№ 2160

С.Э. Мурадханов
А.И. Широков

Информатика и программирование

Основы разработки программ на языке C#

№ 2160

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ «МИСиС»

Кафедра автоматизированных систем управления

С.Э. Мурадханов
А.И. Широков

Информатика и программирование

Основы разработки программ на языке C#

Учебник

Допущено Учебно-методическим объединением
по образованию в области прикладной информатики
в качестве учебика для студентов, обучающихся
по направлению «Прикладная информатика»
и другим экономическим специальностяма

МИСиС
ИЗДАТЕЛЬСКИЙ ДОМ

Москва 2013

УДК 681.30
М91

Рецензент
канд. техн. наук, доц. *И.Н. Лесовская*
(Национальный исследовательский университет
«Высшая школа экономики»)

Мурадханов, С.Э.

М91 Информатика и программирование : основы разработки программ на языке C# : учеб. / С.Э. Мурадханов, А.И. Широков. – М. : Изд. Дом МИСиС, 2013. – 304 с.

ISBN 978-5-87623-735-4

Алгоритмические языки высокого уровня – это основа и фундамент для разработки сложных информационных комплексов и систем. Дисциплина, изучающая эти средства, имеет как теоретическую составляющую (процедурный и объектно-ориентированный подход к анализу предметной области), так и практико-ориентированную (синтез информационной системы с помощью инструментов процедурной или объектно-ориентированной разработки). Такая дисциплина является важной составляющей профессиональной подготовки бакалавров в области информационных технологий. Она может быть основана на разных языковых средствах, а представленный учебник дает как теоретические знания, так и практические навыки по разработке приложений в среде Visual Studio (2005–2012) на языке программирования C#.

Учебник предназначен для студентов специальностей 230100 «Информатика и вычислительная техника», 231300 «Прикладная математика», 230700 «Прикладная Информатика», 220700 «Автоматизация технологических процессов и производств».

УДК 681.30

ISBN 978-5-87623-735-4

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Современный уровень развития общества предполагает широкое использование информационных технологий в различных сферах деятельности. Знания в области информационных технологий и их умелое использование обеспечивают конкурентоспособность специалиста. Специалисты в области информационных технологий должны владеть современными технологиями разработки программ.

В настоящем издании представлены основные понятия языка программирования C# и способы их использования при разработке в среде .Net Framework. Приобретаемые при изучении изложенного материала навыки являются необходимым начальным этапом для углубленного изучения объектно-ориентированного подхода и возможностей платформы Microsoft .Net Framework.

В целом материал учебника ориентирован прежде всего на студентов, обучающихся по специальностям в области информационных технологий, но будет полезен также и для желающих самостоятельно освоить язык программирования C#. Учебник посвящен основам этого языка, без знания которых невозможно его практическое применение.

Язык программирования C# нельзя изучать «линейно» – «от аксиом к теоремам, описаниям и задачам», поэтому предлагаемый материал (программирование на языке C#) излагается «по спирали». Некоторые понятия, использованные в той или иной программе-примере, имеют краткое описание, но в последующих разделах они будут полностью рассмотрены и объяснены.

1. ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

В теории программирования важным является понятие «парадигма». Трудно дать ему однозначное определение, потому что теория программирования непрерывно развивается. Это понятие разные авторы связывают со стилем написания программ, взглядом на то, как организованы вычисления или данные, как реализуются языковые конструкции (термин «парадигма программирования» ввел Роберт Флойд в 1979).

Основные парадигмы программирования появились, развивались и сложились по мере возрастания сложности решаемых задач, методов и методологий, необходимых для их практического решения. Существуют различные парадигмы программирования: процедурная, модульная, функциональная, структурная, логическая, объектно-ориентированная и др. (всего их более трех десятков). Многие языки могут поддерживать несколько парадигм программирования (т.е. являются мультипарадигменными).

Примечание. *Парадигма (греч. paradeigma) означает пример, образец. Американский философ Кун Томас Самюэл дал следующее определение этого понятия:* «*Парадигма – совокупность знаний, методов и ценностей, безоговорочно разделяемых членами научного сообщества*». *Парадигма – определенный взгляд на явления окружающего мира и представление о возможных действиях с ними. Парадигма в науке – концептуальная модель постановки проблем, методов их решений и способов исследований проблем.*

Парадигма программирования *– принцип (подход, модель) построения программной системы, её структурирования и связи её частей.*

Классификация парадигм языков программирования:

• *директивные* (directive), называемые также процедурными (procedural), или императивными (imperative) – Algol, Fortran, Basic, Pascal, C;

• *декларативные* (declarative) – делятся на две группы:

– функциональные (functional) – Lisp, Scheme, Erlang, CAML, Haskell, Clean;

– логические (logic) – Prolog (Пролог), Visual Prolog, Mercury, Godel;

• *объектно-ориентированные* (object-oriented) – C++, C#, Java, Python, Ruby.

Разница между *декларативной* и *директивной* парадигмами состоит в том, что в первой программа заявляет (декларирует), что должно быть достигнуто в качестве цели, а во второй – предписывает, как ее достичь. Обычно это поясняется примером с курьером, которому надо пройти из пункта А в пункт Б. В *декларативной программе* описывается план города, в котором указано, где расположены оба пункта и правила уличного движения. Руководствуясь этими правилами и планом города, курьер должен определить путь от исходного пункта к конечному. В *директивной программе* приводится список команд, задающих точное указание перемещений от начального пункта к конечному.

1.1. Процедурное программирование

Процедурное программирование (появилось в 50-е годы XX в.) – это парадигма программирования, основанная на концепции вызова процедур (подпрограмм). Процедурное программирование – отражение архитектуры ЭВМ (предложено Джоном фон Нейманом в 40-х годах XX в. и применяется до настоящего времени).

Примечание. *Архитектура Джона фон Неймана предполагает, что компьютер должен содержать следующие устройства: а) арифметическо-логическое устройство для выполнения операций; б) устройство управления для организации процесса выполнения программ; в) запоминающее устройство для хранения программ и данных. Кроме структуры компьютера фон Нейман (в 1945 г.) объявил принципы, положенные в основу архитектуры большинства электронных вычислительных машин:*

1) ***принцип программного управления*** *(программа состоит из набора команд, которые выполняются процессором автоматически друг за другом в заданной последовательности);*

2) ***принцип однородности памяти*** *(программы и данные хранятся в одной и той же памяти; над командами можно выполнять такие же действия, как и над данными);*

3) ***принцип адресности*** *(основная память структурно состоит из пронумерованных ячеек – отсюда следует возможность давать имена областям памяти, так чтобы к запомненным в них значениям можно было впоследствии обращаться или менять их в процессе выполнения программ с использованием присвоенных имен).*

Компьютеры, построенные на этих принципах и имеющие объявленную структуру, относят к типу фон-неймановских*.*

Парадигма процедурного программирования гласит: «*Надо определить необходимые процедуры и использовать для их реализации доступные алгоритмы. Реализацию алгоритмов вычислений необходимо создавать с помощью мелких, не зависимых друг от друга процедур, которые вызывают друг друга в соответствии с логикой программы*». Обработка данных осуществляется с помощью алгоритмов, производящих нужные вычисления. Процедуры, подпрограммы, методы или функции (не математические функции, а функции, подобные тем, которые используются в функциональном программировании) содержат последовательность шагов выполнения программы.

