом классе, а определяются различными путями в различных классах потомках. Базовый класс представляет имя метода в иерархии. В случае, когда метод не определен в базовом классе, нельзя создавать объекты этого класса, так как для этого объекта будет не определен этот метод. Такие классы называются *абстрактными.*

Абстрактные классы и интерфейсы похожи, так как не могут иметь объектов. Отличие между ними в том, что абстрактный класс может содержать определения методов, *интерфейсы* содержат только операции (имена методов). То есть интерфейсы абстрактнее абстрактных классов.

Когда вызываем метод напрямую из объекта, а не из операции базового класса, то метод связывается с вызовом при компиляции – *раннее связывание*. При вызове метода не напрямую через объект, а через операцию базового типа он вызывается при выполнении программы – *позднее связывание.* Гибкость позднего связывания обеспечивается физической и логической ценой. Позднее связывание выполняется дольше, чем раннее. При позднем связывании классы-потомки могут заменять базовые классы. Операции могут быть вызваны из интерфейса, а классы-потомки обеспечат правильное выполнение.

Вопросы к п. 3.7

1. Объясните концепцию абстракции. Почему она важна для программной инженерии?

2. Назовите два принципа инкапсуляции.

3. Опишите наследование в контексте ООП.

4. Что такое полиморфизм? Как он связан с ранним и поздним связыванием?

5. Опишите разницу между интерфейсами, абстрактными классами и конкретными классами.

Задания для самостоятельной работы

1. Создать класс счет в банке с закрытыми полями: номер счета, баланс, тип банковского счета (использовать перечислимый тип). Предусмотреть методы для доступа к данным – заполнения и чтения. Создать объект класса, заполнить его поля и вывести информацию об объекте класса на печать.

2. Изменить класс счет в банке из задания 1 таким образом, чтобы номер счета генерировался сам и был уникальным. Для этого надо создать в классе статическую переменную и метод, который увеличивает значение этой переменной.

3. Добавить в класс счет в банке два метода: снять со счета и положить на счет. Метод снять со счета проверяет, возможно ли снять запрашиваемую сумму, и в случае положительного результата изменяет баланс.

3. 8 Использование ссылочных типов данных

C# поддерживает типы данных, такие как int, long и bool, эти типы данных называются значимыми (value type). Переменные такого типа непосредственно содержат значение в самой переменной. Кроме этого, есть также ссылочные типы данных (reference type). Переменные такого типа содержат ссылку на некоторую область памяти с данными. К ссылочным типам данных относятся встроенные в платформу .NET такие типы, как Array, string, классы.

Для того чтобы объявить переменную ссылочного типа данных, используется следующий код:

coordinate c1;

Данная строка только объявляет переменную c1, которая может содержать ссылку на объект типа coordinate. Строка

c1=new coordinate();

создает новый объект и возвращает ссылку на него, которую сохраняет в переменной c1. После создания объекта c1 можно получить доступ к полям этого объекта, используя оператор .(точку), например:

c1.x=10; c1.y=5;

Чтобы освободить ссылочную переменную, ей явно надо присвоить значение null.

При попытке доступа к полю непроинициализированной ссылочной переменной может возникнуть ошибка на этапе компиляции или будет сгенерировано исключение во время выполнения программы. Например, если

сначала проинициализировали переменную, потом присвоили ей null, а потом попытались получить доступ к полю этого объекта, то в этом случае возникнет исключительная ситуация типа NullReferenceException во время выполнения программы.

Операторы проверки равенства (==) и (!=) для ссылочных переменных будут проверять, ссылаются ли две переменные на один и тот же объект – на одну и ту же область памяти. Для типа данных string оператор (==) можно использовать для проверки равенства значений (строк) в двух строковых переменных. Для ссылочных переменных нельзя использовать операторы отношения (>,<.>=,>=).

Две ссылочных переменных могут ссылаться на одну и ту же область памяти. Соответственно изменение значения объекта через одну ссылку изменит значение объекта через другую ссылку.

Ссылочные переменные можно передавать в качестве параметров в методы. В случае передачи параметра по значению будет создана копия ссылки на тот же объект внутри метода. При передаче по ссылке (ключевое слово ref) используется одна ссылка внутри и снаружи метода. При передаче out параметра используется одна ссылка снаружи и внутри метода, но в отличие от передачи по ссылке в данном случае ссылочная переменная должна быть инициализирована внутри метода. При передаче ссылочной переменной в качестве параметра в метод любым из трех способов изменение значения объекта внутри метода приведет к изменению значения объекта вне метода.

Рассмотрим несколько определенных в .NET ссылочных типов данных.

1. Класс **Exception**. Объекты этого класса и только они, используются в операторе генерирования исключительной ситуации throw и операторе обработки исключительной ситуации catch. Кроме того, в C# есть множество классов, наследников от класса Exception, для разного рода исключений.

2. Класс **String** – последовательность неизменяемых символов в кодировке Unicode. Все методы, которые работают со строкой и, казалось бы, изменяют ее, на самом деле создают новый объект класса String и возвращают его в качестве результата. Некоторые методы, определенные в классе:

string str=”alphabet”;

char c=str[5]; //возвращает шестой символ строки;

str[3]=’z’;//приведет к ошибке, так как свойство взятие индекса только для чтения;

int n=str.Length; //возвращает длину строки;

string str1=str.Insert(2,”ABC”); //добавляет во вторую позицию строку ABC и результат возвращает в str1. Значение строки str1 будет равно “aABCalphabet”, значение строки str останется без изменения (str=”alphabet”);

string str2=String.Copy(str); //создается новая строка str2 и в нее копируется значение строки str;

string s=String.Concat(“a”,”b”,”c”); // операция конкатенации эквивалента операции:

string s=”a”+”b”+”c”;

string s=str.Trim(); //создает новый объект String на основе строки str, из которой удалены специальные символы и пробелы;

string s=str.ToUpper(); // переводит строку к верхнему регистру;

string s=str.ToLower(); //переводит строку к нижнему регистру.

Для сравнения значения двух строковых переменных между собой можно использовать оператор == и !=. Также можно использовать метод **Equals** s1.Equals(s2); //вернет true, если строки совпадают;

String.Equals(s1,s2); //вернет true, если строки совпадают.

Метод **Compare** сравнивает две строки в соответствии с их лексикографическим порядком. Метод возвращает отрицательное значение, если первая строка меньше второй; число ноль – если две строки совпадают; положительное число – если первая строка больше второй.

String.Compare(s1,s2);

Метод Compare можно также использовать с тремя параметрами. Третий параметр типа bool, если он равен true, то метод не чувствителен к регистру.

3. Класс **StringBuilder** – класс для хранения строк. В отличие от класса String любой метод этого класса изменяет саму строку, не создавая новый объект класса.

Все классы в C# явным или неявным образом наследуются от класса System.Object (часто вместо полного имени класса используется псевдоним object). У класса Object есть общие методы, которые наследуются всеми классами:

– **ToString** – возвращает строковое представление объекта. Реализация по умолчанию вернет имя типа класса. У каждого класса наследника этот метод можно переопределить.

– **Equals** – определяет, указывают ли ссылки на один и тот же объект. Метод можно перегружать, например, для проверки на равенство значений полей объектов.

– **Finalize** – метод вызывается системой во время выполнения, когда объект становится недоступным.

– **GetType** – позволяет во время выполнения получить информацию о типе.

Reflection (рефлексия)

Процесс получения информации о типе во время выполнения называется рефлексией. В пространстве имен System.Reflection содержатся классы и интерфейсы, которые позволяют получить информацию о типах, методах и полях. Класс System.Type содержит методы для получения информации об объявлении типа: о конструкторах, полях, методах, событиях и свойствах класса. Для получения информации о типе можно использовать оператор typeof:

```
using System;
using System.Reflection;
Type t = typeof(byte);
```

```
Console.WriteLine("Type: {0}", t);
```

Чтобы получить более подробную информацию о классе, можно использовать метод GetMethods у объекта класса Type:

```
using System;
using System.Reflection;
Type t = typeof(string); // Get type information
MethodInfo[ ] mi = t.GetMethods( );
foreach (MethodInfo m in mi) {
Console.WriteLine("Method: {0}", m);
}
```

Оператор typeof можно применять только тогда, когда объект существует на этапе компиляции. Если необходимо информацию о типе получить во время выполнения программы, следует использовать метод GetType класса Object.

Пространства имен

В платформе .NET Framework реализована большая коллекция классов, которые предоставляют интерфейс к общеязыковой среде исполнения, к операционной системе и сети. Все классы сгруппированы в пространства имен.

Пространство имен System.IO содержит классы для работы с операциями ввода/вывода, для работы с файловой системой.

Классы File и Directory предоставляют методы для создания, удаления и управления директориями и файлами в файловой системе.

Классы StreamReader и StreamWriter позволяют программе получать доступ к содержимому файлов как к потоку битов или символов.

Класс FileStream содержит операции чтения и записи в файл, открытия и закрытия файлов в файловой системе, а также методы для изменения других дескрипторов операционной системы для обработки файлов, позволяет задать синхронное или асинхронное выполнение операций чтения и записи.

Классы BinaryReader и BinaryWriter позволяет читать и записывать простые типы данных как двоичные значения в заданной кодировке.

Пространство имен System.Data содержит классы для работы с данными из различных источников. Класс DataSet предназначен для работы с данными из разных источников. Класс DataSet состоит из таблиц (класс DataTable). Пространство имен System.Data.SqlClient содержит классы, которые предоставляют прямой доступ к базе данных SQL Server. Для доступа к другим реляционным БД используются классы из System.Data.OleDb.

Пространство имен System содержит классы для работы со значимыми типами данных и ссылочными типами данных, событиями, делегатами, интерфейсами, атрибутами. Предоставляет методы для преобразования типов данных, управления программой.

Пространство имен System.Net представляет простой программный интерфейс для работы с сетью с помощью различных протоколов.

