Лекция 1. Введение в объектно-ориентированное программирование

От структурного программирования к объектно-ориентированному

На протяжении всех лет своего существования практика программирования требовала совершенствования технологических приёмов и создания на их основе таких средств программирования, которые упростили бы процесс разработки программ, позволяя создавать всё более сложные программные системы.

Первые программы были организованы очень просто. Они состояли из программы на машинном языке и обрабатываемых данных. Сложность программ ограничивалась способностью программиста одновременно мысленно отслеживать последовательность выполняемых операций и местонахождение большого количества данных.

Появление сначала ассемблеров, а затем и языков высокого уровня сделало программу более обозримой за счёт снижения уровня детализации и естественно позволило увеличить её сложность.

Существенно снизило трудоёмкость разработки программ появление в языках программирования средств, позволяющих оперировать подпрограммами. Подпрограммы можно было сохранять и использовать в других программах. В результате были накоплены огромные библиотеки расчётных и служебных подпрограмм, которые по мере надобности вызывались 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.

Слабым местом такой архитектуры было то, что при увеличении количества подпрограмм возрастала вероятность искажения части глобальных данных какой-либо подпрограммой. Например, обычно 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, полученное при последнем делении отрезка в ходе работы подпрограммы.

Необходимость исключения таких ошибок привела к идее использования в подпрограммах локальных данных. И вновь сложность разрабатываемого программного обеспечения стала ограничиваться возможностью программиста отслеживать процессы обработки данных уже на новом уровне. К этому добавились проблемы согласования интерфейса при ведении разработки несколькими программистами. В результате встал вопрос создания технологии разработки сложных программных продуктов, снижающей вероятность появления ошибок.

Усилиями многих авторов такая технология была создана и получила название «структурное программирование»[I].

Структурное программирование представляет собой совокупность рекомендуемых технологических приёмов, охватывающих выполнение всех этапов разработки программного обеспечения.

Были сформулированы основные принципы выполнения разработки:

- *принцип нисходящей разработки*, рекомендующий на всех этапах вначале определять наиболее общие моменты, а затем поэтапно выполнять детализацию (что позволяет последовательно концентрировать внимание на небольших фрагментах разработки);
- собственно *структурное программирование*, рекомендующее определенные структуры алгоритмов и стиль программирования (чем нагляднее текст программы, тем меньше вероятность ошибки);
- *принцип сквозного структурного контроля*, предполагающий проведение содержательного контроля всех этапов разработки (чем раньше обнаружена ошибка, тем проще её исправить).

В основе структурного программирования лежит *декомпозиция* (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших (до 40–50 операторов) подпрограмм. В отличие от используемого ранее интуитивного подхода к декомпозиции, структурный подход требовал представления задачи в виде иерархии подзадач простейшей структуры, для получения которой рекомендовалось применять

[I] Дал У., Дейкстра Э., К.Хоор. Структурное программирование: Пер. с англ. - М.: Мир, 1975. - 247 с.
Хьюз Дж., Мичтом Дж. Структурный подход к программированию: Пер. с англ. - М.: Мир, 1980. - 278 с.

метод пошаговой детализации. С появлением других принципов декомпозиции (объектного, логического и т.д.) данный способ получил название *процедурной* декомпозиции.

Метод пошаговой детализации заключается в следующем:

1. определяется общая структура программы в виде одного из трёх вариантов: последовательного, условного или циклического выполнения подзадач.
2. каждая подзадача, в свою очередь, разбивается на подзадачи с использованием тех же структур.
3. процесс продолжается, пока на очередном уровне не получается простейшая подзадача, которая достаточно просто реализуется средствами используемого языка (1–2 управляющих оператора языка).

Пусть требуется разработать программу, которая в удобной для пользователя форме позволит записывать и затем находить телефоны различных людей и организаций. «Удобная» форма на современном уровне программирования предполагает общение программы с пользователем через «меню».