В ходе выполнения программы любая процедура может быть вызвана из любой точки, включая саму данную процедуру. Работа программы сводится к последовательному выполнению операторов с целью преобразования значений исходных данных в результаты. Основное действие определяется оператором присваивания (он изменяет содержимое областей памяти). С точки зрения программиста, есть программа и память, причем программа последовательно обновляет память. Программа не только определяет, что нужно сделать, но как это сделать и в какой последовательности. Концепция памяти как хранилища значений, содержимое которого может обновляться операторами программы, является фундаментальным принципом в процедурном (императивном – лат. *imperativus* (*повелительный*)) программировании. Для поддержки процедурной парадигмы языки программирования должны содержать механизм передачи параметров функций и получения их значенией. Теоретической моделью процедурного программирования служит алгоритмическая система – «машина Тьюринга».

Примечание. *Алан Тьюринг для обоснования и уточнения понятия алгоритма предложил абстрактный универсальный исполнитель (1936 г.) – абстрактную логическую вычислительную конструкцию, названную впоследствии «машиной Тьюринга» (МТ).* ***МТ*** *– устройство, состоящее из бесконечной ленты, разделенной на ячейки и головки для чтения/записи. Ячейка МТ может содержать символы 3 типов: 0, 1, пусто. В МТ нет ограничений на набор символов: можно применять любые символы. В каждый отдельный момент времени головка МТ находится над одной из ячеек и считывает ее содержимое. В зависимости от содержимого ячейки МТ может выполнить одну из операций: стереть текущее содержимое ячейки (E),*

напечатать что-либо в ячейке (P), сдвинуть головку вдоль ленты на одну ячейку влево (L) или вправо (R), завершить работу (H – операция остановки). МТ всегда находится в одном из помеченных состояний (пометка может быть чем угодно – буквой, словом, числом – главное, чтобы МТ могла отличить одно состояние от других). Работа МТ состоит в считывании символа из ячейки, над которой находится головка, выполнении некоторых операций и переходе к следующему состоянию. Операций может быть несколько (стереть символ, напечатать 1, сдвинуть головку влево: запись – E, P1, L).

Пример. МТ – сложение чисел. *На ленте имеется два числа, представленные в единичной форме (число 3 представляется как последовательность 111), разделенные одной пустой ячейкой. Записать на ленте последовательность – сумму двух чисел. Информация на ленте МТ представлена в виде -111-11111-. В конце на ленте должна остаться последовательность -11111111-. Вначале головка МТ считывает самую левую единицу*:

Состояние	**Текущий символ**	**Операция**	**Следующее состояние**
a	**1**	**R**	**a**
a	**–**	**P1, R**	**b**
b	**1**	**R**	**b**
b	**–**	**L**	**c**
c	**1**	**E, H**	

Начальных состояний может быть сколько угодно. МТ может находиться в состоянии b и считывать символ 1, или находиться в том же состоянии, но считывать пустой символ. Структура программы МТ*: при заданном состоянии и текущем символе требуется выбрать операцию. Программа сканирует последовательность из единиц слева направо до тех пор, пока не будет обнаружена ячейка, содержащая пустой символ. В эту ячейку записывается единица. Чтобы* «*скорректировать*» *сумму для получения правильного значения, необходимо найти самую правую единицу, стереть ее и остановиться.*

Ход работы МТ выглядит так: «-111-11111-» «-111**1**1111**1**-» «-11111111-».

Одним из первых процедурных языков программирования высокого уровня стал Фортран (Fortran – Formula Translator), созданный в 1954–1957 гг. XX в. в США под руководством Джона Бэкуса в кор-

порации IBM (языки программирования Алгол60, Кобол (Cobol), Паскаль (Pascal) и Си (C) продолжили это направление). Совершенствование языков процедурного программирования и технологии привело к появлению модульного программирования (программа проектируется из независимых частей – модулей, в которых используется единый принцип передачи данных между модулями – интерфейс).

1.2. Модульное программирование

Модульное программирование (появилось в начале 60-х годов XX в.) – это парадигма программирования, основанная на концепции разбиения программной системы на раздельно компилируемые компоненты (модули), каждый из которых представляет собой набор связанных процедур вместе с данными, которые обрабатываются только этими процедурами. Концепцию впервые реализовал Н. Вирт в языке Modula, а затем ее «подхватили» и остальные, распространенные в то время языки программирования. Парадигма модульного программирования гласит: «*Надо определить необходимые модули и реализовать их так, чтобы данные и процедуры были скрыты в этих модулях*». Физическим представлением каждого модуля является файл. Модули после раздельной компиляции используются в программной системе в виде независимых частей (это позволяет использовать модули как инструмент разработки библиотек прикладных программ). Для того чтобы обеспечить максимальную независимость модулей друг от друга, надо четко отделить процедуры и данные, которые будут вызываться другими модулями, – *открытые* (public) процедуры и данные – от вспомогательных процедур и данных, которые используются только в этом (данном) модуле, – *закрытых* (private) процедур и данных. Первые перечисляются в отдельной части модуля – *интерфейсе* (interface), вторые участвуют только в *реализации* (implementation) модуля. В разных языках программирования это деление производится по-разному. В языке Pascal модуль специально делится на интерфейс и реализацию; в языке C интерфейс выносится в отдельные «головные» (header) файлы; в языке C++ для описания интерфейса используются абстрактные классы; в языке Java для описания интерфейса используются абстрактные классы и конструкция interface.

В процессе разработки программы ее модульная структура может по-разному формироваться и использоваться для определения порядка программирования и отладки модулей, указанных в этой структу-

ре. Поэтому можно говорить о разных методах разработки структуры программы. Обычно в литературе обсуждаются два метода: метод восходящей разработки и метод нисходящей разработки.