Пространство имен System.Windows.Forms – классы для создания графического интерфейса в Windows приложениях.

Приведение типов данных

Для значимых типов данных различают явное и неявное преобразование. Неявное преобразование происходит тогда, когда переменную одного типа данных пытаются присвоить в переменную другого типа. В C# неявное преобразование разрешается обычно тогда, когда значение может быть преобразовано к другому типу без потери значимости. Например, от int к long:

```
int a=4;
long b;
b=a;
```

При явном преобразовании типов данных используется оператор cast. Если в процессе преобразования возникли проблемы, генерируется исключительная ситуация:

```
try{
        a=checked((int)b);
}
catch (Exception e){
        Console.WriteLint("Problem in cast");
}
```

Все преобразования между различными базовыми типами данных осуществляются классом System.Convert.

Можно преобразовать объект класса потомка к объекту родительского класса и наоборот. Преобразование к родительскому классу возможно всегда:

```
Animal a;
Bird b=new Bird(); //класс Animal – класс предок для класса Bird
a=b;
```

Можно использовать оператор приведения типов:

```
a=(Animal) b;
```

Преобразование к классу потомка необходимо осуществлять явно:

```
Bird b=(Bird) a;
```

В данном случае в процессе работы программы будет выполнена проверка, действительно ли объект *a* типа Bird. Если это не так, будет сгенерирована исключительная ситуация типа InvalidCastException. Код, осуществляющий явное приведение типов данных, лучше размещать в блоке try-catch.

В C# есть возможность проверить, можно ли преобразовать объект одного типа данных к другому типу данных. Для этого используется оператор is, который возвращает true или false в зависимости от того, возможно ли приведение типов:

```
if (a is Bird)
        b=(Bird)a; //Приведение типов возможно
else
        Console.WriteLine("a не является птицей");
```

Оператор as преобразует объект одного типа данных к другому, если

преобразование возможно, и возвращает null, если преобразование не возможно:

```
Bird b=a as Bird;
 if (b==null)
        Console.WriteLine("a не является птицей");
```

В .NET все ссылочные типы данных наследуют от класса object, поэтому объект любого ссылочного типа данных можно привести к типу данных object.

Преобразования типов можно выполнять, работая с интерфейсами:

```
IHashCodeProvider hcp;
hcp = (IHashCodeProvider) x;
```

При помощи оператора is можно узнать, определяет ли класс интерфейс

```
if (x is IHashCodeProvider) ...
```

Также можно использовать оператор as вместо оператора преобразования

```
IHashCodeProvider hcp;
hcp = x as IHashCodeProvider;
```

C# может переводить структурные типы в ссылочные и наоборот (boxing и unboxing).

```
static void Show(object o)
{
        Console.WriteLine(o.ToString( ));
}
Show(42);
```

В случае явного преобразования типов:

```
object o = (object) 42; // Box Console.WriteLine(o.ToString( )).
```

Вопросы к п. 3.8

1. Как распределяется память для переменных ссылочного типа?
2. Какое значение присваивают ссылочной переменной, чтобы показать, что она не указывает на объект? Что произойдет, если обратиться к ней как к объекту?
3. Какой класс является базовым для всех классов C#?
4. Объясните разницу между операцией преобразования типа (cast) и оператором as.

Задания для самостоятельной работы

Задания на использование переменных ссылочного типа, передачу их в качестве параметров методам, преобразование типов данных.

1. В класс банковский счет, созданный в задании 1 п. 3.7 добавить метод, который переводит деньги с одного счета на другой. У метода два параметра: ссылка на объект класса банковский счет, откуда снимаются деньги, второй параметр – сумма.
2. Реализовать метод, который в качестве входного параметра принимает строку string, возвращает строку типа string, буквы в которой идут в обратном

порядке. Протестировать метод.

3. Написать программу, которая спрашивает у пользователя имя файла. Если такого файла не существует, то программа выдает пользователю сообщение и заканчивает работу, иначе в выходной файл записывается содержимое исходного файла, но заглавными буквами.

4. Реализовать метод, который проверяет, реализует ли входной параметр метода интерфейс System.IFormattable. Использовать оператор is и as. (Интерфейс IFormattable обеспечивает функциональные возможности форматирования значения объекта в строковое представление.)

5. Работа со строками. Дан текстовый файл, содержащий ФИО и e-mail-адрес. Разделителем между ФИО и адресом электронной почты является символ #:

Иванов Иван Иванович # iviviv@mail.ru

Петров Петр Петрович # petr@mail.ru

Сформировать новый файл, содержащий список адресов электронной почты. Предусмотреть метод, выделяющий из строки адрес почты. Методу в качестве параметра передается символьная строка s, e-mail возвращается в той же строке s:

public void SearchMail (ref string s).

6. Список песен. В методе Main создать список из четырех песен. В цикле вывести информацию о каждой песне. Сравнить между собой первую и вторую песни в списке.

Песня представляет собой класс с методами для заполнения каждого из полей, методом вывода данных о песне на печать, методом, который сравнивает между собой два объекта:

```
class Song{
        string name; //название песни
        string author; //автор песни
        Song prev; //связь с предыдущей песней в списке
        //метод для заполнения поля name
        //метод для заполнения поля author
       //метод для заполнения поля prev
       //метод для печати названия песни и ее исполнителя
       public string Title(){… /*возвращ название+исполнитель*/ …}
      //метод, который сравнивает между собой два объекта-песни:
      public bool override Equals(object d){…}
}
```

3.9 Создание и удаление объектов

В этом пункте изучим, что происходит при создании объекта, как конструкторы инициализируют объекты и как используются деструкторы для уничтожения объектов. Узнаем, что происходит с объектом после уничтожения и как работает сборщик мусора.

Использование конструкторов

Конструкторы – это специальные методы, которые используются для инициализации объектов при создании. Если конструктор не описан в классе, то запускается конструктор по умолчанию.

Процесс создания объекта проходит в два этапа, но записывается всё в одно выражение. Первый шаг – это выделение памяти, за это отвечает оператор **new**. Второй шаг – инициализация объекта при помощи конструктора, в этот момент полям класса присваиваются начальные значения, либо те, которые указаны в конструкторе, либо значения по умолчанию (для чисел – это ноль, для ссылочных типов данных – это null).

При создании объекта создаётся конструктор по умолчанию, если не создан свой. Иногда конструктор по умолчанию не подходит для инициализации объекта, тогда можно определить свой конструктор. Есть несколько причин, когда не подходит конструктор по умолчанию: не подходит доступ public, инициализация нулями неправильна, невидимый код тяжело понимать.

Все поля, которые не определены в конструкторе, инициализируются нулями. Если конструктор отработал, то с объектом можно работать, если произошла ошибка в конструкторе, то объект не создался.

Конструкторы, как и любые другие методы, можно перегрузить.

```
class Overload
{
        public Overload( ) { this.data = -1; }
        public Overload(int x) { this.data = x; } private int data;
}
class Use
{
      static void Main( )
      {
            Overload o1 = new Overload( );
            Overload o2 = new Overload(42);
      }
}
```

Если определен свой конструктор, то компилятор не создаст конструктор по умолчанию и его придется прописать самостоятельно.

При определении конструкторов можно использовать конструкцию, называемую списком инициализации. Определение одного конструктора при помощи вызова другого перегруженного конструктора.

```
class Date
{
        public Date() : this(1970, 1, 1) {   }
        public Date(int year, int month, int day) {   }
```

```
}
```

Ограничения – нельзя вызывать конструктор со списком инициализации из другого метода

```
class Point
{
        public Point(int x, int y) { ... }
        public void Init( ) : this(0, 0) { }  // Ошибка при компиляции
}
```

нельзя вызывать самого себя

```
class Point
{               // Ошибка при компиляции
      public Point(int x, int y) : this(x, y) { }
}
```

нельзя использовать ключевое слово this в списке инициализации.

```
class Point
{     // Ошибка при компиляции
        public Point( ) : this(X(this), Y(this)) { }
        public Point(int x, int y) { ... }
        private static int X(Point p) { ... }
        private static int Y(Point p) { ... }
}
```

При определении конструкторов необходимо определить константы и поля только для чтения. Поля, которые не могут быть переприсвоены, называются полями только для чтения. Есть три варианта инициализации полей только для чтения:

– нулём неявно;

– присвоением в конструкторе, что разрешено;

– присвоением при объявлении поля в классе.

```
class SourceFile
{
      public SourceFile( ) {     }
      private readonly ArrayList lines = new ArrayList( );
}
```

Синтаксис конструкторов одинаков и для структур. Отличие в том, что для структур необходимо инициализировать все поля.

Закрытый (private) конструктор – это особый конструктор экземпляров. Обычно он используется в классах, содержащих только статические элементы.

Если в классе один или несколько закрытых конструкторов и ни одного открытого конструктора, то прочие классы (за исключением вложенных классов) не смогут создавать экземпляры этого класса. Объявление пустого конструктора запрещает автоматическое создание конструктора по умолчанию. Стоит заметить, что если не использовать с конструктором модификатор доступа, то по умолчанию он все равно будет закрытым. Однако обычно

используется модификатор private, чтобы ясно обозначить невозможность создания экземпляров данного класса.

Закрытые конструкторы используются, чтобы не допустить создание экземпляров класса при отсутствии полей или методов экземпляра, например для класса Math, или когда осуществляется вызов метода для получения экземпляра класса.

```
class Math
{
        public static double Cos(double x) { ... }
        public static double Sin(double x) { ... }
        private Math( ) { ... }
}
class LessCumbersome
{
        static void Main( )
        {
                double answer = Math.Cos(42.0);
        }
}
```

Достоинства статичных методов – простота и быстрота, не надо создавать объект. Статичный конструктор вызывается при загрузке класса в память.

```
class Example
{
    static Example( ) { ... }
}
```

Уничтожение объектов

В приложении необходимо знать, что случается, когда объект выходит из области видимости или уничтожается. Процесс уничтожения объекта в C# состоит из двух шагов: деинициализация объекта и возвращение памяти в управляемую кучу.