Анализ задачи показывает, что программу можно строить как последовательность подпрограмм. Следовательно, на первом шаге декомпозиции с использованием пошаговой детализации получаем:

Основная программа:
Начать работу.
Вывести меню на экран.
Ввести команду.
Выполнить цикл обработки вводимых команд.
Завершить работу.

Первые три подзадачи, выявленные на данном шаге, представляются несложными, поэтому на следующем шаге детализируем действие «Выполнить цикл обработки вводимых команд»:

Выполнить цикл обработки вводимых команд:
цикл-пока *команда ≠«завершить работу»*
Выполнить команду.
Ввести команду
все-цикл.

После этого детализируем операцию «Выполнить команду». Выполняем декомпозицию, используя сразу несколько конструкций ветвления:

Выполнить_команду:
если *команда= «открыть книжку»*
то *Открыть книжку*
иначе если *команда= «добавить»*
то *Добавить запись*
иначе если *команда= «найти»*
то *Найти запись*
все-если
все-если
все-если.

На этом шаге можно пока остановиться, так как оставшиеся действия достаточно просты. «Вложив» результаты пошаговой детализации, получим структурное представление алгоритма основной программы объемом не более 20–30 операторов:

Основная программа:

Начать работу.
Вывести меню на экран.
Ввести команду.
цикл-пока *команда ≠«завершить работу»*
если *команда= «открыть книжку»*
то *Открыть книжку*
иначе если *команда= «добавить»*
то *Добавить запись*
иначе если *команда= «найти»*
то *Найти запись*

все-если
все-если
все-если
Ввести команду
все-цикл
Завершить работу.

Окончательно, на первом уровне выделены подзадачи:

- вывести меню;
- ввести команду;
- открыть книжку;
- добавить запись;
- найти запись.

На следующем уровне определяются подзадачи задач второго уровня, например:

Открыть_книжку:
Ввести имя файла
если *существует файл Имя_книжки*
то *Открыть файл*
иначе *Вывести сообщение об ошибке*
все-если

На этом этапе получаем подзадачи:

- ввести имя файла;
- открыть файл.

Поступив аналогично с наиболее сложными подзадачами первого уровня, получаем схему двухуровневой алгоритмической декомпозиции задачи (рис. 1.1). На данной схеме показано, из каких подпрограмм будет состоять разрабатываемая система и взаимодействие последних по вызовам.

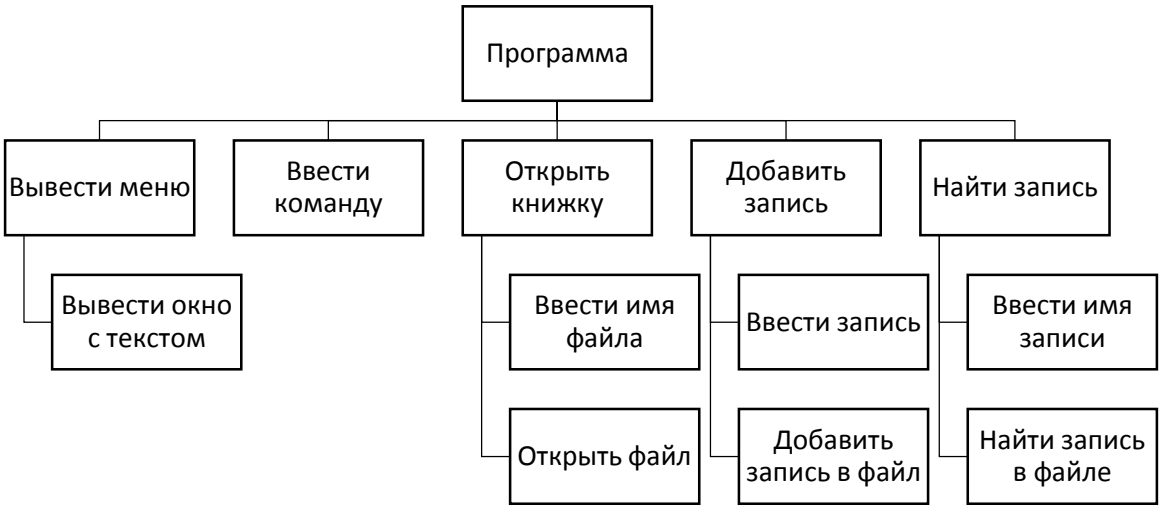

Рис. 1.1. Алгоритмическая декомпозиция системы «Записная книжка»