Метод восходящей разработки заключается в следующем. Сначала строится модульная структура программы в виде дерева. Затем поочередно программируются модули программы, начиная с модулей самого нижнего уровня (листья дерева модульной структуры программы), в таком порядке, чтобы для каждого программируемого модуля были уже запрограммированы все модули, к которым он может обращаться. После того как все модули программы запрограммированы, производится их поочередное тестирование и отладка в таком же (восходящем) порядке, в каком производилось их программирование.

Метод нисходящей разработки предполагает, как и предыдущий метод, сначала построение модульной структуры программы в виде дерева. Затем поочередно программируются модули программы, начиная с модуля самого верхнего уровня (головного), с переходом к программированию какого-либо другого модуля только в том случае, если уже запрограммирован модуль, который к нему обращается. После того как все модули программы запрограммированы, производится их поочередное тестирование и отладка в таком же (нисходящем) порядке.

Парадигма модульного программирования также известна как «принцип сокрытия данных», который легко расширяется до понятия «сокрытие информации», что позволяет скрыть от пользователя не только детали реализации каждого модуля, но и детали реализации процедур. На основе модульности строятся библиотеки процедур и функций, которые обеспечивают полный набор действий с библиотекой с использованием общих структур данных, процедур и функций.

1.3. Структурное программирование

Идеи структурного программирования появились в начале 70-х годов XX в. в компании IBM, в их разработке участвовали известные ученые Э. Дейкстра, Х. Милс, Э. Кнут, С. Хоор. Развитие технологии *структурного программирования* связано с публикацией письма Э. Дейкстры (1968 г.) в Ассоциацию вычислительной техники (ACM – *Association for Computing Machinery*) «О вреде использования операторов GOTO». В те времена программы создавались с активным использованием операторов безусловного перехода. Непра-

вильное и необдуманное использование произвольных переходов в тексте программы приводит к получению запутанных, плохо структурированных программ («***спагетти-код», или «лапша»*** *– плохо спроектированная, запутанная и трудная для понимания программа, содержащая много операторов GOTO, исключений и других конструкций,* «*ухудшающих*» *структурированность*), по тексту которых практически невозможно понять порядок исполнения и взаимозависимость фрагментов. Обращая внимание на недостатки таких программ, Дийкстра предложил концепцию *структурного программирования* (развитую впоследствии Николаусом Виртом), позволяющую избежать использования таких операторов.

Структурное программирование **–** это парадигма программирования, основанная на концепции представления структуры любой программной системы в виде 4 базовых управляющих конструкций:

• последовательное исполнение – однократное выполнение операций в том порядке, в котором они записаны в тексте программы [*последовательность действий*]*;*

• ветвление – однократное выполнение одной из двух или более операций, в зависимости от выполнения некоторого заданного условия [*альтернатива-условие*];

• цикл – многократное исполнение одной операции до тех пор, пока выполняется некоторое заданное условие (условие продолжения цикла) [*повторение-цикл*];

• вызов процедур (подпрограмм).

Управляющие конструкции допускают вложенность. В управляющие операторы можно включать вызовы процедур (оформленных как модули), так что структурное программирование использует *модульный принцип.* Модуль может содержать обращения к другим модулям. Это обеспечивает принцип *нисходящего проектирования* программ (принцип *сверху вниз).* По этому принципу алгоритм сначала разбивается на этапы, затем каждый этап делится на более мелкие этапы второго уровня и т.д. После этого каждый этап проектируется как отдельный модуль. Нисходящее проектирование считается наиболее продуктивным, так как реализует общий принцип *системного подхода*, при котором легко распределять разработку программы по отдельным исполнителям и осуществлять общее руководство.

Примечание. *Методы управления* – это система способов воздействия субъекта управления на объект для достижения определенно-

го результата. Основу системы методов, используемых в управлении, составляет *общенаучная методология*, предусматривающая системный, комплексный подход к решению проблем, а также применение таких методов, как моделирование, экспериментирование, конкретно-исторический подход, экономико-математические и социологические измерения и т.д. *Системный подход* применяется в менеджменте как способ упорядочения управленческих проблем, посредством которого осуществляется их структурирование, определяются цели решения, выбираются варианты, устанавливаются взаимосвязи и зависимости элементов проблем, а также факторы и условия, оказывающие воздействие на их решение. *Системный подход* – это такое направление в методе логики научного познания и практической деятельности, в основе которого лежит исследование любого объекта как сложной целостной кибернетической социально-экономической системы. В наиболее общем виде под системой понимается совокупность взаимосвязанных элементов, образующих определенную целостность, некоторое единство.

Основные принципы системного подхода (системного анализа):

1. ***Целостность***, позволяющая рассматривать одновременно систему как единое целое и в то же время как подсистему для вышестоящих уровней.

2. ***Иерархичность строения***, т.е. наличие множества (по крайней мере, двух) элементов, расположенных на основе подчинения элементов низшего уровня элементам высшего уровня. Реализация этого принципа хорошо видна на примере любой конкретной организации. Как известно, любая организация представляет собой взаимодействие двух подсистем: управляющей и управляемой. При этом одна подчиняется другой.

3. ***Структуризация***, позволяющая анализировать элементы системы и их взаимосвязи в рамках конкретной организационной структуры. Как правило, процесс функционирования системы обусловлен не столько свойствами ее отдельных элементов, сколько свойствами самой структуры.

4. ***Множественность***, позволяющая использовать множество кибернетических, экономических и математических моделей для описания отдельных элементов и системы в целом.

Применение системного подхода позволяет наилучшим образом организовать процесс принятия решений на всех уровнях в системе управления.

Системный подход – это методология исследования разного рода комплексов, позволяющая глубже и лучше осмыслить их сущность

(структуру, организацию и другие особенности) и найти оптимальные пути и методы воздействия на развитие таких комплексов и систему управления ими. **Системный подход** – это всеобъемлющий комплексный подход, предполагающий всесторонний учёт специфических характеристик соответствующего объекта, определяющих его структуру, а следовательно, и организацию. Системный подход к управлению предполагает анализ управления как процедуры или процесса принятия управленческих решений.

В основе структурного программирования лежит *декомпозиция* (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших подпрограмм. Структурный подход требует представления задачи в виде иерархии подзадач простейшей структуры, для получения которой применяется метод пошаговой детализации (*процедурной декомпозиции*). Метод пошаговой детализации заключается в определении общей структуры системы в виде одной из 4 базовых управляющих конструкций (каждая подзадача, в свою очередь, разбивается на подзадачи с использованием тех же конструкций). Процесс продолжается до тех пор, пока на очередном уровне не получается подзадача, которая достаточно просто реализуется средствами используемого языка. Других средств управления последовательностью выполнения операций не предусматривается.