Отметим различия во времени жизни переменных структурных типов и объектов. Переменные структурного типа обычно имеют короткое время жизни, ограниченное блоком, в котором они объявлены. Также их отличает детерминированное создание и уничтожение. Для объектов время жизни дольше и время уничтожения недетерминированное.

В C# нельзя вручную удалить объект, так как эта возможность вызывала много различных ошибок в других языках: забывание удаления объектов, попытка уничтожения объекта дважды, удаление активного объекта.

Сборщик мусора удаляет объекты за программиста. Сборщик гарантирует, что объекты будут удалены, причем только один раз. Удаляются только недостижимые объекты.

```
class Example
```

```
{
        void Method(int limit)
        {
                for(int i = 0; i < limit; i++){
                        Example eg = new Example();
                        ….
                 }
                // eg за пределами блока, существует ли он ещё?
        }
}
```

Как было указано, удаление объекта – двухшаговый процесс. На первом шаге память очищается, на втором возвращается в кучу. Второй шаг одинаков для всех классов, а вот первый индивидуален для каждого класса. Для описания первого шага определяется деструктор или метод Finalize. Можно описать деструктор для определения очистки объекта. В C# нельзя вручную вызвать деструктор или метод Finalize.

В C# программист не знает, когда будет уничтожен объект, только известно, что это произойдёт после того, как он станет недостижимым. Порядок и время вызова деструкторов неопределенны. Желательно избегать деструктора, так как он использует ресурсы сборщика мусора. Если в классе нет неуправляемых частей, то сборщик мусора сам уничтожит объекты и деструктор не нужен.

Память от удаленного объекта освободится, когда сборщик мусора уничтожит объект, однако, кроме памяти объект может содержать другие ресурсы, которые лучше освободить быстрее: подключение к БД, открытые файловые потоки. Для этого применяют метод Dispose, для его использования необходимо, чтобы класс определял интерфейс IDisposable, и, описав метод Dispose, убедиться что метод не вызывается дважды.

Оператор using определяет область видимости – объект, в конце этого блока для объекта вызывается метод Dispose.

```
Resource r1 = new Resource( );
try {
        r1.Test( );
}
finally {
        if (r1 != null) ((IDisposable)r1).Dispose( );
}
```

Рассмотрим более подробно как происходит работа с памятью в языке C#.

При создании объекта память размещается в управляемой куче, и переменная хранит только ссылку на расположение объекта. Для типов

в управляемой куче требуются служебные данные и при их размещении, и при их удалении функциональной возможностью автоматического управления памятью среды CLR, также известной как сборка мусора. Сборка мусора в высокой степени оптимизирована, и в большинстве сценариев она не создает проблем с производительностью.

Автоматическое управление памятью является одной из служб, которые предоставляет среда CLR во время управляемого выполнения. Сборщик мусора среды CLR управляет освобождением и выделением памяти для приложения. Для разработчиков это означает, что при разработке управляемого приложения не нужно писать код для управления памятью. Автоматическое управление памятью позволяет устранить распространенные проблемы, такие как не освобожденный по забывчивости объект, вызывающий утечку памяти, или попытки доступа к памяти для уже удаленного объекта.

Выделение памяти. При инициализации нового процесса среда выполнения резервирует для него непрерывную область адресного пространства. Это зарезервированное адресное пространство называется управляемой кучей. Эта управляемая куча содержит указатель адреса, с которого будет выделена память для следующего объекта в куче. Изначально этот указатель устанавливается в базовый адрес управляемой кучи. Все ссылочные типы размещаются в управляемой куче. Когда приложение создает первый ссылочный тип, память для него выделяется, начиная с базового адреса управляемой кучи. При создании приложением следующего объекта сборщик мусора выделяет для него память в адресном пространстве, непосредственно следующем за первым объектом. Пока имеется доступное адресное пространство, сборщик мусора продолжает выделять пространство для новых объектов по этой схеме.

Выделение памяти из управляемой кучи происходит быстрее, чем неуправляемое выделение памяти. Поскольку среда выполнения выделяет память для объекта путем добавления значения к указателю, это осуществляется почти так же быстро, как выделение памяти из стека. Кроме того, поскольку выделяемые последовательно новые объекты и располагаются последовательно в управляемой куче, приложение может получать доступ к объектам очень быстро.

Освобождение памяти. Механизм оптимизации сборщика мусора определяет наилучшее время для выполнения сбора, основываясь на произведенных выделениях памяти. Когда сборщик мусора выполняет очистку, он освобождает память, выделенную для объектов, которые больше не используются приложением. Он определяет, какие объекты больше не используются, основываясь на корнях приложения. Каждое приложение имеет набор корней. Каждый корень либо ссылается на объект, находящийся в управляемой куче, либо имеет значение NULL. Корни приложения содержат указатели глобальных и статических объектов, локальные переменные и параметры ссылочных объектов в стеке потока, а также регистры процессора. Сборщик мусора имеет доступ к списку активных корней, которые

поддерживаются JIT-компилятором и средой выполнения. С помощью этого списка он проверяет корни приложения и в процессе проверки создает граф, содержащий все объекты, к которым можно получить доступ из этих корней.

Объекты, не входящие в этот граф, являются недостижимыми из данных корней приложения. Сборщик мусора считает недостижимые объекты мусором и будет освобождать выделенную для них память. В процессе очистки сборщик мусора проверяет управляемую кучу, отыскивая блоки адресного пространства, занятые недостижимыми объектами. При обнаружении недостижимого объекта он использует функцию копирования памяти для уплотнения достижимых объектов в памяти, освобождая блоки адресного пространства, выделенные под недостижимые объекты. После уплотнения памяти, занимаемой достижимыми объектами, сборщик мусора вносит необходимые поправки в указатель, чтобы корни приложения указывали на новые расположения объектов. Он также устанавливает указатель управляемой кучи в положение после последнего достижимого объекта. Память уплотняется, только если при очистке обнаруживается значительное число недостижимых объектов. Если после сборки мусора все объекты в управляемой куче остаются на месте, то уплотнение памяти не требуется.

Для повышения производительности среда выполнения выделяет память для больших объектов в отдельной куче. Сборщик мусора автоматически освобождает память, выделенную для больших объектов. Однако для устранения перемещений в памяти больших объектов эта память не сжимается.

Поколения и производительность. С целью оптимизации производительности сборщика мусора управляемая куча подразделяется на три поколения: 0, 1 и 2. Сборщик мусора среды выполнения хранит новые объекты в поколении 0. Для созданных ранее объектов, оставшихся после сборок мусора, их уровень повышается, и они переводятся в поколения 1 и 2. Поскольку быстрее сжать часть управляемой кучи, чем всю кучу, эта схема позволяет сборщику мусора освобождать память в определенном поколении, а не освобождать память для всей кучи каждый раз при сборке мусора.

В действительности сборщик мусора выполняет очистку при заполнении поколения 0. Если приложение пытается создать новый объект, когда поколение 0 заполнено, сборщик мусора обнаруживает, что в поколении 0 не осталось свободного адресного пространства для объекта. Сборщик мусора выполняет сборку, пытаясь освободить для этого объекта адресное пространство в поколении 0. Сборщик мусора начинает проверять объекты в поколении 0, а не все объекты в управляемой куче. Это наиболее эффективный подход, поскольку, как правило, новые объекты имеют меньшее время жизни и можно ожидать, что многие из объектов в поколении 0 к моменту проведения сборки мусора уже не используются приложением. Кроме того, сборка мусора только в поколении 0 зачастую освобождает достаточно памяти для того, чтобы приложение могло продолжить создавать новые объекты.

После того как сборщик мусора выполнит освобождение для поколения 0, он уплотняет память для достижимых объектов. Затем сборщик мусора

повышает уровень этих объектов и считает эту часть управляемой кучи поколением 1. Поскольку объекты, оставшиеся после сборок мусора, как правило, имеют большее время жизни, имеет смысл повысить их уровень до более старшего поколения. В результате сборщику мусора не обязательно выполнять повторную проверку объектов поколений 1 и 2 при каждой сборке мусора в поколении 0.

После того как сборщик мусора выполнил свою первую очистку для поколения 0 и повысил уровень достижимых объектов до поколения 1, он считает оставшуюся часть управляемой кучи поколением 0. Он продолжает выделять память для новых объектов в поколении 0 до тех пор, пока оно не заполнится и не появится необходимость в следующей сборке мусора. В этот момент оптимизатор сборщика мусора определяет, есть ли необходимость проверки объектов в более старых поколениях. Например, если при очистке поколения 0 не освободится достаточно памяти для того, чтобы приложение смогло успешно завершить свою попытку создания нового объекта, сборщик мусора может выполнить очистку поколения 1, а затем - поколения 0. Если и при этом не освобождается достаточно памяти, он может выполнить очистку поколений 2, 1 и 0. После каждой сборки мусора сборщик уплотняет объекты в поколении 0 и продвигает их в поколение 1. Объекты поколения 1, оставшиеся после сборки мусора, продвигаются в поколение 2. Поскольку сборщик мусора поддерживает только три поколения, объекты поколения 2, оставшиеся после сборки мусора, остаются в этом поколении до тех пор, пока при очередной очистке они не будут определены как недостижимые.

Освобождение памяти для неуправляемых ресурсов. Для большинства объектов, созданных приложением, сборщик мусора автоматически выполнит необходимые задачи по управлению памятью. Однако для неуправляемых ресурсов требуется явная очистка.