Созданная по результатам декомпозиции программа будет иметь правильную с точки зрения структурной технологии организацию, т. е. будет включать 12 относительно небольших подпрограмм, вызываемых из основной программы или подпрограмм более высокого уровня.

Сформулированная таким образом методика декомпозиции закрепила сложившийся в то время *процедурный* или *алгоритмический* подход к программированию, при котором основное внимание концентрируется на определении последовательности *действий*.

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

Дальнейший рост сложности и размеров разрабатываемого программного обеспечения потребовал развития *структурирования данных* и, соответственно, в языках появляется возможность определения пользовательских типов данных. Одновременно усиливается стремление разграничить доступ к глобальным данным программы для уменьшения количества ошибок. Результатом было появление и развитие технологии *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. Связи между модулями осуществляются через специальный интерфейс, в то время как доступ к реализации модуля (телам подпрограмм и некоторым «внутренним» переменным) запрещён. Эту технологию поддерживают современные версии языков Pascal и C (C++), языки Ада и Modula.

Использование модульного программирования существенно упрощает ведение разработки несколькими программистами, каждый из которых разрабатывает свои модули. Внутренняя организация модулей скрыта от остальных и потому может изменяться независимо. Взаимодействие модулей осуществляется через специально оговорённые интерфейсы модулей. Кроме того, модули в дальнейшем могут использоваться в других разработках, что увеличивает производительность труда программистов.

Практика программирования показывает, что структурный подход в сочетании с модульным программированием позволяет получать достаточно надёжные программы, размер которых не превышает 100000 операторов. Узким местом модульного программирования является 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 свыше 100000 операторов обычно возрастает сложность межмодульных интерфейсов, и предусмотреть взаимовлияние отдельных частей программы становится практически невозможно.

Стремление уменьшить количество связей между отдельными частями программы привело к появлению *объектно-ориентированного программирования* (ООП).

Основное отличие объектно-ориентированного программирования от структурного программирования можно считать более совершенный способ организации программы. В структурном программировании, как правило, программы организуются вокруг действий. Их можно рассматривать как код, действующий на данные. В объектно-ориентированном программировании программы организуются вокруг данных. Именно данные определяют, какие действия можно над ними выполнять.

Основные принципы и понятия объектно-ориентированного программирования

В теории программирования **объектно-ориентированное программирование (ООП)** определяется как технология создания сложного программного обеспечения, которая основана на представлении программы в виде совокупности *объектов*, каждый из которых является экземпляром определенного типа (*класса*), а классы образуют иерархию с *наследованием свойств*[I]. Взаимодействие программных объектов в такой системе осуществляется путем передачи *сообщений* от одного объекта к другому.

Основное достоинство ООП — сокращение количества межмодульных вызовов и уменьшение объёмов информации, передаваемой между модулями, по сравнению с модульным программированием. Это достигается за счёт более полной локализации данных и интегрирования их с подпрограммами обработки, что позволяет вести практически независимую разработку отдельных частей (объектов) программы.

Кроме этого, объектный подход предлагает новые технологические средства разработки, такие как, наследование, полиморфизм, композиция, наполнение, позволяющие конструировать более сложные объекты из более простых. В результате существенно увеличивается показатель повторного использования кодов,

---

[I] Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд. : Пер. с англ. - М.: Бином, СПб.: Невский диалект,1998. - 560 с.

появляется возможность создания библиотек объектов для различных применений, и разработчикам предоставляются дополнительные возможности создания систем повышенной сложности.

Основным недостатком ООП является некоторое снижение быстродействия за счет более сложной организации программной системы.

В основу ООП положены следующие принципы.