В результате использования структурного программирования получается программная система, в которой принципиально отсутствует оператор перехода GOTO, поэтому такая технология называется «программирование без GOTO». Типичными структурными языками программирования являются Си (C) и Паскаль (Pascal). И хотя в этих языках такие операторы присутствуют (они оставлены для совместимости с ранними версиями языков) – при создании сложных программных систем они не используются.

1.4. Функциональное программирование

Теория, положенная в основу функционального подхода, родилась в 20–30-х годах XX в. В числе разработчиков математических основ функционального программирования можно назвать Мозеса Шёнфинкеля (Германия и Россия) и Хаскелла Карри (Англия), разработавших комбинаторную логику, а также Алонзо Чёрча (США), создателя λ-исчисления. Теория так и оставалась теорией, пока в начале 50-х годов прошлого века Джон МакКарти не разработал язык LISP.

λ-исчисление стало теоретической базой для описания и вычисления функций. Являясь математической абстракцией, а не языком программирования, оно составило базис почти всех языков функционального программирования на сегодняшний день. Сходное теоретическое понятие – комбинаторная логика – является более абстрактным, нежели λ-исчисление, и было создано раньше. Эта логика используется в некоторых «эзотерических» языках, например в Unlambda. И λ-исчисление, и комбинаторная логика были разработаны для более ясного и точного описания принципов и основ математики.

Функциональное программирование (появилось в конце 50-х годов XX в.) – это парадигма программирования, основанная на концепции трактовки процесса ***вычисления*** как ***вычисления*** значений функций в математическом понимании (единственный результат работы – возвращаемое значение, без побочных эффектов, без изменения данных, как при использовании процедурного программирования). *Выполнение программы представляет собой* ***вычисление*** *некоторого выражения, описывающего применение функции к входным данным. Функциональная программа состоит из совокупности определений функций, которые в свою очередь представляют собой вызовы других функций и предложений, управляющих последовательностью вызовов. Каждая функция возвращает некоторое значение в вызвавшую его функцию, вычисление которой после этого продолжается; этот процесс повторяется до тех пор, пока не будет получен результат.* Функциональное программирование противопоставляется парадигме процедурного (императивного) программирования, в которой программе предписывается последовательность выполняемых действий, в то время как в функциональном программировании способ решения задачи описывается при помощи зависимости функций друг от друга (в том числе возможны рекурсивные зависимости), но без указания последовательности шагов. Первым *функциональным языком программирования* стал **Лисп** (LISP – LISt Processing), созданный в 1958 г. Джоном Маккарти в период его работы в Массачусетсском технологическом институте и реализованный первоначально для IBM 700/7000. В Лисп программа и обрабатываемые ею данные представляются в одной форме – в виде списка. Используемый в Лисп функциональный подход к программированию основывается на том, что вся обработка информации для получения результата может быть представлена в виде вложенных (рекурсивных) вызовов функций, выполняющих некоторые действия (значение одной функции используется как аргумент другой; значение этой функции становится

аргументом следующей и т.д., пока не будет получен результат – решение задачи). Программы строятся из логически ***распределенных, или разделенных,*** определений функций. Последние состоят из управляющих структур, организующих вычисления, и из вложенных вызовов функций. Основными методами функционального программирования являются композиция и рекурсия, которые представляют собой реализацию идей теории рекурсивных функций («вызывающих самих себя»). *Синтаксис такого языка выглядит следующим образом*:

Функция_n (... функция_2 (функция_1 (данные))...)

Пример вычисления корней квадратного уравнения $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ ***на условном языке функционального программирования:***

Если определены базовые функции: **СУМ**(x,y) – сумма (x + y); **ВЫЧ**(x,y) – разница (x - y); **УМН**(x,y) – произведение (x * y); **ДЕЛ**(x,y) – деление (x / y); **ККВ**(x) – извлечения квадратного корня из x, то корни квадратного уравнения можно вычислить так:

КореньX1= ДЕЛ(СУМ(ВЫЧ(0,B),ККВ(ВЫЧ(УМН(B,B),УМН(4,УМН(A,C))))),УМН(2,A))

КореньX2= ДЕЛ(ВЫЧ(ВЫЧ(0,B),ККВ(ВЫЧ(УМН(B,B),УМН(4,УМН(A,C))))),УМН(2,A))

Примечание. ***Краткие сведения о языке Лисп***

1. *В основе Лисп лежат символьные выражения, называемые S-выражениями, которые и образуют область определения для функциональных программ.*

S-выражение (Simbolic expresion) – основная структура данных в Лисп:

(ДЖОН СМИТ 33 ГОДА) *и* ***((МАША 21) (ВАСЯ 24) (ПЕТЯ 1))*** *– S-ВЫРАЖЕНИЯ*

S-выражение – это либо атом, либо список. Атомы – это простейшие объекты Лисп, из которых строятся остальные структуры. Атомы бывают двух типов – символьные и числовые. Символьные атомы – последовательность букв и цифр, при этом должен быть по крайней мере один символ, отличающий его от числа: ***ДЖОН AB13 B54 10A***

2. *В Лисп* ***список*** *– это последовательность элементов. Элементами являются или атомы, или списки.Списки заключаются в скобки, их элементы разделяются пробелами:*

(банан) *– 1 атом* ***(б а н а н)*** *– 5 атомов* ***(a b (c d) e)*** *– 4 элемента*

3. Функции «Математическая запись», «Запись на Лисп»:

f(x) g(x, y) h(x, g(y, z)) (f x) (g x y) (h x (g y z)) ***(+ 1) = 1 (+ 1 2 3 4 5) = 15***

*x + y*z x + y < z (+ x (* y z)) (< (+ x y) z)* ***(* 1 2 3 4 5) = 120***

В некоторых случаях не требуется вычисление значений выражений, а требуются само выражение. Если прямо ввести **** (+ 2 3)****, то* ***5*** *получится как значение. S-выражения, которые не надо вычислять, помечают для интерпретатора апострофом «'».*

Для задания функций в Лисп используется форма: ***(defun <имя-функции> <параметр ><тело-функции>)*** *Вызов функции:* ***(<имя-функции><значения аргументов>).***

Пример*: * (defun double (num) (* 2 num)) * (double 7) Результат: 14.*

4. *Предикат в Лисп – функция,которая определяет, обладает ли аргумент определенным свойством, и возвращает в качестве значения* ***T*** *или* ***NIL*** *(значение не-NIL – истина).*

ATOM *проверяет, является ли аргумент атомом – значение будет T, если является атомом, и nil – в обратном случае:* ****(atom ' (a b)) = = nil****. Предикат* ***EQ*** *сравнивает два символа и возвращает T, если они одинаковые, и nil – в обратном случае:* **** (eq ' cat ' dog) = nil.***