Основным типом неуправляемых ресурсов являются объекты, образующие упаковку для ресурсов операционной системы, такие как дескриптор файлов, дескриптор окна или сетевое подключение. Хотя сборщик мусора может отслеживать время жизни управляемого объекта, инкапсулирующего неуправляемые ресурсы, он не имеет определенных сведений о том, как освобождать эти ресурсы. При создании объекта, инкапсулирующего неуправляемый ресурс, рекомендуется предоставлять необходимый код для очистки неуправляемого ресурса в общем методе Dispose. Предоставление метода Dispose дает возможность пользователям объекта явно освобождать память при завершении работы с объектом.

Вопросы к п.3.9

1. Как происходит создание и удаление объектов?
2. В каких случаях используются закрытые конструкторы?
3. Возможна ли перегрузка конструкторов и деструкторов? Приведите примеры.
4. Какой метод вызывает сборщик мусора, даже если память ещё не

переполнена?

5. В чем смысл использования оператора *using*?

Задания для самостоятельной работы

Задания на создание конструкторов, деструкторов, обращение к сборщику мусора

1. В классе банковский счет, созданном в предыдущих упражнениях, удалить методы заполнения полей. Вместо этих методов создать конструкторы. Переопределить конструктор по умолчанию, создать конструктор для заполнения поля баланс, конструктор для заполнения поля тип банковского счета, конструктор для заполнения баланса и типа банковского счета. Каждый конструктор должен вызывать метод, генерирующий номер счета.

2. Создать новый класс BankTransaction, который будет хранить информацию обо всех банковских операциях. При изменении баланса счета создается новый объект класса BankTransaction, который содержит текущую дату и время, добавленную или снятую со счета сумму. Поля класса должны быть только для чтения (readonly). Конструктору класса передается один параметр – сумма.

В классе банковский счет добавить закрытое поле типа System.Collections.Queue, которое будет хранить объекты класса BankTransaction для данного банковского счета; изменить методы снятия со счета и добавления на счет так, чтобы в них создавался объект класса BankTransaction и каждый объект добавлялся в переменную типа System.Collections.Queue.

3. В классе банковский счет создать метод Dispose, который данные о проводках из очереди запишет в файл. Не забудьте внутри метода Dispose вызвать метод GC.SuppressFinalize, который сообщает системе, что она не должна вызывать метод завершения для указанного объекта.

3.10 Наследование в C#

Наследование – это свойство объектно-ориентированной системы наследовать данные и функциональность базового класса. Можно в класс-потомок к методам и полям родительского класса добавить необходимые поля и методы. Класс-потомок может замещать методы родительского класса. Надо помнитьчто при изменении родительского класса класс-потомок может оказаться не рабочим.

Наследование от класса называется расширением базового класса. Если класс А наследует от класса В, то класс А называется потомком, а В – предком. Синтаксически это пишется следующим образом:

class A:B {…}

Класс-потомок наследует все элементы базового класса, кроме конструктора и деструктора. Все public элементы базового класса остаются неявно public в потомке, private элементы, хоть и наследуются, но доступны

только для объектов базового класса. Класс-потомок не может быть более доступным, чем базовый класс.

```
class Example
{
        private class NestedBase { }
        public class NestedDerived: NestedBase { } // Ошибка
}
```

Класс-потомок имеет доступ ко всем protected полям и методам родительского класса, класс не являющийся потомком, доступа к protected-членам не имеет.

```
class Token
{
        protected string name;
}
class CommentToken:Token
{
        public string Name()
        {
                return name;  // Доступ разрешен
        }
}

class CommentToken: Token
{
         void Fails(Token t)
         {
                Console.WriteLine(t.name); // Ошибка при компиляции
          }
}
```

Для вызова конструктора базового класса из класса-потомка используется ключевое слово base. Вызов конструктора базового класса пишется после заголовка конструктора класса-потомка через двоеточие:

```
A(список параметров): base(параметры для передачи в конструктор базового класса)
{
         //тело конструктора у класса-потомка
}
```

Ключевое слово base обращается к базовому классу. Для конструктора по умолчанию вызов базового конструктора производится по умолчанию, то есть его можно не прописывать. Для конструкторов не по умолчанию необходимо явно вызывать конструктор базового класса.

```
class Token
{
```

```
        protected Token(string name) { ... }
    }
    class CommentToken: Token
    {
        public CommentToken(string name) {...} //ошибка при компиляции
    }
```

В примере выше получим ошибку при компиляции, так как будет вызываться конструктор по умолчанию, но он отсутствует в классе Token. Для правильной работы необходимо записать конструктор в следующем виде:

```
public CommentToken(string name) : base(name) { ... }
```

Если конструктор базового класса private, то нельзя создавать конструктор класса-потомка.

```
class NonDerivable
{
      private NonDerivable(){  }
}
class Impossible: NonDerivable
{
      public Impossible() {     }   // Ошибка при компиляции
}
```

В классе-потомке можно переопределять методы базового класса, если они для этого предназначены. Виртуальные методы можно полиморфно переопределять в классах-потомках. В C# по тому, содержит ли базовый класс виртуальные методы, можно определить, был ли этот класс разработан для наследования. Невиртуальный метод имеет только одно определение, одинаковое для всех потомков. Для объявления виртуального метода используется ключевое слово virtual. Виртуальный метод должен содержать тело в базовом классе. Нельзя объявлять виртуальными статичные и private-методы. Статичные методы не могут быть виртуальными, так как полиморфизм – это свойство, относящееся к объектам, а не к классам.

Перегруженные методы могут создаваться только для виртуальных методов. Для задания перегруженных методов используется ключевое слово override. В базовом классе метод объявляется виртуальным virtual, в базовом классе есть тело у этого метода, в классе-потомке этот метод определяется перегруженным override и содержит свое тело – таким образом задается перегруженный метод.

```
class Token
{
      public virtual string Name() {…}
}
class CommentToken : Token
{
      public override string Name() {…}
}
```

Можно перегружать только абсолютно идентичные методы. Должны совпадать: имя метода, тип возвращаемого значения, список параметров, уровень доступа. Перегруженный метод должен быть virtual или override. Нельзя объявлять перегруженный метод одновременно виртуальным, т.е. override virtual – нельзя. Нельзя, чтобы перегруженные методы были статичными или private.

Можно скрыть наследуемый метод в иерархии классов, заменив его новым идентичным методом при помощи ключевого слова new. Метод родительского класса не будет наследоваться потомком и заменится на новый идентичный метод.

```
class Token
{
        public int LineNumber() {…}
}
class CommentToken : Token
{
        new public int LineNumber() {…}
}
```

Ключевое слово new прячет как виртуальные, так и невиртуальные методы, разрешает проблему совпадения имен, скрывает методы с одинаковой сигнатурой.

В C# можно объявить класс ненаследуемым при помощи ключевого слова sealed.

```
public sealed class String {..}
public class MyStr:String; //ошибка – от класса String наследовать нельзя
```

Использование интерфейсов

Интерфейс – синтаксический и семантический шаблон, которого все классы-наследники должны придерживаться. Интерфейс говорит, что он умеет делать, классы определяют, как они это делают. Интерфейс представляет собой класс без какого-либо кода. Все интерфейсы по умолчанию public, модификатор доступа у интерфейсов не используется. У методов также не используется модификатор доступа, по умолчанию методы public. У методов в интерфейсе не должно быть тела, только заголовки методов.

```
interface IToken
{
        public int LineNumber( ){ … }; // Ошибка при компиляции: 1.
        Модификатор доступа у метода public //2. Есть тело метода
}
```

C# позволяет наследовать от одного класса и множества интерфейсов.

Интерфейс может наследовать от многих интерфейсов.

```
interface IToken { ... }
interface IVisitable { ... }
interface IVisitableToken: IVisitable, IToken { ... }
```

```
class Token: IVisitableToken { ... }
```

Класс может быть более доступным, чем интерфейс

```
class Example
{
        private interface INested { }
        public class Nested: INested { } // Разрешено
}
```

Класс должен определить все методы всех интерфейсов, от которых он наследует как напрямую, так и косвенно. Метод интерфейса, определяемый классом, должен быть идентичен, то есть должны совпадать параметр доступа, имя, возвращаемое значение и список параметров. Интерфейсные методы, реализуемые в классе, могут быть объявлены как virtual. В этом случае классы-наследники могут перегружать эти методы в дальнейшем.

Другой способ реализации интерфейсных методов – явная реализация. При явной реализации необходимо указать полное имя метода: Имя_интерфейса.имя_метода. При явном определении метод не может быть виртуальным, должен отсутствовать модификатор доступа. При вызове метода к нему нет прямого доступа, только через интерфейс.

```
class Token: IToken, IVisitable
{
        string IToken.Name( )
        {
                ...
        }
        private void Example( )
        {
                Name( );              // Ошибка при компиляции
                ((IToken)this).Name( ); // Правильно
        }
        ...
}
```

Явная реализация позволяет:

– исключить определение интерфейса из класса, если он не интересен пользователям класса.

– обеспечивать классу несколько определений различных методов интерфейсов одинаковой сигнатуры.

```
interface IArtist
{
        void Draw();
}
interface ICowboy
{
        void Draw();
```

```
}
class ArtisticCowboy: IArtist, ICowboy
{
        void IArtist.Draw() {…}
        void ICowbowt.Draw() {…}
}
```

Использование абстрактных классов

Абстрактные классы используются для частичной реализации классов, которые могут быть полностью реализованы в конкретных классах-потомках. Абстрактный класс объявляется с помощью ключевого слова abstract. Правила создания абстрактного класса совпадают с правилами создания обычных классов. Однако в абстрактных классах можно объявлять абстрактные методы. Нельзя создавать объекты абстрактного класса. Абстрактный класс может являться наследником неабстрактного класса. Все методы интерфейса, определяемого абстрактным классом, должны быть определены в абстрактном классе.

