 В основе того или иного языка программирования лежит некая руководящая идея, вызванная потребностями или, чаще всего, кризисом в области программирования и создания программного обеспечения, которая оказывает существенное влияние на стиль программирования и помогает преодолеть указанный кризис [9]. Рассмотрим вкратце историю появления и развития основных стилей программирования и процедурных алгоритмических языков.  
      ***Машинно-ориентированное программирование***появилось одновременно с созданием электронных вычислительных машин.Сначала это были программы в машинных кодах, затем появился язык программирования Assembler (Автокод), который немного «очеловечил» написание программы в машинном коде. Этот стиль программирования предполагает доскональное знание возможностей конкретной архитектуры ЭВМ и операционной системы и используется до сих пор тогда, когда другие стили бессильны, или нужно получить максимальное быстродействие в рамках той или иной операционной системы с использованием архитектуры данной ЭВМ.  
      ***Процедурное программирование.***Основная идея этого стиля – алгоритмизация процесса решения задачи и выбор наилучшего алгоритма (по расходу памяти или по быстродействию). Реализация этой идеи началась с 1957 года с появлением алгоритмических языков Fortran и затем Algol-60, когда все большее и большие число специалистов занялось решением достаточно сложных инженерных и научных задач. И нужен был стиль программирования максимально близкий к человеческому (математическому) стилю. При этом знание тонкостей архитектуры ЭВМ не требовалось. Программа на алгоритмическом языке (при наличии соответствующих трансляторов) должна была в идеале работать на ЭВМ любой архитектуры. Но это были программы прикладные. Разработку же системных программ (самих трансляторов, систем ввода-вывода) по-прежнему надо было делать на Ассемблере.  
      ***Структурное программирование.*** Здесь основная идея прекрасно выражена Н. Виртом в его книге "Алгоритмы + структуры данных = программы".Это был ответ на кризис в области программирования, начавшийся в середине 60-х годив, когда объем исходного программного кода перешел рубеж в 1000 строк. В 1971 году появился алгоритмический язык Pascal и немного позже, в 1972 году, язык С. Алгоритмические языки стали более мощными, более "интеллектуальными", на них уже можно было писать элементы трансляторов (компиляторов) и драйверов (подпрограмм обработки ввода-вывода). Компиляторы с языков С и Fortran выдают, например, по требованию программиста и листинг программы на Ассемблере. Знающий Ассемблер программистможет его проанализировать, что-то подправить и перекомпилировать, но уже на Ассемблере. В этом случае можно достаточно быстро и эффективно получать системные программы.  
     ***Модульное программирование.*** Здесь основная идея заключалась в том, чтобы "спрятать" данные и процедуры внутри независимых программных единиц - модулей*.*Эту идею впервые реализовал Н. Вирт в алгоритмическом языке Modula (1975-1979 годы), а затем "подхватили" и остальные, распространенные в то время языки программирования. Например, известные системы программирования Turbo Pascal и Turbo С.   
     На  этом можно было бы остановиться, т.к. я дошел темы моей работы, но я не могу обойти без внимания дальнейшее развитие технологий программирования, поэтому продолжу хронологию развития программирования.  
     ***Объектно-ориентированное программирование.*** С середины 80-х годов объем исходного программного кода перешел рубеж в 100 000 строк. Нужно было сделать не случайное объединение данных и алгоритмов их обработки в единое целое, а - смысловое. То есть необходимо было создать модульное программирование нового уровня,когда основной акцент делается на смысловую связь структур данных и алгоритмов их обработки. Сейчас практически все основные языки программирования (их более 100, в том числе такие распространенные, как Object Pascal, C++, Smalltalk) базируются на этой идее, а предком их является язык Simula, созданный еще в 1960 году.  
     ***Обобщенные  технологии разработки приложений.*** Идеология объектно-ориентированного программирования породила CASE-технологии разработки и сборки программ на основе уже известных программных моделей, содержащих интерфейсы и прототипы (шаблоны*—*template) данных: COM (Component Object Model), STL (Standard Template Library), ATL (Active Template Library). Все эти новшества поддерживают визуальныесреды разработки, например, такие известные, как Visual C++, Borland C++ Builder, Borland Delphi.  
     Теперь  подробно рассмотрим технологию модульного программирования.  
 