1. *Абстрагирование* — процесс выделения абстракций в предметной области задачи.

Абстракция — это совокупность существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, чётко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа. В соответствии с определением, применяемая абстракция реального предмета существенно зависит от решаемой задачи: в одном случае нас будет интересовать форма предмета, в другом — вес, в третьем — материалы, из которых он сделан, в четвёртом — закон движения предмета и т.д. 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).

2. *Ограничение доступа* — это сокрытие отдельных элементов реализации абстракции, не затрагивающих существенных характеристик её как целого.

   Необходимость ограничения доступа предполагает разграничение двух частей в описании 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 её поведения).

Ограничение доступа в ООП позволяет разработчику выполнять конструирование системы поэтапно, не отвлекаясь на особенности реализации используемых абстракций, и при этом, легко модифицировать реализацию отдельных объектов, что в правильно организованной системе не потребует изменения других объектов. Сочетание объединения всех свойств предмета (составляющих его состояния и поведения) в единую абстракцию и ограничения доступа к реализации этих свойств получило название **инкапсуляции**.

3. *Модульность* — это принцип разработки программной 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. Принцип унаследован от модульного программирования, следование ему упрощает проектирование и отладку программы.
4. *Иерархичность* предполагает использование иерархий при разработке программных систем.

Иерархия — это ранжированная или упорядоченная система абстракций. В ООП используются два вида иерархии:

1) *иерархия «целое — часть»* — показывает, что некоторые абстракции включены в рассматриваемую абстракцию, как её части, например, лампа состоит из цоколя, нити накаливания и колбы. Этот вариант иерархии используется в процессе разбиения системы на разных этапах проектирования: на логическом уровне — при декомпозиции предметной области на объекты; на физическом уровне — при декомпозиции системы на модули и при выделении отдельных процессов в мультипроцессной системе.
2) *иерархия «общее — частное»* — показывает, что некоторая абстракция является частным случаем другой абстракции, например, «обеденный стол — конкретный вид стола», «столы — конкретный вид мебели». Используется при разработке структуры классов, когда сложные классы строятся на базе более простых путём добавления к ним новых характеристик и, возможно, уточнения имеющихся.

   Один из важнейших механизмов ООП — наследование свойств в иерархии «общее — частное».

Наследование — это такое соотношение между абстракциями, когда одна из них использует структурную или функциональную часть другой или нескольких других абстракций (соответственно простое и множественное наследование).

5. *Типизация* — это ограничение, накладываемое на свойства объектов и препятствующее взаимозаменяемости абстракций различных типов (или сильно сужающее возможность такой замены). В языках с жёсткой типизацией для каждого программного объекта (переменной, подпрограммы, параметра и т. д.) объявляется тип, который определяет множество операций над соответствующим программным объектом.

   Использование принципа типизации обеспечивает:

- раннее обнаружение ошибок, связанных с недопустимыми операциями над программными объектами (ошибки обнаруживаются на этапе компиляции программы при проверке допустимости выполнения данной операции над программным объектом);
- упрощение документирования;
- возможность генерации более эффективного кода.

Тип может связываться с программным объектом статически (тип объекта определён 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 существенно расширяет возможности языка.

6. *Параллелизм* — свойство нескольких абстракций одновременно находиться в активном состоянии, т.е. выполнять некоторые операции.

Существует целый ряд задач, решение которых требует наличия одновременного выполнения 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 за счёт разделения времени процессора между задачами управления различными процессами. В зависимости от типа используемой операционной системы (одно- или мультипрограммной) разделение времени может выполняться либо разрабатываемой системой (как в MS DOS), либо используемой ОС (как в системах Windows).

7. *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.

   Различают:

- *временные* объекты, хранящие промежуточные результаты некоторых действий, например, вычислений;
- *локальные* объекты, существующие внутри подпрограмм, время жизни которых исчисляется от вызова подпрограммы до её завершения;
- *глобальные* объекты, существующие пока программа загружена в память;
- *сохраняемые* объекты, данные которых хранятся в файлах внешней памяти между сеансами работы программы.