И абстрактные классы, и интерфейсы предназначены для наследования. Однако класс может наследовать только от одного абстрактного класса. Только абстрактные классы могут иметь абстрактные методы. У абстрактного метода отсутствует тело метода. Абстрактные методы виртуальные, переопределенные абстрактные методы у классов-потомков будут override. Абстрактные методы могут переопределять virtual- и override- методы.

```
class Token
{
        public virtual string Name( ) { ... }
}
abstract class Force: Token
{
        public abstract override string Name( );
}
```

Вопросы к п. 3.10

1. В чем отличие между public-, private- и protected-полями?
2. Как переопределить метод базового класса у класса-потомка?
3. Что такое абстрактный класс?
4. Что такое интерфейс? В чем отличие интерфейса от абстрактного класса?
5. Допустимо ли множественное наследование?

Задания для самостоятельной работы

Задания на наследование, определение и использование интерфейсов, абстрактных классов, виртуальные методы

1. Создать интерфейс ICipher, который определяет методы поддержки

шифрования строк. В интерфейсе объявляются два метода – encode() и decode(), которые используются для шифрования и дешифрования строк соответственно.

Создать класс ACipher, реализующий интерфейс ICipher. Класс шифрует строку посредством сдвига каждого символа на одну «алфавитную» позицию выше. Например, в результате такого сдвига буква А становится буквой Б.

Создать класс BCipher, реализующий интерфейс ICipher. Класс шифрует строку, выполняя замену каждой буквы, стоящей в алфавите на i-й позиции, на букву того же регистра, расположенную в алфавите на i-й позиции с конца алфавита. Например, буква В заменяется на букву Э.

Написать программу, демонстрирующую функционирование классов.

2. Создать класс Figure для работы с геометрическими фигурами. В качестве полей класса задают цвет фигуры, состояние «видимое/невидимое». Реализовать операции: передвижение геометрической фигуры по горизонтали, по вертикали, изменение цвета, опрос состояния (видимый/невидимый). Метод вывода на экран должен выводить состояние всех полей объекта. Создать класс Point (точка) как потомок геометрической фигуры. Создать класс Circle (окружность) как потомок точки. В класс Circle добавить метод, который вычисляет площадь окружности. Создать класс Rectangle (прямоугольник) как потомок точки, реализовать метод вычисления площади прямоугольника.

Точка, окружность, прямоугольник должны поддерживать методы передвижения по горизонтали и вертикали, изменения цвета.

Подумать, какие методы можно объявить в интерфейсе, нужно ли объявлять абстрактный класс, какие методы и поля будут в абстрактном классе, какие методы будут виртуальными, какие перегруженными.

3.11 Агрегации, пространства имен, сборки и модули

Использование внутренних (internal) классов, методов и данных

Модификаторы доступа определяют возможность доступа к элементам класса, таким как методы и свойства. При разработке класса необходимо явно указывать модификатор доступа у каждого члена класса. Модификаторы доступа:

– **Public** – элементы доступны всюду внутри области видимости;

– **Protected** – элементы доступны внутри класса и у всех потомков класса;

– **Private** – элементы доступны только внутри класса;

– **Internal** – элементы доступны внутри одной сборки Microsoft.NET (одного исполняемого файла или одного .dll файла), из других сборок доступ запрещён;

– **Protected internal** – элементы доступны внутри классов-потомков и во всех классах внутри одной сборки.

Public – слишком открытый доступ, private – слишком закрытый, protected – открыт только для потомков класса. Для создания промежуточного типа доступа был создан внутренний (internal) доступ. Типы доступа public и private – логические, то есть не зависят от физического размещения класса. Для internal

классов или элементов размещение определяет область доступа: если класс находится в исполняемом файле, то он доступен для всех классов файла; если класс относится к конкретной сборке, то доступ разрешен только внутри данной сборки. В исполняемом файле может быть несколько сборок, тогда доступ к internal-классу из другой сборки будет запрещен.

Когда класс определен как internal, доступ к нему имеют все классы из сборки. Классы и элементы protected internal доступны всей сборке, а также всем потомкам текущего класса.

Ограничения на использование модификаторов доступа:

1. Вне классов и пространств имен нельзя объявлять классы protected и private, эти типы доступа можно использовать только внутри какого-либо класса. Для элементов внутри класса модификатором доступа по умолчанию будет private. В C# есть возможность внутри одного класса объявлять другой класс.

2. При объявлении типа в глобальной области видимости можно использовать модификаторы доступа public и internal. В глобальной области видимости или в пространстве имен модификатор доступа по умолчанию будет internal.

Использование агрегаций

Агрегации используются для группировки объектов вместе в иерархию объектов, которую можно неоднократно использовать. Агрегации определяют связь целое/часть между объектами, не классами. В одном случае в этой связи время жизни «целого» и «части» могут быть не связаны. Тогда агрегация называется агрегацией по ссылке. Если время жизни «целого» и «части» связаны друг с другом, агрегация называется агрегацией по значению. В агрегации «целое» – это всего лишь класс, который используется для группировки «частей»-классов в единое целое, т.е. «целый» класс реально не существует. Например, что такое компьютер? Это всего лишь имя, которое используется для описания конкретных частей: CPU, монитор, клавиатура и т.д. Через это имя можно получить доступ к методам, которые работают с составными частями.

Сравнение агрегаций и наследования:

– Агрегации определяют связь на уровне объектов, наследование – на уровне классов. В случае агрегации у целого объекта может быть несколько объектов частей одного типа. Например, у агрегации компьютер может быть несколько мониторов, несколько процессоров. При наследовании речь идет не о конкретных объектах, а о классах. В этом случае компьютер и монитор – это просто два разных класса, никак не связанные друг с другом.

– В агрегации при изменении методов «частей» методы «целого» автоматически не меняются. В наследовании же при изменении методов у предков меняются и потомки.

– В агрегации в объект «целого» можно добавлять и удалять объекты «части» без каких-либо ограничений. В главном объекте

сохраняются ссылки на части, при необходимости они могу изменяться, можно создавать новые, удалять старые. В механизме наследования все статично: класс потомок всегда будет оставаться потомком.

Фабрики классов

Иногда бывает необходимо запретить создание объекта класса напрямую, разрешить создавать объекты только определенным объектам некоторого другого класса. Например, самостоятельно нельзя открыть счет в банке. Чтобы открыть счет, нужно пойти в банк и банковский работник открывает счет. В программировании, пусть есть два класса – Bank и BankAccount. Открывать счет в банке – создавать объект класс BankAccount можно только внутри банка, вызвав внутри класса Bank нужный конструктор:

```
public class Bank
{
        public BankAccount OpenAccount( )
        {
                BankAccount opened = new BankAccount( );
                accounts[opened.Number( )] = opened;
                return opened;
        }
        private Hashtable accounts = new Hashtable( );
}
public class BankAccount
{
        internal BankAccount( ) { ... }
        public long Number( ) { ... }
        public void Deposit(decimal amount) { ... }
}
```

В примере для хранения всех открытых в банке счетов внутри класса Bank предусмотрено поле accounts в виде хэш-таблицы. Каждый открытый банковский счет сохраняется в этой хэш-таблице. В данном случае класс Bank будет являться фабрикой. Для того чтобы конструктор класса BankAccount был доступен внутри класса Bank, его объявили с модификатором доступа internal.

Пространства имен

Рассмотрим пример

```
public class Bank
{
        public class Account
        {
                public void Deposit(decimal amount)
                {
                        balance += amount;
                }
```

```
                private decimal balance;
    }
    public Account OpenAccount( ) { ... }
    }
```

В этом примере четыре области видимости:

– глобальная область – в ней располагается класс Bank;

– область класса Bank – в ней класс Account и метод OpenAccount;

– область класса Account – в ней метод Deposit и поле balance;

– область метода Deposit – в ней объявляется параметр amount.

Если объект или метод находится вне области видимости или его имя скрыто, то вызвать его можно только при помощи полного пути.

```
class Top
{
      public void M(){…}
}
class Bottom:Top
{
      new public void M()
      {
            M();          // рекурсия
            base.M();    //обращение к скрытому методу
      }
}
```

При обращении к элементам базового класса используется ключевое слово base. Точно так же и для полей класса:

```
public struct Point
{
        public Point(int x, int y)
        {
            this.x = x;
            this.y = y;
        }
        private int x,y;
}
```

В сложных проектах, в которых используется большое количество классов, когда над проектом работает несколько программистов, возможна ситуация, что в одной области видимости окажутся два класса с одинаковым именем. В таком случае следует использовать пространства имен (namespaces). Пространства имен разделяют большой проект на несколько подсистем, в каждой подсистеме – своя глобальная область видимости. Рекомендуется внутри одного пространства имен размещать логически связанные друг с другом элементы.

```
namespace Outer
{
```

```
        namespace Inner
        {
            class Widget{…}
        }
}
```

В C# можно записать предыдущий класс короче.

```
namespace Outer.Inner
{
    class Widget{…}
}
```

Для того чтобы использовать класс из другого пространства имен, необходимо использовать полное его имя или подключить соответствующее пространство имен, используя директиву using. Например, для обращения к классу Widget из предыдущего примера из другого пространства имен можно написать

```
Outer.Inner.Widget w; //полный путь к классу Widget
```

или

```
using Outer.Inner; //подключаем пространство имен
Widget w; //объявляем объект класса Widget без указания полного пути к классу.
```

Директива using должна располагаться перед объявлением каких-либо элементов либо в глобальной области видимости, либо внутри пространства имен. Директива разрешает доступ по имени к классам только данного пространства имен, то есть доступ к классам вложенных пространств имен по короткому имени запрещен.

```
namespace Microsoft.PowerPoint
{
        public class Widget { ... }
}
namespace VendorB
{
        using Microsoft; // но не Microsoft.PowerPoint
        ass SpecialWidget: Widget { ... } // ошибка на этапе компиляции
}
```

Кроме того, если подключить при помощи директивы два пространства имен с несколькими классами с одним именем, то при вызове этого класса получим ошибку компиляции.

```
namespace Test
{
    using VendoreA;     // содержит класс Widget
    using VendoreB;     // также содержит класс Widget
     // Здесь ошибки нет
    class Application
    {
```

```
        static void Main()
        {
            Widget w = new Widget(); // ошибка при компиляции
        }
    }
}
```

Ошибка возникает, так как компилятор не знает, какой класс нужен. Надо явно прописать необходимое пространство имен, например

VendoreA.Widget w = new VendoreA.Widget();

Директиву using можно использовать для объявления коротких имён классов напрямую.

using Widget = VendoreA.SuiteB.Widget

В этом случае можно сразу обращаться по короткому имени к данному классу. Все ограничения на использование директивы остаются. Директиву using можно использовать в обеих её возможностях одновременно, единственное, что надо помнить, что действие директивы начинается с первого элемента пространства имен, в которой она объявлена, то есть друг на друга директивы не действуют, и следующая конструкция даст ошибку:

using System;
using TheConsole = Console; //ошибка, хотя Console класс из System

Правильно будет:

using System;
using TheConsole = System.Console;

Модули и сборки

В C# присутствует возможность компилировать исходные .cs файлы в управляемый модуль – промежуточный язык MSIL, который содержит метаданные для описания модуля. Для этого в опциях компилятору надо задать параметр /target:module имя_файла.cs. При выполнении программы управляемый модуль преобразовывается в машинный код. При компиляции исходный файл компилируется в управляемый модуль с расширением

.netmodule. Команда командной строки:

csc /target:module Bank.cs

На выходе получим файл Bank.netmodule.