**Цель  модульного программирования**

     Приступая к разработке каждой программы, следует иметь в виду, что она, как правило, является большой системой, поэтому надо принять меры для ее упрощения. Для этого такую программу разрабатывают по частям, которые называются программными модулями. А сам такой метод разработки программ называют модульным программированием. Программный модуль - это любой фрагмент описания процесса, оформляемый как самостоятельный программный продукт, пригодный для использования в описаниях процесса. Это означает, что каждый программный модуль программируется, компилируется и отлаживается отдельно от других модулей программы, и тем самым, физически разделен с другими модулями программы. Более того, каждый разработанный программный модуль может включаться в состав разных программ, если выполнены условия его использования, декларированные в документации по этому модулю. Таким образом, программный модуль может рассматриваться и как средство борьбы со сложностью программ, и как средство борьбы с дублированием в программировании (т.е. как средство накопления и многократного использования программистских знаний).  
     Модульное программирование является воплощением в процессе разработки программ обоих общих методов борьбы со сложностью: и обеспечение независимости компонент системы и использование иерархических структур. Для воплощения первого метода формулируются определенные требования, которым должен удовлетворять программный модуль, т.е. выявляются основные характеристики «хорошего» программного модуля. Для воплощения второго метода используют древовидные модульные структуры программ (включая деревья со сросшимися ветвями).  
 

**Основные  характеристики программного модуля**

     Не  всякий программный модуль способствует упрощению программы. Выделить хороший с этой точки зрения модуль является серьезной творческой задачей. Для оценки приемлемости выделенного модуля используются некоторые критерии. Так, Хольт предложил следующие два общих таких критерия: 

хороший модуль снаружи проще, чем внутри;  
хороший модуль проще использовать, чем построить.

     Майерс  предлагает для оценки приемлемости программного модуля использовать более конструктивные его характеристики: 

размер модуля;  
прочность модуля;  
сцепление с другими модулями;  
рутинность модуля (независимость от предыстории обращений к нему).

      ***Размер*** модуля измеряется числом содержащихся в нем операторов или строк. Модуль не должен быть слишком маленьким или слишком большим. Маленькие модули приводят к громоздкой модульной структуре программы и могут не окупать накладных расходов, связанных с их оформлением. Большие модули неудобны для изучения и изменений, они могут существенно увеличить суммарное время повторных трансляций программы при отладке программы. Обычно рекомендуются программные модули размером от нескольких десятков до нескольких сотен операторов.  
      ***Прочность модуля*** - это мера его внутренних связей. Чем выше прочность модуля, тем больше связей он может спрятать от внешней по отношению к нему части программы и, следовательно, тем больший вклад в упрощение программы он может внести. Для оценки степени прочности модуля Майерс предлагает упорядоченный по степени прочности набор из семи классов модулей. Самой слабой степенью прочности обладает модуль, *прочный по совпадению*. Это такой модуль, между элементами которого нет осмысленных связей. Такой модуль может быть выделен, например, при обнаружении в разных местах программы повторения одной и той же последовательности операторов, которая и оформляется в отдельный модуль. Необходимость изменения этой последовательности в одном из контекстов может привести к изменению этого модуля, что может сделать его использование в других контекстах ошибочным. Такой класс программных модулей не рекомендуется для использования. Вообще говоря, предложенная Майерсом упорядоченность по степени прочности классов модулей не бесспорна. Однако, это не очень существенно, так как только два высших по прочности класса модулей рекомендуются для использования. Эти классы я рассмотрю подробнее.  
      *Функционально прочный* модуль - это модуль, выполняющий (реализующий) одну какую-либо определенную функцию. При реализации этой функции такой модуль может использовать и другие модули. Такой класс программных модулей рекомендуется для использования.  
      *Информационно прочный* модуль - это модуль, выполняющий (реализующий) несколько операций (функций) над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Для каждой из этих операций в таком модуле имеется свой вход со своей формой обращения к нему. Такой класс следует рассматривать как класс программных модулей с высшей степенью прочности. Информационно прочный модуль может реализовывать, например, абстрактный тип данных.  
      В модульных языках программирования как минимум имеются средства для задания функционально прочных  модулей (например, модуль типа FUNCTION в  языке ФОРТРАН). Средства же для задания  информационно прочных модулей  в ранних языках программирования отсутствовали. Эти средства появились только в более поздних языках. Так в языке программирования Ада средством задания информационно прочного модуля является пакет.  
      ***Сцепление модуля*** - это мера его зависимости по данным от других модулей. Характеризуется способом передачи данных. Чем слабее сцепление модуля с другими модулями, тем сильнее его независимость от других модулей. Для оценки степени сцепления Майерс предлагает упорядоченный набор из шести видов сцепления модулей. Худшим видом сцепления модулей является *сцепление по содержимому*. Таким является сцепление двух модулей, когда один из них имеет прямые ссылки на содержимое другого модуля (например, на константу, содержащуюся в другом модуле). Такое сцепление модулей недопустимо. Не рекомендуется использовать также *сцепление по общей области* - это такое сцепление модулей, когда несколько модулей используют одну и ту же область памяти. Такой вид сцепления модулей реализуется, например, при программировании на языке ФОРТРАН с использованием блоков COMMON. Единственным видом сцепления модулей, который рекомендуется для использования современной технологией программирования, является *параметрическое сцепление*(сцепление по данным по Майерсу) - это случай, когда данные передаются модулю либо при обращении к нему как значения его параметров, либо как результат его обращения к другому модулю для вычисления некоторой функции. Такой вид сцепления модулей реализуется на языках программирования при использовании обращений к процедурам (функциям).  
      ***Рутинность  модуля*** - это его независимость от предыстории обращений к нему. Модуль называется *рутинным*, если результат (эффект) обращения к нему зависит только от значений его параметров (и не зависит от предыстории обращений к нему). Модуль называется *зависящим от предыстории*, если результат (эффект) обращения к нему зависит от внутреннего состояния этого модуля, изменяемого в результате предыдущих обращений к нему. Майерс не рекомендует использовать зависящие от предыстории (непредсказуемые) модули, так как они провоцируют появление в программах хитрых (неуловимых) ошибок. Однако такая рекомендация является неконструктивной, так как во многих случаях именно зависящий от предыстории модуль является лучшей реализаций информационно прочного модуля. Поэтому более приемлема следующая (более осторожная) рекомендация: 