Все указанные выше принципы в той или иной степени реализованы в различных версиях *объектно-ориентированных языков*. Язык считается объектно-ориентированным, если в нём реализованы первые четыре из рассмотренных семи принципов.

Кроме этого, в теории программирования принято различать *объектно-ориентированные* и *объектные* языки программирования. Последние отличаются тем, что они не поддерживают наследования свойств в иерархии абстракций. Несмотря на то, что принципиально ООП возможно на многих языках программирования, желательно для создания объектно-ориентированных программ использовать объектно-ориентированные языки, включающие специальные средства.

Во всех объектно-ориентированных языках основными понятиями являются «объект» и «класс».

Класс описывает модель какой-либо сущности. Фактически он описывает устройство объекта, являясь своего рода чертежом. **Объект** — это экземпляр класса. При этом, в некоторых системах класс также может представляться некоторым объектом при выполнении программы посредством динамической идентификации

типа данных. Обычно классы разрабатывают таким образом, чтобы их объекты соответствовали объектам предметной области.

В современных объектно-ориентированных языках программирования каждый объект является значением, относящимся к определённому классу. Класс представляет собой объявленный программистом составной тип данных, имеющий в составе:

- *поля данных* — параметры объекта, задающие его состояние. Физически поля представляют собой значения (переменные, константы), объявленные как принадлежащие классу;
- *методы* — процедуры и функции, связанные с классом. Они определяют действия, которые можно выполнять над объектом такого типа, и которые сам объект может выполнять;
- *свойства* — с точки зрения использования в программе аналогичны полям объекта, однако они не хранят данные, а позволяют управлять значениями полей объекта. Фактически, при обращении к свойству объекта вызывается связанная с ним подпрограмма, которая выполняет необходимые действия над полями объекта.

Обычно свойства и методы, которые описывают соответственно *характеристики* и *поведение* объектов, можно использовать в создаваемых на основе класса объектах, в то время как поля класса доступны только через его свойства и методы. Такой подход является хорошим стилем объектно-ориентированного программирования.

Платформа Microsoft .NET Framework и язык C#

.NET Framework — программная платформа, созданная Microsoft для разработки приложений под управлением Windows. Однако существуют альтернативные версии для работы под другими операционными системами. Одним из примеров является версия Mono, которая распространяется с открытым исходным кодом (вместе с компилятором C#) и способна работать под управлением нескольких операционных систем, включая различные варианты Linux и Mac OS. Помимо этого, существует и версия Microsoft .NET Compact Framework, которая, по сути, представляет собой усечённый вариант полной версии .NET Framework и может применяться в устройствах класса карманных персональных компьютеров (КПК) и даже в некоторых смартфонах. Одним из главных стимулов к применению платформы .NET Framework является возможность использовать её в качестве средства интеграции различных операционных систем.

Платформа .NET Framework спроектирована так, чтобы её можно было использовать из любого языка, включая C#, а также языков C++, Visual Basic, JScript и др. Для этого были выпущены специальные .NET-версии этих языков, и постоянно появляются новые. Все они не только имеют доступ к .NET Framework, но и могут взаимодействовать друг с другом. Это позволяет разработчикам на C# применять код, который был написан разработчиками на Visual Basic, и наоборот.

Платформа .NET Framework по большей части состоит из гигантской библиотеки кода, который можно использовать из клиентских языков (вроде C#) с помощью приёмов объектно-ориентированного программирования. Эта библиотека поделена на различные модули, которые применяются в зависимости от требуемых результатов. Например, в одном модуле содержатся компоновочные блоки для приложений Windows, в другом — для программирования сетевого обмена, в третьем — для разработки веб-приложений. Некоторые модули разбиты на более конкретные подмодули: к примеру, модуль для разработки веб-приложений содержит подмодуль для создания веб-служб.