Выполняемый файл может запускать модули только в составе какой-либо сборки. В сборке физически размещается группа взаимодействующих классов. Классы, находящиеся в одной сборке, имеют доступ к internal-элементам друг друга. Для классов из других сборок доступ к этим элементам закрыт.

Сборка – это многократно используемый, безопасный и самоописываемый модуль, содержащий типы и ресурсы с поддержкой версий; это первичный стандартный блок приложения .NET. Сборка состоит из двух логических частей:

– наборов типов и ресурсов, которые формируют некоторый логический

модуль из функциональных возможностей,

– метаданные, которые описывают, как эти элементы связаны и от чего зависит их правильная работа.

Метаданные, которые описывают сборку, называют манифестом. В манифесте содержится следующая информация:

– Уникальность. Уникальность сборки включает в себя простое текстовое название, номер версии и дополнительный открытый ключ, который гарантирует уникальность названия и защищает от нежелательного использования.

– Содержание. Сборка содержит типы и ресурсы. Манифест перечисляет названия всех типов и ресурсов, которые видимы извне сборки. Содержит информацию о том, где они могут быть найдены во время трансляции.

– Ссылки. Каждая сборка явно описывает другие сборки, от которых она зависит.

В простейшем случае сборка состоит из одного файла, который содержит код, ресурсы, метаданные и манифест. В более общем случае, сборка состоит из нескольких файлов, тогда сборка существует как автономный файл или содержится в одном из файлов PE, которые содержат типы и ресурсы.

Соответствующие команды компилятора:

csc /target:library /out:Bank.dll Bank.cs Account.cs

csc /t:library /addmodule:Account.netmodule /out:Bank.dll Bank.cs

Каждая сборка имеет свой уникальный номер версии. Две сборки, которые отличаются только номером, для исполняющей среды различны. Номер версии состоит из четырех частей:

старшая версия . младшая версия . номер компиляции . редакция Например, номер 1.5.12540.0 означает, что старшая версия 1, младшая 5,

номер компиляции 12540 и редакция 0.

Пространства имен – это логический механизм компиляции, он обеспечивает структуру имен сущностей исходного кода. При выполнении программы пространства имен не рассматриваются. Сборки – это физический механизм выполнения, обеспечивающий структуру компонент при исполнении программы. Можно разместить классы одного пространства имен в разных сборках, также в одной сборке могут быть классы разных пространств имен. В сборку можно подключать внешние модули, тогда в сборке записывается именованная ссылка на управляемый модуль.

Вопросы к п. 3.11

1. Пусть есть два cs файла. Файл alpha.cs содержит класс Alpha с internal-методом Method. Файл beta.cs содержит класс Beta с internal методом с тем же названием Method. Может ли Alpha.Method вызвать Beta.Method, и наоборот?

2. Агрегации – это связь объектов или классов?

3. Откомпилируется ли следующий код без ошибок?

```
namespace Outer.Inner
{
        class Wibble{}
}
namespace Test
{
        using Outer.Inner;
        class SpecialWible:Inner.Wible{}
}
```

Задания для самостоятельной работы

Модификатор доступа internal. Задание на агрегацию, пространство имен.

1. Создать новый класс, который будет являться фабрикой объектов класса банковский счет. Изменить модификатор доступа у конструкторов класса банковский счет на internal. Добавить в фабричный класс перегруженные методы создания счета CreateAccount, которые бы вызывали конструктор класса банковский счет и возвращали номер созданного счета. Использовать хеш-таблицу для хранения всех объектов банковских счетов в фабричном классе. В фабричном классе предусмотреть метод закрытия счета, который удаляет счет из хеш-таблицы (методу в качестве параметра передается номер банковского счета).

Использовать утилиту ILDASM для просмотра структуры классов.

2. Разбить созданные классы, связанные с банковским счетом, и тестовый пример в разные исходные файлы. Разместить классы в одно пространство имен и создать сборку. Подключить сборку к проекту и откомпилировать тестовый пример со сборкой. Получить исполняемый файл, проверить с помощью утилиты ILDASM, что тестовый пример ссылается на сборку и не содержит в себе классов, связанных с банковским счетом.

3.12 Операции, делегаты, события

Операции

Представим, что надо сложить между собой четыре комплексных числа класса Complex. Пусть в классе реализован статичный метод Add, который складывает два объекта типа Complex и возвращает объект типа Complex. Тогда для того, чтобы сложить четыре комплексных числа, придется написать следующую строку:

```
Var = Complex.Add( Var1, Complex.Add(Complex.Add ( Var2, Var3) ,
Var4);
```

Если бы для комплексных чисел (в классе Complex) была реализована операция сложения, код выглядел бы гораздо приятнее:

```
Var =Var1+Var2 +Var3 + Var4;
```

Операции в C# определяются как обычные методы. Компилятор и среда исполнения автоматически переводят выражения с операциями в

соответствующую серию вызовов методов.

В C# есть большое количество предопределенных операций. Для различных типов данных одна и та же операция может реализовывать различные действия. Это свойство называется перегрузкой операций.

Операции следует определять только тогда, когда они упрощают выражения и когда класс или структура хранят какие-либо данные, для которых действие этой операции понятно. Например, для класса Employee семантически не понятно действие оператора инкремента(emp++).

Все операции – это public static методы, их имена задаются по шаблону operatorop, где op – знак операции. Для операции сложения – operator+. Список параметров и их типы должны быть определены. Операции возвращают объект класса.

```
public static Time operator+(Time t1, Time t2)
{
        int newHours = t1.hours + t2.hours;
        int newMinutes = t1.minutes + t2.minutes;
        return new Time(newHours, newMinutes);
}
```

Перегрузка операций сравнения производится попарно: < и >, <= и =>, == и !=. При перегрузке одной операции из пары обязательно надо определить и вторую. Кроме того, при перегрузке операций равенства и неравенства необходимо определить виртуальный метод Equals, наследуемый от класса Object, чтобы при сравнении объектов с помощью метода Equals не получить результат, противоположный сравнению операцией равенства (==). Для той же цели перегружается ещё один метод, наследуемый от Object – GetHashCode, для равных объектов хеш-код также должен быть одинаков.

Для логических операций && и || нет прямой перегрузки, для этого используются побитовые операторы. Перегрузив операторы &, |, true и false, можно определить логическое И и логическое ИЛИ. Пусть x и y – переменные типа T, тогда логические операторы определяются следующим образом:

– x && y – будет выражаться через T.false(x) ? x : T.&(x,y), что означает, что если x в терминах класса T является ложью, то результат равен x, иначе в результате получим побитовое И x и y в терминах класса T.

– x || y – выражается через T.true(x) ? x : T.|(x,y), то есть если x в терминах класса T является истинным, то результат равен x, иначе получим побитовое ИЛИ x и y в терминах класса T.

При задании операторов преобразования типов данных необходимо указать явным или неявным образом будет осуществляться преобразование. Для задания явного преобразования используется ключевое слово **explicit**, для неявного – **implicit**:

– public static explicit operator Time (int minutes) – явное преобразование типа int в Time;

– public static explicit operator Time (float minutes) – явное преобразование типа float к Time;

– public static implicit operator int (Time t1) – неявное преобразование типа Time к типу int;

– public static explicit operator float (Time t1) – явное преобразование типа Time к типу float;

– public static implicit operator string (Time t1) – если для класса предусмотрена перегрузка преобразования в строку, то необходимо аналогично перегрузить метод ToString(), унаследованный от класса Object.

Операции можно перегружать несколько раз в зависимости от параметров:

– public static Time operator+ (Time t1, int hours) {…}

– public static Time operator+ (Time t1, float hours) {…}

Создание и использование делегатов

Делегаты позволяют вызывать методы неявно, не используя прямое обращение по имени. Фактически, делегаты предназначены для ситуаций, когда нужно передать метод другому методу. Примером одной из таких ситуаций может быть возникновение события. Например, в программировании графического интерфейса – пользователь нажал на кнопку, сгенерировал событие – необходимо предусмотреть обработку этого события. Написать метод и метод каким-то образом связать с этим событием.

Самый простой способ передать метод в качестве параметра:

void Method1() {…} //объявление Method1

MyMethod(Method1); //вызов MyMethod в качестве параметра, которому передан указатель на Method1

Такой способ возможен для некоторых языков, например для C, C++. С точки зрения идеологии платформы .NET Framework такой прямой подход вызывает проблемы с безопасностью типов и игнорирует тот факт, что в объектно-ориентированном программировании методы редко существуют в изоляции: для вызова метода обычно должен быть создан объект класса. В связи с этим в C# такой подход синтаксически не допустим. Вместо этого, когда нужно передать метод, подробную информацию о нем следует поместить в специальную оболочку – делегат. Делегат содержит подробную информацию о методе.