5. *Предложение* ***cond*** *является основным средством ветвления вычислений. Структура условного предложения:*

(COND (p_1 ***a*** $_1$***)…(*** p_n ***a*** $_n$***))****, где* p_i *– <проверка-i >,* a_i *– <действие-i>*

Значение предложения ***COND*** *определяется так: 1) выражения <проверка-i >, которые выполняют роль предикатов, вычисляются последовательно, слева направо, до тех пор, пока не встретится выражение, значением которого не является* ***NIL****; 2) вычисляется выражение (результирующее), соответствующее этому предикату, и полученное значение возвращается в качестве значения всего предложения* ***COND****.*

Лисп – пример программы: вычисление факториала неотрицательного числа:

DEFINE (Факториал N n)	«Факториал» – имя функции, N – параметр (переменная)
(COND ((=N 1) 1)	Описание функции «факториал»
((>N 1) (*N (fac (- N 1))))))	Условие: если N <= 1, то возвращается единица
Вызов: **(fac 20)** Результат: **24**	если N <= 1, то возвращается N* (Факториал (N-1)

*При N = 4 факториал(4) (4!) будет равен (по приведенной программе): Факториал(4) = 4*Факториал(3) = 4*(3*Факториал(2)) = 4*3*(2*Факториал(1)) = 4*3*2*1=24*

В отличие от традиционного подхода к программированию (императивное программирование), при котором выполнение программы рассматривается как последовательная смена состояний в памяти компьютера (т.е. изменение значений переменных), в функциональном программировании нет понятия переменной и присваивания, функция не имеет явного внутреннего состояния, а оперирует только входными данными. Из-за этого отсутствуют побочные эффекты, программа становится более простой в отладке, а также допускает более естественное «распараллеливание» на многоядерных процессорах. В последнее время программные системы становятся всё сложнее. Возникает потребность в параллельном программировании, которое при использовании императивного подхода является очень сложной задачей, поэтому интерес к функциональному программированию быстро растёт и оно начинает всё больше использоваться в индустрии разработки ПО.

1.5. Логическое программирование

Логическое программирование (появилось в конце 60-х годов XX в.) – это парадигма программирования (и раздел дискретной математики), основанная на концепции системы правил. В логическом программировании программы выражены в виде формул математической логики, и решение задачи достигается путем вывода логических следствий из них. Теория формальных систем и, в частности, математическая логика являются формализацией человеческого мышления и представления наших знаний. Если предположить, что можно аксиоматизировать наши знания и построить алгоритм, позволяющий реализовать процесс вывода ответов на запрос из знаний, то в результате можно получить формальный метод для исследования неформальных результатов. Логическое программирование строится не с помощью некоторой последовательности абстракций и преобразований, отталкивающейся от машинной архитектуры фон Неймана и присущего ей набора операций, а на основе абстрактной модели, которая никак не связана с каким-либо типом машинной модели. *Логическое программирование базируется на убеждении, что не человека следует обучать мышлению в терминах операций компьютера (на опреде-*

ленном историческом этапе некоторые ученые и инженеры считали подобный путь простым и эффективным), а компьютер должен выполнять инструкции, свойственные человеку. В своем предельном и чистом виде логическое программирование предполагает, что сами инструкции не задаются, а вместо этого явно, в виде логических аксиом, формулируются сведения о задаче и предположения, достаточные для ее решения. Такое множество аксиом является альтернативой обычной программе. Подобная программа может выполняться при постановке задачи, формализованной в виде логического утверждения, подлежащего доказательству. Такое утверждение называется *целевым утверждением*. Выполнение программы состоит в попытке решить задачу, т.е. доказать целевое утверждение, используя предположения, заданные в логической программе. Логическое программирование возникло как результат естественного желания автоматизировать процесс логического вывода, поэтому оно является ветвью теории формальных систем. Логическое программирование (в широком смысле) представляет собой семейство таких методов решения задач, в которых используются приемы логического вывода для манипулирования знаниями, представленными в декларативной форме.

В основе логического программирования лежит описание задачи совокупностью утверждений на некотором формальном логическом языке и получение решения с помощью вывода в некоторой формальной (аксиоматической) системе (*аксиоматической системой* называется способ задания множества путем указания исходных элементов и правил вывода, которые описывают, как строить новые элементы из исходных элементов).

Центральное понятие логического программирования – ***отношение***. Программа логического программирования представляет собой совокупность определения отношений между объектами. В логическом программировании нужно только специфицировать факты, на которых основывается алгоритм, а не определять последовательность шагов, которые следует выполнить. Это свидетельствует о декларативности языка логического программирования, выражаемой высказыванием: «алгоритм = логика + управление». Языки логического программирования осуществляют проверку наличия необходимого разрешающего условия и в случае его обнаружения выполняют соответствующее действие. Выполнение программы на подобном языке похоже на выполнение программы, написанной на императивном языке. Однако операторы выполняются не в той последовательности,

в которой они определены в программе. Порядок выполнения определяют разрешающие условия. *Синтаксис таких языков выглядит следующим образом*:

разрешающее условие_1 → действие_1
разрешающее условие_2 → действие_2
разрешающее условие_n → действие_n

Наиболее известным логическим языком является Пролог (PROLOG – PROgramming in LOGic), созданный в 1971 г. во Франции в Марсельском университете Аленом Кольмеро.

Примечание. ***Краткие сведения о языке Пролог***

1. Пролог *– язык исчисления предикатов (исчисление – совокупность правил для определения истинности/ложности логических предложений). Программа на языке Пролог включает следующие основные разделы:* ***описание имен и структур объектов*** *(domains);* ***описание предикатов*** *– названий отношений между объектами (predicates);* ***раздел целевых утверждений*** *(goal).*

Предикат – логическая формула от одного или нескольких аргументов (предикат – это функция, отображающая любое множество в множество вида {ложно, истинно}).

Обозначение: F(x), F(x, y) F(x, y, z).

Факт «Коля работает слесарем» на Пролог запишется так: ***профессия(коля, слесарь).***

Факт «Борису 10 лет» можно представить в виде: ***возраст(«Борис», 10).***

2. Запросы*. Запрос (вопрос программе Пролог:* **?-***) – это последовательность из одного или множества предикатов, разделяемых запятыми и завершающаяся точкой:*

?- 5+4<3. /Запрос | No /Ответ ПРОЛОГ
'Сегодня солнечно'. /Факт|?- 'Сегодня солнечно'. /Запрос | Yes /Ответ ПРОЛОГ

3. Правила*. Правило отличается от факта тем, что факт всегда является истиной, а правило описывает утверждение, которое будет истиной, если выполнено некоторое условие.*

child(Y, X) :-parent (X, Y). *читается так: Для всех X и Y* ***Y -child X****, если* ***X -parent Y****.*

В правилах буква X (или любая другая заглавная буква, или любое слово, начинающееся с заглавной буквы) обозначает переменную, которая может принимать разные значения.