всегда  следует использовать рутинный модуль, если это не приводит к плохим (не рекомендуемым) сцеплениям модулей;  
зависящие от предыстории модули следует использовать только в случае, когда это необходимо для обеспечения параметрического сцепления;  
в спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость таким образом, чтобы было возможно прогнозировать поведение (эффект выполнения) данного модуля при разных последующих обращениях к нему.

     В связи с последней рекомендацией  может быть полезным определение  внешнего представления (ориентированного на информирование человека) состояний  зависящего от предыстории модуля. В этом случае эффект выполнения каждой функции (операции), реализуемой этим модулем, следует описывать в терминах этого внешнего представления, что существенно упростит прогнозирование поведения данного модуля.  
 

**3. Проектирование модуля**

     Модульное проектирование относится к процессу расчленения больших проблем  на более узкие, более управляемые подпроблемы. Первым шагом проектирования является решение, в каком месте должна быть граница между этими подпроблемами.  
     Для получения максимальных преимуществ  от использования модульного программирования каждая подпроблема или модуль должны иметь один вход и один выход. В этом случае можно легко отслеживать поток управления в программе. В любом месте модуля должна иметься возможность увидеть точку входа в модуль и определить точное значение переменных и регистров в этой точке, а затем проследить функционирование модуля без тревоги об искажении программы. Один вход обеспечивает возврат потока управления в точку вызова. По этой причине, модульные программы почти всегда выполняются как структуры «CALL - RET».  
     Использование нескольких предложений RET в модуле не должно нарушать правило одного входа, поскольку все инструкции RET возвращают управление в одну и туже точку. Точно также, переход к общему RET в конце модуля, не изменяет его структуру, а добавляет лишь коды и увеличивает его сложность. С другой стороны, вход или выход из модуля не по этому правилу перечеркивает наибольшие преимущества модульного программирования: ясность и удобство сопровождения.  
     Имеется исключение из правила входа в  модуль. Это происходит при использовании  таблицы переходов для реализации потока управления внутри программы. Таблица перехода используется путем «проталкивания» адреса возврата в стек, вычисления индекса требуемого адреса перехода в таблице и выполнения перехода в памяти.  
     При практическом выполнении декомпозиции модулей можно самим найти некоторое количество альтернативных решений. Прежде чем осуществить правильный выбор, необходимо знать альтернативы. Цель состоит в выборе таких вариантов, которые создадут наилучшие условия проектирования.