В одном из разделов библиотеки .NET Framework содержатся определения ряда базовых *типов.* Типы — это представление данных, и указание некоторых наиболее фундаментальных из них (например, «32-битное целое число со знаком») способствует совместимости между языками, использующими .NET Framework. Это носит название *общей системы типов* (Common Type System — CTS).

Помимо библиотеки, в состав .NET Framework входит и *общеязыковая исполняющая среда* (Common Language Runtime — CLR) .NET, которая отвечает за обслуживание процесса выполнения всех приложений, разработанных с помощью библиотеки .NET.

Написание приложения с помощью .NET Framework означает написание кода с использованием одного из языков, поддерживающих .NET Framework, и кодовой библиотеки .NET. Мы будем использовать язык C#,

однако, все основные принципы создания приложений остаются справедливыми и для других .NET-ориентированных языков.

Чтобы код на языке C# мог выполняться, он должен быть преобразован в код на языке, понятном целевой операционной системе. Такой код называется *машинным кодом* (native code), а сам процесс преобразования — *компиляцией*. Компиляция выполняется механизмом, который называется *компилятором*, и в .NET Framework состоит из двух этапов.

При компиляции кода, в котором используется библиотека .NET, он не преобразуется сразу же в код для конкретной операционной системы. Вместо этого он сначала преобразуется в код *CIL* (Common Intermediate Language — общий промежуточный язык). Этот код не является специфическим ни для какой-либо операционной системы, ни для языка C#.

Перед запуском приложения вызывается так называемый *JIT-компилятор* (*Just-in-Time compiler* — оперативный компилятор), который компилирует CIL в машинный код, отвечающий требованиям конкретной операционной системы и архитектуры компьютера. Только после этого операционная система может запустить приложение. Аббревиатура *JIT* в названии компилятора означает, что CIL-код компилируется только при необходимости.

Раньше часто приходилось компилировать код в несколько приложений, каждое из которых предназначалось для конкретной операционной системы и архитектуры ЦП. Обычно так делали для оптимизации (например, чтобы код работал быстрее на микросхемах AMD), но в некоторых случаях это было необходимостью (например, если нужно было, чтобы приложения могли работать как в средах Win9x, так и в средах WinNT/2000). Сейчас такой необходимости нет, поскольку JIT-компиляторы используют CIL-код, который не зависит ни от компьютера, ни от операционной системы, ни от процессора. Существуют несколько JIT-компиляторов, каждый из которых рассчитан на конкретную архитектуру, и для создания машинного кода применяется тот, который подходит в данном случае. Такой механизм требует гораздо меньших усилий, поскольку разработчику не нужно думать о специфических деталях системы.

При компиляции приложения создаваемый CIL-код сохраняется в *сборке* (*assembly*). В состав сборок входят исполняемые файлы приложений, которые имеют расширение .exe и могут запускаться прямо в среде Windows безо всяких других программ, и файлы библиотек, которые имеют расширение .dll и предназначены для использования другими приложениями.

Помимо CIL-кода, сборки также содержат *метаинформацию* (т.е. информацию о содержащихся в сборке данных, также называемую *метаданными)* и файлы *ресурсов* (дополнительные данные, используемые в MSIL-коде, вроде звуковых файлов и изображений). Метаинформация позволяет сборкам быть полностью самостоятельными: для использования сборок не нужно никакой дополнительной информации, а это исключает ситуации вроде невозможности добавления требуемых данных в системный реестр и т.д., что часто было проблемой при разработках с помощью других платформ.

Процесс развёртывания приложений обычно сводится просто к копированию файлов в каталог на удалённом компьютере. Никакой дополнительной информации на целевых системах не требуется, поэтому пользователь может просто запустить исполняемый файл из этого каталога и (если в системе установлена CLR-среда .NET) приступить к работе с приложением.

При этом, вовсе необязательно хранить всё необходимое для работы приложения в одном месте. Разработчик может написать и какой-нибудь код, решающий задачи, требуемые несколькими приложениями. В подобных ситуациях зачастую удобно поместить многократно используемый код в место, доступное всем приложениям. В .NET Framework таким местом является *глобальный кэш сборок* (Global Assembly Cache — GAC). Помещение в него кода осуществляется очень просто: нужно просто сохранить сборку, содержащую нужный код, в каталог, где находится этот кэш.