Так, правило ***город (X, европа) :- город (X, польша)*** *означает, что любой польский город является одновременно европейским городом, определенным в предметной области.*

4. Списки*. Список – это набор объектов одного и того же типа. При записи список заключают в квадратные скобки, а элементы списка разделяют запятыми:[1,2,3]; [1.1,1.2,3.6];[«вчера», «сегодня», «завтра»]. Непустой список может быть разделен на «голову» – первый элемент списка и «хвост» – остальные элементы списка.*

Операция разделения списка на голову и хвост обозначается с помощью вертикальной черты: описание списка (где ***X*** *– обозначает голову, а* ***Y*** *– хвост):* ***[X|Y]****.(список может содержать имена, числа и т.д). Если необходимо определить, содержится ли элемент в указанном списке, то в Пролог это можно сделать, определив, совпадает ли данный элемент с головой списка. Если совпадает, то задача решена. Если нет, то надо проверить, есть ли элемент в хвосте списка (снова проверяется голова, но относящаяся уже к хвосту списка, затем проверяется голова очередного хвоста списка и т.д.; если в конце список будет пустым, то указанного элемента в исходном списке нет). Для записи отношения принадлежности между объектом и списком можно использовать предикат: целевое утверждение* ***принадлежит(X, Y)*** *является истинным («выполняется»), если переменная (терм), связанная с* ***X****, является элементом списка, связанного с* ***Y****. Для определения истинности предиката необходимо проверить два условия:*

– первое условие: X будет элементом списка Y, если X совпадает с головой списка Y.

Запись: ***принадлежит(X,[X |_]).*** *– X является элементом списка, имеющего X в качестве головы. Здесь используется анонимная переменная '_' для обозначения хвоста списка;*

– второе условие: X будет элементом списка Y, если X входит в хвост этого списка Y.

Запись: ***принадлежит(X,[_ |Y]):- принадлежит(X,Y).*** *X является элементом списка, если X является элементом хвоста этого списка.*

Два этих условия в совокупности определяют предикат для отношения принадлежности и указывают, каким образом просматривать список от начала до конца при поиске некоторого элемента в списке.

Пролог – пример программы: определение истинности вхождения элемента в список:

MEMBER(X, [X|_]). /1-я строка
MEMBER (X, [_|T]) :- MEMBER (X, T). /2-я строка

Программа читается так: «X является членом списка, если он совпадает с головой списка (1-я строка), или является членом хвоста списка (2-я строка)».

В программе объявлен один предикат (отношение) – MEMBER.

Пролог-система работает в форме диалога с пользователем (через вопросы).

?- MEMBER (2, [1,2,3]). /Вопрос Пролог-системе: является ли **2** членом списка [1,2,3]?

YES Пролог-система вначале применяет 1-е условие для предиката MEMBER, сравнивая 2 с головой списка [1,2,3]. Это сравнение даст неверный результат, после этого система применяет 2-е условие, рекурсивно вызывающее предикат MEMBER, с аргументами 2 и [2,3]. В этом рекурсивном вызове сработает 1-е условие (так как 2 совпадает с головой списка [2,3]), и Пролог-система выдаст результат – YES

?- MEMBER (g,[a,b,c,d]). /Вопрос Пролог-системе: является ли **g** членом списка [a,b,c,d]?
NO /Ответ Пролог-системы

Можно констатировать, что по большому счету Пролог не занял в 90-е годы XX в. позиций, которые ему пророчили в начале 80-х годов. Большинство реальных задач относится к условиям неполноты и некорректности данных и знаний. Поэтому большинство существующих экспертных систем активно используют такие механизмы обработки неточных и неопределенных знаний, как нечеткая логика и нейронные сети. Тем не менее интерес к Пролог не ослабевает, поскольку логическое мышление остается основным инструментом умственной деятельности человека, поэтому эпоха массового использования Пролог просто наступит несколько позже.

1.6. Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) в последнее время – наиболее интенсивно развивающееся направление и попу-

лярное средство разработки программного обеспечения. Начало развитию ООП положил язык имитационного моделирования Simula 67, который был разработан в конце 1960-х гг. в Норвегии. На его основах построены современные объектно-ориентированные языки программирования: C++, C#, Object Pascal, Python, Ada, Smalltalk и др.

Объектно-ориентированное программирование (сокращают до ООП, появилось в начале 80-х годов XX в.) – парадигма программирования, основанная на концепции представления предметной области (и/или проблемной области) в виде системы взаимосвязанных абстрактных объектов (классов) и их реализаций (объектов – экземпляров классов).

В теории программирования ООП – технология создания сложного программного обеспечения, основанная на представлении программной системы 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 (рис. 1.1). В отличие от обычного процедурного программирования, в

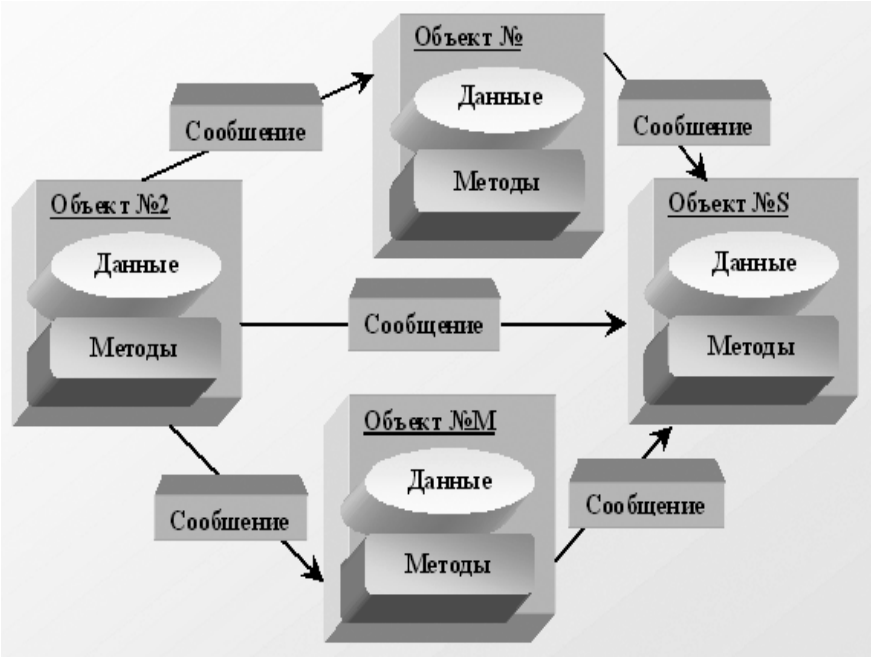

Рис. 1.1. Архитектура системы при ООП

ООП основными элементами программы являются не переменные и методы (процедуры и функции), а объекты и сообщения.