***3.1. Функциональная декомпозиция***

     При обращении к проблеме на стадии проектирования первым альтернативным выбором должна быть функциональная декомпозиция, т.е. разбиение проблемы на более узкие, вполне поддающиеся управлению функциональные единицы, где каждая единица выполняет завершенную, легко идентифицируемую задачу. Имеется множество путей определения содержания задачи. Вот лишь некоторые примеры подобных единиц, которые выполняют определенные функции: получение квадратного корня некоторого числа; выполнение всех операций относительно указанного устройства таких, как операции в/в диска, операции в/в клавиатуры; выполняющие общую группу действий в указанное время такие, как инициализация областей данных; и единицы, которые взаимодействуют последовательно или используют общие элементы данных такие, как считывание данных с клавиатуры и преобразование их в целые значения.  
     В настоящее время в области  программирования на языках высокого уровня чаще всего принимаются такие решения, которые представляют собой наилучший способ по использованию сегментации программ. Часто обнаруживается, что некоторые модули связываются с помощью одного набора критериев, а другие модули - с помощью другого. Каждый модуль должен включать легко понимаемые программные секции.

***3.2. Минимизации количества передаваемых параметров***

     Иногда  обнаруживается, что после определения  модулей программы создано нечто громоздкое и неуклюжее. Это часто случается тогда, когда модули при выполнении возложенных на них задач требуют доступа к обширному количеству данных. Чаще всего это легко может произойти, если на модуль возложено выполнение нескольких опций. Чтобы знать состояние программы в данное время, модуль должен принимать очень много различных переменных. Если это так, и выявлено, что модуль принимает большое количество параметров, необходимо ответить на следующие две группы вопросов:

В этом модуле предпринята попытка выполнения нескольких функций? Требует ли модуль параметры, используемые в не относящихся к данному модулю секциях? Если ответы на эти вопросы положительные, то необходимо снова обратиться к дальнейшей сегментации этого модуля.  
Модуль представляет собой функциональный разрез? Являются ли на самом деле вызывающий и вызываемый модули частью одной и той же функции? Если это так, то поместите их вместе в один модуль, даже если результирующий модуль окажется слишком большим. Затем попробуйте выполнить сегментацию модуля снова различными способами.

     Сегментация модулей через функциональный разрез часто происходит тогда, когда программист обнаруживает, что две программные секции идентичны или сильно похожи друг на друга. Программист затем пытается создать из них один модуль. Это не модульное программирование, поскольку результирующий модуль имеет не функциональное соединение.  
     Если  в процессе проектирования будет обнаружено, что ничего сделать нельзя, чтобы избежать использования большого числа ссылок на данные или передачи меток параметров, надо вернуться обратно в начало проектирования и проверить корректность поставленной проблемы.

***3.3. Минимизации количества необходимых вызовов***

     Одним из существенных преимуществ модульного программирования является то, что программа основного уровня очень часто может быть сконструирована для чтения как последовательность вызываемых процедур. Этот факт существенно повышает «понимаемость» программы, поскольку читатель может познакомиться с ее основным потоком и функционированием после прочтения только одной - двух страниц программного кода. Однако эта особенность может также иметь и недостатки. Одна из многих верхних статистических оценок программирования говорит о том, что 90% времени выполнения типовых программ расходуется в 10 % кода программы. При этом подразумевается, что если эти 10 % содержат большое количество цепочечных вызовов процедур, то суммарное время, затрачиваемое на управление выполнением программы, может стать непреодолимым препятствием на пути использования этого подхода.  
     Прежде  чем отказаться от модульности проектируемой  программы, надо проверить, что скрывается под зависимостью программы от времени. Во-первых, большинство программ затрачивают большую часть времени выполнения на ожидание ввода информации с клавиатуры. После нажатия клавиши требуемые функции, с точки зрения выполнения длительного процесса, обычно не расходуют время. Различие между 100 микросекунд и 100 миллисекунд для среднего пользователя неразличимо.  
     Противоположным для некоторых мнением является то, что действующий механизм пары CALL - RET не перекрывает потребляемое время. По сравнению с инструкциями перехода инструкция CALL выполняется на 30-50% дольше, а RET в среднем длиннее на 1 цикл. Только когда во внимание принимаются накладные расходы передачи параметров, сохранения регистров и т.д., называемые служебными расходами, модульные программы начинают выглядеть медленнее по сравнению с немодульными программами. В дополнение к тому, что модули модульных программ обычно являются более общими, чем их неструктурированные дубликаты, модули модульных программ могут использовать ссылки на память или стек с большей частотой. Дополнительное время, расходуемое на вычисление действительного адреса в теле модуля, может привести к замедлению выполнения конкретного модуля, чем узко закодированная конкретная программа.  
     Преимущества  служебных программ и программ общего назначения заключаются в том, что модуль может быть использован виртуально в некотором месте программы. При написании немодульной программы программист может потратить несколько часов, пытаясь открыть: используется ли регистр (переменная), или хуже того, верно ли то, что он должен использоваться. При модульном программировании программист не интересуется тем, какие регистры (переменные) он использует в настоящий момент, пока вызываемый модуль копирует его параметры в стек и сохраняет весь набор регистров на входе. Эти особенности создают возможность сначала использовать приемы модульного программирования для повышения скорости кодирования и затем переработки программы для удаления "узких" мест.  
     Для областей, чувствительных к скорости работы, лучшей рекомендацией является выбор основной ветви программы. Если модуль упоминается только в чувствительной к скорости работы программной секции, то он может быть включен в "ветвь" внутри вызывающего модуля. Если другие секции используют модуль, то они могут быть скопированы в вызывающий модуль в необходимое место. В связи с тем, что основной вызывающий модуль станет большим, необходимо вставить комментарии в его тело, помечающие включаемый модуль как блок его владельца. Будущие читатели смогут затем прочитать комментарии для определения функций модуля и пропустить его мимо для возобновления чтения основного кода.  
 