Код, написанный с помощью .NET Framework, во время выполнения находится под управлением среды CLR. Это означает, что CLR-среда обслуживает приложения: управляет памятью, обеспечивает безопасность, позволяет выполнять межъязыковую отладку и т.д. Приложения, которые выполняются не под управлением CLR-среды, называются *неуправляемыми* (*unmanaged*), и некоторые языки позволяют создавать такие прило-

жения — например, для доступа к низкоуровневым функциям операционной системы. В языке C#, однако, разрешается писать только код, выполняющийся в управляемой среде, т.е. использовать управляемые средства CLR и позволять среде .NET самостоятельно взаимодействовать с операционной системой.

Одной из наиболее важных возможностей управляемого кода является *сборка мусора* (*garbage collection*). Она гарантирует в .NET полное освобождение использованной приложением памяти, когда она уже не нужна. До появления .NET это было в основном заботой программистов, и несколько простых ошибок в коде могли привести к таинственному исчезновению крупных блоков памяти в результате их неправильного выделения. Обычно это приводило к постепенному замедлению работы компьютера и в конечном итоге к краху системы.

Механизм сборки мусора в .NET время от времени проверяет память компьютера и удаляет из неё всё, что больше не нужно. Никакой периодичности для этой операции нет; она может выполняться тысячу раз в секунду, или каждые несколько секунд, или через любой другой промежуток времени, но она обязательно произойдет.

Таким образом, при создании .NET- приложения должны быть выполнены следующие действия:

1. Создание кода приложения на совместимом с .NET языке, например, C#.
2. Созданный код компилируется в CIL-код, который сохраняется в сборке.
3. Перед выполнением этого кода (в результате либо запуска как исполняемого файла, либо запроса из другого кода) он вначале компилируется в машинный код с помощью JIT-компилятора.
4. Скомпилированный машинный код выполняется в контексте управляемой CLR-среды вместе со всеми другими запущенными приложениями или процессами.

Язык C# является одним из языков, которые можно использовать для создания приложений, работающих в .NET CLR. Он был разработан Microsoft на базе языков C и C++ специально для работы с платформой .NET.

C# — достаточно мощный язык, и существует очень мало вещей, которые приходится делать на C++ из-за того, что их нельзя сделать на C#. Однако возможности C#, которые аналогичны более мощным возможностям языка C++, вроде прямых обращений к системной памяти, доступны только с помощью кода, помеченного как *небезопасный* (*unsafe*). Такие дополнительные приёмы программирования потенциально опасны, поскольку они позволяют перезаписать критические для системы блоки памяти — возможно, с катастрофическими последствиями.

В отличие от C++, C# представляет собой *безопасный к типам* (*type-safe*) язык. Это означает, что после назначения данных какому-то типу их затем нельзя преобразовать в другой непохожий тип. Из-за этого преобразования между типами подчиняются строгим правилам, и для решения той же самой задачи на C# часто требуется писать более объёмный код, чем на C++. Однако это даёт определённые преимущества: код получается более надежным, отладка — более простой, а .NET всегда может определить тип любого фрагмента данных. Поэтому в C# невозможно, например, взять область памяти размером 4 байта из данных длиной 10 байтов и интерпретировать это как X — хотя это не обязательно недостаток.

Язык C# — единственный язык, который с самого начала разрабатывался специально для .NET Framework, и поэтому он обязательно входит в состав версий .NET, переносимых на другие операционные системы. Чтобы языки вроде .NET-версии Visual Basic оставались максимально похожими на своих предшественников и всё же совместимыми с CLR, некоторые средства из кодовой библиотеки .NET в них поддерживаются не полностью. В отличие от них, C# позволяет пользоваться всеми средствами этой библиотеки. В последней версии .NET в язык C# было добавлено несколько расширений, которые еще более повышают его мощь.