Сначала дается определение делегата, который нужно будет использовать. Его определение означает сообщение компилятору, какого рода метод будет представлять делегат. Затем необходимо создать экземпляр этого делегата. Синтаксис объявления делегатов:

delegate void VoidOpetation(int x);

В данном случае определен делегат, каждый экземпляр которого может содержать ссылку на метод, принимающий один параметр типа int и возвращающий тип void. При определении делегата ему сообщается полная сигнатура метода, который он может представлять. Таким образом, экземпляр делегата может ссылаться на любой метод любого объекта, если сигнатура этого метода совпадает с сигнатурой делегата.

Синтаксис определения делегата подобен определению метода за исключением того, что за таким определением не следует тело метода, и перед определением используется ключевое слово delegate. Объявление делегата можно поместить в любое место, где может находиться объявление класса, то есть внутри какого-либо класса, либо вне любого класса, либо в пространстве имен объекта высшего уровня. Перед делегатом можно использовать любой модификатор доступа.

Пример:

```
private delegate string MyDelegate(int x);
private string MyMethod (int x){
        return x.ToString();
}
static void Main(string[] args)
{
         MyDelegate myD=new MyDelegate(MyMethod); //создаем делегат и связываем с MyMethod
        Console.WriteLine(myD(5)); //строка вызовет метод MyMethod с параметром 5
}
```

События

События позволяют объектам зарегистрироваться на интересующие его изменения в другом объекте. У события есть отправитель – тот, кто генерирует событие, – и получатель. Получателем события может выступать любое приложение, объект или компонент, который нуждается в уведомлении, когда что-то происходит.

Отправитель события ничего не знает о том, кто его получает. Где-то внутри получателя есть метод, который отвечает за обработку события. Этот обработчик события должен запускаться всякий раз, когда возникает зарегистрированное для него событие. Как раз здесь и нужны делегаты. Поскольку отправитель не знает, кто будет получателем, между ними не может быть прямых ссылок, нужен посредник. Таким посредником является делегат. Отправитель определяет делегат, который будет использован получателем. Получатель регистрирует обработчик события, привязывает метод обработки к событию.

Пример. В Windows-приложении разместим на форме кнопку btnOne и создадим для нее обработчик события нажатия на кнопку. Для этого два раза нажмем на кнопку, откроется файл, в котором будет сгенерирован заголовок метода btnOne_Click:

private void btnOne_Click(object sender, EventArgs e){…}, в который необходимо вписать его реализацию. Также автоматически в коде появится строка

btnOne.Click+=new EventHandler(btnOne_Click);

Это означает, что когда произойдет событие Click для кнопки btnOne, то

должен быть выполнен метод btnOne_Click. **EventHandler** – это делегат, который использует событие для назначения обработчика (btnOne_Click) событию (Click). Для добавления метода к списку делегатов использовался оператор +=.

Событию можно назначить больше одного обработчика. Однако нет никакой гарантии того, в каком порядке будут вызваны методы, с которыми связан делегат. С разными событиями можно связать один и тот же метод. Допустим, на форме есть кнопка btnTwo, свяжем событие Click этой кнопки с методом btnOne_Click:

```
btnTwo.Click+=new EventHandler(btnOne_Click);
```

Делегат EventHandler определен средой .NET и находится в пространстве имен System. Все события, определенные .NET, используют этот делегат. Делегат EventHandler использует два параметра object и EventArgs. Первый параметр – это объект, который сгенерировал событие, в данном примере это кнопка btnOne или btnTwo. Внутри метода с помощью первого параметра можно узнать, кто именно сгенерировал событие. Второй параметр – это объект, содержащий другую полезную информацию, может быть любого типа, унаследованного от класса EventArgs.

Важно отметить, что обработчики событий всегда возвращают void. Рассмотрим, как создавать свои собственные события.

Для определения события отправитель объявляет делегат и связывает его с событием.

```
public delegate void StartPumpCallback(object
    sender, CoreOverheatingEventArgs args);
private event StartPumpCallback CoreOverheating;
```

Здесь CoreOverheatingEventArgs – класс, наследник от класса System.EventArgs для передачи дополнительных параметров:

```
public class CoreOverheatingEventArgs : EventArgs
{
    private readonly int temperature;
    public CoreOverheatingEventArgs(int temperature)
    {
        this.temperature = temperature;
     }
     public int GetTemperature()
    {
        return temperature;
    }
}
```

Подписчики определяют метод, который будет вызываться при возникновении события. Если событие еще не создано, то подписчик определяет делегат, ссылающийся на метод при создании события. Если событие уже существует, подписчик добавляет делегат, вызывающий метод при возникновении события.

```
ElectricPumpDriver ed1 = new ElectricPumpDriver();
PneumaticPumpDriver pd1 = new PneumaticPumpDriver();
…
CoreOverheating = new StartPumpCallback(ed1.StartElectricPump);
```
//делегат, который //указывает на метод StartElectricPump объекта ed1, подписывается на событие //CoreOverhating

```
CoreOverheating += new StartPumpCallback(pd1.SwitchOn);
```
//делегат, который //указывает на метод SwitchOn объекта pd1, также подписывается на событие //CoreOverhating

Для уведомления подписчиков необходимо вызывать событие.

```
public void SwitchOnAllPumps()
{
        CoreOverheatingEventArgs e=new CoreOverheatingEventArgs(37);
        if(CoreOverheating != null) //если на событие кто-то подписан, то генерируем //событие
            CoreOverheating(this,e);    //событие вызовет два метода
        //StaertElectricPump и SwithcOn, в которые передаст параметр this и e
}
```

Если событие происходит, то вызываются все делегаты. Заметим, что сначала надо проверить, есть ли хоть один подписчик, так как при отсутствии подписчиков вызов делегата сгенерирует исключение.

Кроме того, что произошло событие, подписчикам бывает интересно, при каких условиях оно произошло. Для этого события могут передавать параметры. Для передачи параметров событиями в C# выделен отдельный класс System.EventsArg.

Вопросы к п. 3.12

1. Может ли арифметическое присваивание (+=, -=, *=, /= и %=) быть перегружено?
2. В каком случае операция преобразования типа должна быть явной?
3. Что такое делегат?
4. Как можно подписаться на событие?
5. Каким образом можно вызвать метод, подписавшийся на событие?

Задания для самостоятельной работы

Задания на определение операторов сложения, умножения, вычитания, деления, равенства, переопределение методов Equals(), ToString(), GetHashCode(); публикацию событий, передачу параметров событиям.

1. Для класса банковский счет переопределить операторы == и != для сравнения информации в двух счетах. Переопределить метод Equals аналогично оператору ==, не забыть переопределить метод GetHashCode(). Переопределить метод ToString() для печати информации о счете. Протестировать

функционирование переопределенных методов и операторов на простом примере.

2. Создать класс рациональных чисел. В классе два поля – числитель и знаменатель. Предусмотреть конструктор. Определить операторы ==, != (метод Equals()), <, >, <=, >=, +, – , ++, --. Переопределить метод ToString() для вывода дроби.

Если останется время, то определить операторы преобразования типов между типом дробь, float, int. Определить операторы *, /, %.

3.13 Свойства и индексаторы

Свойства

Свойства – это наборы функций, которые могут быть доступны клиенту таким же способом, как открытые поля класса. Идея свойства заключается в методе или паре методов, которые ведут себя с точки зрения клиентского кода как поле. Чтобы определить свойство в C#, используется следующий синтаксис:

```
class Person
{
   private string name;

   public string Name
   {
      get
      {
         return name;
      }

      set
      {
         name = value;
      }
   }
}
```

Здесь у нас есть закрытое поле name и есть общедоступное свойство Name. Хотя они имеют практически одинаковое название за исключением регистра, но это не более чем стиль, названия у них могут быть произвольные и не обязательно должны совпадать.

Через это свойство мы можем управлять доступом к переменной name. Стандартное определение свойства содержит блоки get и set.

Средство доступа **get** не принимает никаких параметров и должно возвращать значение того типа, который объявлен для свойства. Для средства **set** в явном виде не передается никаких параметров, но компилятор

предполагает, что оно принимает один параметр со значением value, относящийся к типу, указанному для свойства.

Внутри метода get и set можно написать дополнительный код, осуществляющий какую-либо обработку, допустим, внутри set поместить код для проверки корректности введенных данных.

Свойство описывается как член класса, предоставляет доступ к полям класса. В соответствии с идеологией ООП имеет смысл поля делать private, а свойства более доступными: public, protected, internal (в зависимости от целей). Также рекомендуется имена полей начинать с маленькой буквы, а имя свойства, предоставляющего доступ к полю, задавать таким же, как и у поля, но начинать с заглавной буквы.

Существует возможность создать свойство, доступное только для чтения. Для этого надо исключить set из определения свойства. Чтобы определить свойство, доступное только для записи, необходимо из его определения исключить get. Кроме того, для set и get можно применять различные модификаторы доступа, но надо помнить, что модификатор доступа одного из средств get или set должен совпадать с модификатором доступа всего свойства.

Индексаторы

Индексация позволяет индексировать объекты таким же способом, как массив или коллекцию, дает возможность обращаться к членам класса как к массиву.

Пусть объект состоит из нескольких подэлементов (например, listbox состоит из нескольких строк). Индексация позволяет обращаться к подэлементам, как в массивах.

```
class StringList
{
        private string[] list;
        public string this[int index]
        {
                get{ return list[index];} set{ list[index] = value;}
        }
}
```

Индексатор – это свойство с именем this, квадратными скобками и индексом в них. Для использования индексатора обращаемся к объекту:

```
StringList myList = new StringList();
myList[3] = “ok”;              // индексатор записывает
string myString = myList[3];      // индексатор читает
```

Индексаторы могут быть статичными элементами класса и их можно перегружать для различных типов индексов. Индексаторы не хранят значения и поэтому не могут быть переданы как ref- и out-параметры.