Основные идеи ООП опираются на следующие положения:

• программа представляет собой *модель* некоторого реального процесса, части реального мира; модель содержит не все признаки и свойства представляемой ею части реального мира, а только те, которые существенны для разрабатываемой программной системы;

• модель реального мира или его части могут быть описаны как совокупность взаимодействующих между собой *объектов*;

• объект описывается набором *атрибутов* (свойств), значения которых определяют состояние объекта, и набором *операций* (действий), которые может выполнять объект;

• взаимодействие между объектами осуществляется посылкой специальных *сообщений* от одного объекта к другому; сообщение, полученное объектом, может потребовать выполнения определенных действий, например изменения состояния объекта;

• объекты, описанные одним и тем же набором атрибутов и способные выполнять один и тот же набор операций, представляют собой *класс* однотипных объектов.

Классы отражают строение объектов реального мира (в «мире» ООП нет функций и данных, там «живут» объекты и только объекты) – каждый предмет или процесс обладает набором характеристик и отличительных черт (свойствами и поведением). Класс является определяемым пользователем абстрактным типом данных, описывающим свойства и поведение какого-либо предмета или процесса посредством полей данных (аналогично структуре), функций для работы с ними и сообщений (рис. 1.2).

Объект является автономным действующим лицом в системе. Реальными кандидатами на роли объектов выступают люди, места, вещи, организации, концепции и события. Объект обладает состоянием, поведением и идентичностью (***идентичность*** – это такое свойство объекта, которое отличает его от всех других объектов); структура и поведение схожих объектов определяют общий для них класс.

Взаимодействие объектов происходит с помощью ***сообщений***, которые передаются между объектами. Сообщение – некоторый набор информации (в том числе «пустой», так как сам факт прихода сообщения несет в себе информацию). Сообщение переносит информацию в обе стороны: в объект-получатель и из объекта-получателя. Но считается, что сообщением владеет тот объект, который его получает,

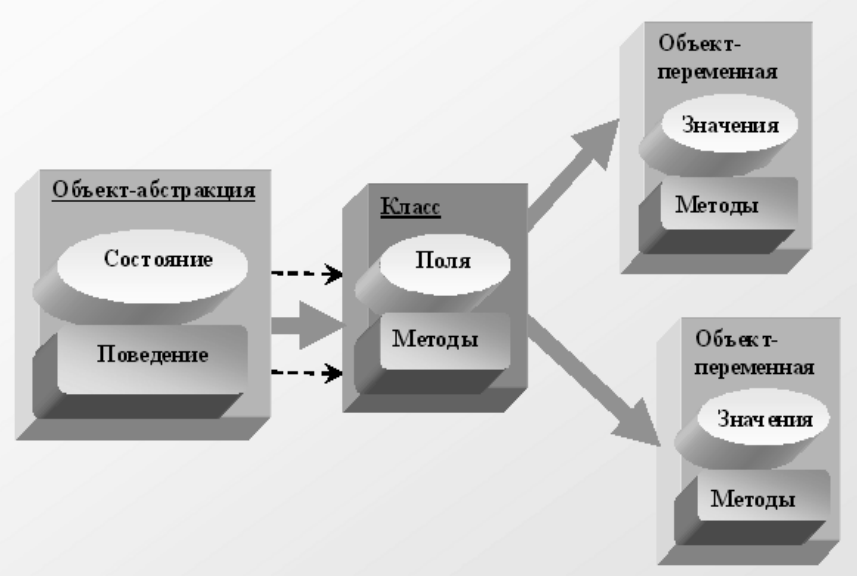

Рис. 1.2. Объект-абстракция, класс, объект-переменная (экземпляр)

обрабатывает информацию и отправляет результат обработки отправителю. Первоначально (Smalltalk) взаимодействие объектов представлялось как «настоящий» обмен сообщениями, т.е. пересылка от одного объекта к другому специального объекта-сообщения. Такая модель (общая) подходит для описания параллельных вычислений с помощью *активных объектов*, каждый из которых имеет собственный поток исполнения и работает одновременно с прочими. Такие объекты могут вести себя как отдельные, абсолютно автономные вычислительные единицы.

Однако общность механизма обмена сообщениями («полноценная» передача сообщений) требует дополнительных языковых и вычислительных затрат, что не всегда оправданно. Поэтому в большинстве современных объектно-ориентированных языков используется концепция «отправка сообщения как вызов метода» (сообщение – запрос на выполнение действия, содержащий набор необходимых параметров; механизм сообщений реализуется с помощью вызова соответствующих функций) – объекты имеют доступные извне методы, вызовами которых и обеспечивается взаимодействие объектов (данный подход в настоящее время наиболее распространён в объектно-ориентированных языках).

С помощью ООП реализуется *событийно-управляемая модель* (активные данные управляют вызовами фрагментов программного кода). Примером реализации событийно-управляемой модели служит любая программа, управляемая с помощью меню. После запуска такая программа пассивно ожидает действий пользователя и реагирует на любое из них. Событийная модель является противоположностью традиционной (директивной), когда код управляет данными: программа предлагает пользователю выполнить некоторые действия (ввести данные, выбрать режим) в соответствии с жестко заданным алгоритмом.

Принято выделять следующие приципы ООП: *абстрагирование, ограничение доступа, модульность, иерархичность, типизацию, параллелизм, устойчивость*.

Абстрагирование – процесс выделения абстракций в предметной области задачи (предметная область – это *абстрактное пространство*, в котором формулируется определенная задача, т.е. набор понятий, представляющих важные аспекты решаемой задачи). Абстракция – совокупность существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа. Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от несущественных. Такое разделение смысла и реализации называют *барьером абстракции*. Рассмотрим системный блок компьютера. При наборе текста на компьютере пользователю не важно, из каких частей состоит этот блок. Для него это – коробка белого цвета с кнопками и дисководом. Он абстрагируется от таких понятий, как «процессор» или «оперативная память». Для программиста, пишущего программы в машинных кодах, барьер абстракции лежит намного ниже. Ему необходимо знать устройство процессора и его команды. Современный уровень абстракции предполагает объединение всех свойств абстракции (как касающихся состояния анализируемого объекта, так и определяющих его поведение) в единую программную единицу – некий абстрактный тип (класс).