**4. Методы разработки  структуры модульной  программы**

     Как уже отмечалось выше, в качестве модульной структуры программы  принято использовать древовидную  структуру, включая деревья со сросшимися ветвями. В узлах такого дерева размещаются программные модули, а направленные дуги (стрелки) показывают статическую подчиненность модулей, т.е. каждая дуга показывает, что в тексте модуля, из которого она исходит, имеется ссылка на модуль, в который она входит. Другими словами, каждый модуль может обращаться к подчиненным ему модулям, т.е. выражается через эти модули. При этом модульная структура программы, в конечном счете, должна включать и совокупность спецификаций модулей, образующих эту программу. Спецификация программного модуля содержит:

синтаксическую спецификацию его входов, позволяющую построить на используемом языке программирования синтаксически правильное обращение к нему (к любому его входу);  
функциональную спецификацию модуля (описание семантики функций, выполняемых этим модулем по каждому из его входов).

     В процессе разработки программы ее модульная  структура может по-разному формироваться  и использоваться для определения  порядка программирования и отладки  модулей, указанных в этой структуре. Поэтому можно говорить о разных методах разработки структуры программы. Обычно в литературе обсуждаются два метода: метод восходящей разработки и метод нисходящей разработки.

***4.1. Метод восходящей разработки***

     Данный  метод 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 принципе в таком же (восходящем) порядке, в каком велось их программирование. Такой порядок разработки программы на первый взгляд кажется вполне естественным: каждый модуль при программировании выражается через уже запрограммированные непосредственно подчиненные модули, а при тестировании использует уже отлаженные модули. Однако, современная технология не рекомендует такой порядок разработки программы. Во-первых, для программирования какого-либо модуля совсем не требуется наличия текстов используемых им модулей - для этого достаточно, чтобы каждый используемый модуль был лишь специфицирован (в объеме, позволяющем построить правильное обращение к нему), а для тестирования его возможно (и даже, как мы покажем ниже, полезно) используемые модули заменять их имитаторами (заглушками). Во-вторых, каждая программа в какой-то степени подчиняется некоторым внутренним для нее, но глобальным для ее модулей соображениям (принципам реализации, предположениям, структурам данных и т.п.), что определяет ее концептуальную целостность и формируется в процессе ее разработки. При восходящей разработке эта глобальная информация для модулей нижних уровней еще не ясна в полном объеме, поэтому очень часто приходится их перепрограммировать, когда при программировании других модулей производится существенное уточнение этой глобальной информации (например, изменяется глобальная структура данных). В-третьих, при восходящем тестировании для каждого модуля (кроме головного) приходится создавать ведущую программу (модуль), которая должна подготовить для тестируемого модуля необходимое состояние информационной среды и произвести требуемое обращение к нему. Это приводит к большому объему «отладочного» программирования и в то же время не дает никакой гарантии, что тестирование модулей производилось именно в тех условиях, в которых они будут выполняться в рабочей программе.

***4.2. Метод нисходящей разработки***

     Метод нисходящей разработки заключается  в следующем. Как и в предыдущем методе сначала строится модульная  структура программы в виде дерева. Затем поочередно программируются  модули программы, начиная с модуля самого верхнего уровня (головного), переходя 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. При этом первым тестируется головной модуль программы, который представляет всю тестируемую программу и поэтому тестируется при «естественном» состоянии информационной среды, при котором начинает выполняться эта программа. При этом те модули, к которым может   
и т.д.................