При объявлении индексаторов необходимо определить, по крайней мере,

один параметр, а также значения для всех параметров. Можно определить несколько параметров для одного индексатора:

```
class MultipleParameters
{
        public string this[int one, int two]
        {
                get{…}
                set{…}
         }
}
…
MultipleParameters mp = new MultipleParameters(); string s = mp[2,3];
…
```

Вопросы к п. 3.13

1. Каким образом можно объявить свойство только для чтения?
2. Может ли класс содержать индексатор, зависящий от двух параметров разного типа?
3. Как в классе объявить индексатор только для чтения?

Задания для самостоятельной работы

Задания на свойства, создание и использование индексаторов

1. Из класса банковский счет удалить методы, возвращающие значения полей номер счета и тип счета, заменить эти методы на свойства только для чтения. Добавить свойство для чтения и записи типа string для отображения поля держатель банковского счета.

Изменить класс BankTransaction, созданный для хранений финансовых операций со счетом, - заменить методы доступа к данным на свойства для чтения.

2. Добавить индексатор в класс банковский счет для получения доступа к любому объекту BankTransaction.

3.14 Атрибуты в NET

Атрибуты в .NET представляют специальные инструменты, которые позволяют встраивать в сборку дополнительные метаданные. Атрибуты могут применяться как ко всему типу (классу, интерфейсу и т.д.), так и к отдельным его частям (методу, свойству и т.д.). Основу атрибутов составляет класс System.Attribute, от которого образованы все остальные классы атрибутов.

В .NET имеется множество различных классов атрибутов. Например, при сериализации в различные форматы используются атрибуты [Serializable] и [NonSerialized]. С помощью рефлексии стандартные классы .NET получают использованные атрибуты и производят определенные

действия. Например, атрибут [Serializable] указывает классу BinaryFormatter, что объекты с данным атрибутом можно сохранять в бинарный файл. В то же время, пока к классу с атрибутом не применена рефлексия, атрибут не размещается в памяти и никакого влияния на данный класс не оказывает.

Допустим, нам надо проверять пользователя на соответствие некоторым возрастным ограничениям. Создадим свой атрибут, который будет хранить пороговое значение возраста, с которого разрешены некоторые действия:

```
public class AgeValidationAttribute : System.Attribute
{
   public int Age { get; set; }

   public AgeValidationAttribute()
   { }

   public AgeValidationAttribute(int age)
   {
      Age = age;
   }
}
```

По сути это обычный класс, унаследованный от System.Atribute. Теперь применим его к некоторому классу:

```
[AgeValidation(18)]
public class User
{
   public string Name { get; set; }
   public int Age { get; set; }
   public User(string n, int a)
   {
      Name = n;
      Age = a;
   }
}
```

Пусть некоторый класс User применяет атрибут. Для этого имя атрибута указывается в квадратных скобках. Причем суффикс Attribute указывать необязательно. Обе записи [AgeValidation(18)] и [AgeValidationAttribute(18)] будут авноправны.

Если конструктор атрибута предусматривает использование параметров (public AgeValidationAttribute(int age)), то после имени атрибута мы можем указать значения для параметров конструктора. В данном случае передается значение для параметра age. То есть фактически мы говорим, что в AgeValidationAttribute свойство Age будет иметь значение 18.

В качестве альтернативы можно использовать именованные параметры для всех свойств атрибута, если класс атрибута имеет конструктор без параметров: [AgeValidation(Age = 18)]

Теперь получим с помощью рефлексии атрибут класса User и используем его для проверки объектов данного класса:

```
class Program
{
   static void Main(string[] args)
   {
      User tom = new User("Tom", 35);
      User bob = new User("Bob", 16);
      bool tomIsValid = ValidateUser(tom);    // true
      bool bobIsValid = ValidateUser(bob);    // false

      Console.WriteLine($"Результат валидации Тома: {tomIsValid}");
      Console.WriteLine($"Результат валидации Боба: {bobIsValid}");
      Console.ReadLine();
   }
   static bool ValidateUser(User user)
   {
      Type t = typeof(User);
      object[] attrs = t.GetCustomAttributes(false);
      foreach (AgeValidationAttribute attr in attrs)
      {
         if (user.Age >= attr.Age) return true;
         else return false;
      }
      return true;
   }
}
```

В данном случае в методе ValidateUser через параметр получаем некоторый объект User и с помощью метода GetCustomAttributes вытаскиваем из типа User все атрибуты. Далее берем из атрибутов атрибут AgeValidationAttribute при его наличии (ведь мы можем его и не применять к классу) и проверям допустимость возраста пользователя. Если пользователь прошел проверку по возрасту, то возвращаем true, иначе возвращаем false. Если атрибут не применяется, возвращаем true.

Ограничение применения атрибута

С помощью атрибута AttributeUsage можно ограничить типы, к которым будет применяться атрибут. Например, мы хотим, чтобы определенный выше атрибут мог применяться только к классам:

```
[AttributeUsage(AttributeTargets.Class)]
public class RoleInfoAttribute : System.Attribute
{
//.....................................
}
```

Ограничение задает перечисление AttributeTargets, которое может принимать еще ряд значений:

All: используется всеми типами.

Assembly: атрибут применяется к сборке.

Constructor: атрибут применяется к конструктору.

Delegate: атрибут применяется к делегату.

Enum: применяется к перечислению.

Event: атрибут применяется к событию.

Field: применяется к полю типа.

Interface: атрибут применяется к интерфейсу.

Method: применяется к методу.

Property: применяется к свойству.

Struct: применяется к структуре.

С помощью логической операции ИЛИ можно комбинировать эти значения. Например, пусть атрибут может применяться к классам и структурам: [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]

Вопросы к п. 3.14:

1. Можно ли отменить один объект класса используя атрибут?
2. Опишите какой механизм используется для определения значения атрибутов при выполнении приложения?
3. Где хранятся значения атрибутов?

Задания для самостоятельной работы:

Выполните задание на использование условного атрибута (ConditionalAttribute), создание пользовательского атрибута/

1. Использование предопределенного условного атрибута для условного выполнения кода (указывает компиляторам, что при отсутствии символа условной компиляции, вызов метода или атрибут следует игнорировать).

В классе банковский счет добавить метод DumpToScreen, который отображает детали банковского счета. Для выполнения этого метода использовать условный атрибут, зависящий от символа условной компиляции DEBUG_ACCOUNT. Протестировать метод DumpToScreen.

2. Создать пользовательский атрибут DeveloperInfoAttribute. Этот атрибут позволяет хранить в метаданных класса имя разработчика и, дополнительно, дату разработки класса. Использовать этот атрибут для записи имени разработчика класса рациональные числа.

СПИСОК ЛИТЕРАТУРЫ

1. Абрамов, С.А. Элементы программирования / С.А. Абрамов. – М.: Наука, 1982. – 96с.
2. Бен-Ари, М. Языки программирования. Практический сравнительный анализ / М. Бен-Ари; пер. с англ. – М.: Мир, 2000. – 366с.
3. Вирт, Н. Алгоритмы + структуры данных = программы / Н. Вирт; пер. с англ.– М.: Мир,1985.– 406с.
4. Евдокимов, П.В. C# на примерах / П.В. Евдокимов. – СПб.: Наука и техника, 2016. – 304 с.
5. Жоголев, Е.А. Технология программирования / Е.А. Жоголев. – М.: Научный мир, 2004. – 216 с.
6. Зелковец, М. Принципы разработки программного обеспечения / М. Зелковец, А. Шоу, Дж. Гэннон; пер. с англ. – М.: Мир, 1982. – 368с.
7. Липаев, В.В. Качество программного обеспечения / В.В. Липаев. – М.: Финансы и статистика, 1983. – 263с.
8. Липаев, В.В. Мобильность программ и данных в открытых информационных системах / В.В. Липаев, Е.Н. Филиппов. – М.: Научная книга, 1997. – 360с.
9. Майерс, Г. Надежность программного обеспечения / Г. Майерс; пер. с англ. – М.: Мир, 1980. – 360с.
10. Пахомов, Б.И. C# для начинающих / Б.И. Пахомов. – СПб.: БХВ-Петербург, 2014. – 432 с.
11. Программирование: В 2 т. Т. 1: учебник для студ. Учреждений высш. проф. образования / Э.А. Нигматулина, Н.И. Пак, М.А. Сокольская, Т.А. Степанова; под ред. Н.И. Пака. – М.: Издательский центр «Академия», 2013. – 272 с. – (Сер. Бакалавриат).
12. Скотт, Д. Теория решеток, типы данных и семантика / Данные в языках программирования / Д. Скотт; пер. с англ. – М.: Мир, 1982. – С. 25-53.
13. Турский, В. Методология программирования / В. Турский; пер с англ. – М.: Мир, 1981. – 264с.
14. Фуксман, А.Л. Технологические аспекты создания программных систем / А.Л. Фуксман. – М.: Статистика, 1979. – 183с.

СОДЕРЖАНИЕ

Учебное издание

Елена Викторовна Киргизова

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ: ОТ ТЕОРИИ К ПРАКТИКЕ

Редактор И.А. Вейсиг
Компьютерная верстка автора

Подписано в печать 15.02.2021 г. Формат 60*84/16
Усл. печ. л. 7,8. Бумага офсетная
Тираж 100 экз. Заказ

Библиотечно-издательский комплекс
Сибирского федерального университета
660041, Красноярск, пр. Свободный, 82 а
Тел. (391) 206-26-67; http://bik.sfu-kras.ru
E-mail publishing_house@sfu-kras.ru