Ограничение доступа – сокрытие отдельных элементов реализации абстракции, не затрагивающих ее существенных характеристик. Необходимость ограничения доступа предполагает разграничение 2 частей описания абстракции: *интерфейс* – совокупность доступных извне элементов реализации абстракции (основные характеристики

состояния и поведения); *реализация* – совокупность недоступных извне элементов реализации абстракции (внутренняя организация абстракции и механизмы реализации ее поведения). Объединение всех свойств реального предмета (описывающих его состояние и поведение) в единую абстракцию и ограничение доступа к реализации этих свойств называется *инкапсуляцией*. Абстракция и инкапсуляция дополняют друг друга (абстрагирование направлено на наблюдение за объектом, инкапсуляция – на его внутреннее устройство).

Модульность – принцип разработки системы, предполагающий её реализацию в виде отдельных частей (модулей). При выполнении декомпозиции системы на модули желательно объединять логически связанные части, по возможности обеспечивая сокращение количества внешних связей между модулями. В ООП по модулям распределяют классы и объекты. Правильное разделение программы на модули – сложная проблема. Для простых задач допустимо наличие 1 модуля, но для большинства программ лучше будет сгруппировать логически связанные элементы в отдельный модуль, оставив открытыми только те элементы, которые необходимо «видеть» другим модулям.

Иерархичность (наличие множества элементов, расположенных на основе подчинения элементов низшего уровня элементам высшего уровня). *Иерархия* – ранжированная, или упорядоченная система абстракций (расположение абстракций по уровням). Принцип иерархичности предполагает использование иерархий при разработке программных систем. В ООП используются два вида иерархии.

1. *Иерархия* «*целое/часть*» (иерархия композиции и агрегирования) – показывает, что некоторые абстракции включены в рассматриваемую абстракцию как ее части (лампа состоит из цоколя, нити накаливания и колбы). Иерархия композиции описывает тесную связь между частями и целым: абстракции-части создаются, «живут» и уничтожаются вместе с абстракцией-целым; абстракция-часть может принадлежать только одной абстракции-целому (пример иерархии композиции: монитор содержит блок управления, корпус, TFT-панель, подставку монитора, кнопку монитора, интерфейсный кабель, силовой кабель). Иерархия агрегирования описывает менее тесную связь между частями и целым: абстракции-части разрешается добавлять и/или исключать из абстракции-целого; при этом абстракцию-целое обычно называют агрегатом. Исходя из возможности добавления/удаления из состава абстракции-целого (агрегата), абстракция-часть может разделяться несколькими агрегатами, т.е. входить в состав не-

скольких абстракций-целых. В процессе моделирования сущностных классов, как правило, используют иерархию агрегирования, в то время как иерархия композиции чаще всего используется при моделировании объектов имитационной модели. Такой тип иерархии используется в процессе разбиения системы на разных этапах проектирования (логический уровень – при декомпозиции предметной области на объекты, физический уровень – при декомпозиции системы на модули).

2. *Иерархия* «*общее/частное*» – показывает, что некоторая абстракция является частным случаем другой абстракции (обеденный стол – конкретный вид стола, столы – конкретный вид мебели). Такой тип иерархии используется при разработке структуры классов, когда сложные классы строятся на базе более простых путем добавления к ним новых характеристик и, возможно, уточнения имеющихся. Один из важнейших механизмов ООП – наследование свойств в иерархии общее/частное. Наследование – такое соотношение между абстракциями, когда одна из них использует структурную или функциональную часть другой или нескольких других абстракций (соответственно простое и множественное наследование).

Типизация – ограничение, накладываемое на свойства объектов и препятствующее взаимозаменяемости абстракций различных типов (или сильно сужающее возможность такой замены). В языках с жесткой типизацией для каждого программного объекта (переменной, подпрограммы, параметра и т.д.) объявляется тип, который определяет множество операций над соответствующим программным объектом (язык программирования C# используют строгую степень типизации). Тип может связываться с программным объектом статически (тип объекта определен на этапе компиляции – *раннее связывание)* и динамически (тип объекта определяется только во время выполнения программы – *позднее связывание).* Реализация позднего связывания в языке программирования позволяет создавать переменные – указатели на объекты, принадлежащие различным классам *(полиморфные объекты),* что расширяет возможности языка.

Параллелизм – свойство нескольких абстракций одновременно находиться в активном состоянии (выполнять некоторые операции). Существует целый ряд задач, для решения которых требуется одновременное выполнение некоторых последовательностей действий. К таким задачам, например, относятся задачи автоматического управления несколькими процессами. Реальный параллелизм достигается только при реализации задач такого типа на многопроцессорных

системах, когда имеется возможность выполнения каждого процесса отдельным процессором. Системы с одним процессором имитируют параллелизм путем разделения времени процессора между задачами управления различными процессами. Разделение времени выполняется используемой ОС (как в системах Windows и др).

Устойчивость – свойство абстракции существовать во времени независимо от процесса, породившего данный программный объект, и/или в пространстве, перемещаясь из адресного пространства, в котором он был создан. Классификация объектов: *временные* объекты (хранящие промежуточные результаты некоторых действий/вычислений), *локальные* объекты (существующие внутри подпрограмм, время жизни которых исчисляется от вызова подпрограммы до ее завершения), *глобальные* объекты (существующие, пока программа загружена в память), *сохраняемые* объекты (данные о которых хранятся в файлах внешней памяти между сеансами работы программы).

Все указанные принципы в той или иной степени реализованы в различных версиях объектно-ориентированных языков (язык программирования считается объектно-ориентированным, если в нем реализованы первые четыре из рассмотренных принципов).

При использовании технологии ООП решение представляется в виде *результата взаимодействия отдельных функциональных элементов* некоторой системы, имитирующей процессы, происходящие в предметной области поставленной задачи. В такой системе каждый функциональный элемент, получив некоторое входное воздействие (называемое *сообщением)* в процессе решения задачи, выполняет заранее определенные действия (может изменить собственное состояние, выполнить некоторые вычисления, нарисовать окно или график и в свою очередь воздействовать на другие элементы). Процессом решения задачи управляет *последовательность сообщений.* Передавая эти сообщения от элемента к элементу, система выполняет необходимые действия. Функциональные элементы системы, параметры и поведение которой определяются условием задачи, обладающие самостоятельным поведением (т.е. «умеющие» выполнять некоторые действия, зависящие от полученных сообщений и состояния элемента), получили название *объектов.* Процесс представления предметной области задачи в виде совокупности объектов, обменивающихся сообщениями, называется *объектной декомпозицией.*

Принципы построения объектной декомпозиции заключаются в следущем: