Методология разработки программного модуля.

В условиях индустриального подхода к разработке и сопровождению программного обеспечения особый вес приобретают технологические характеристики разрабатываемых программ. Для обеспечения необходимых технологических свойств применяют специальные технологические приемы: правила декомпозиции, методы проектирования, программирования и контроля качества, которые под общим названием "структурный подход к программированию» были сформулированы еще в 60-х годах XX в. В его основу были положены следующие основные концепции:

- нисходящая разработка;
- модульное программирование;
- структурное программирование;
- сквозной структурный контроль.

Понятие технологичности программного обеспечения

Технологичность - качество проекта программного продукта, от которого зависят трудовые и материальные затраты на его реализацию и последующие модификации.

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

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

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

Стиль программирования, под которым понимают стиль оформления программ и их «структурность», также существенно влияет на читаемость программного кода и количество ошибок программирования. Кризис 60-х годов XX в. был вызван в том числе и стилем программирования, при котором программа напоминала клубок спутанных ниток или блюдо спагетти, и отсутствием языковых конструкций поддержки «структурного» стиля.

Модули и их свойства

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

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

Модулем называют *автономно компилируемую* программную единицу.

Термин «модуль» традиционно используется в двух смыслах. Первоначально, когда размер программ был сравнительно невелик, и все подпрограммы компилировались отдельно, под модулем понималась подпрограмма, т. е. *последовательность связанных фрагментов программы, обращение к которой выполняется по имени.* Со временем, когда размер программ значительно вырос, и появилась возможность создавать библиотеки ресурсов: констант, переменных, описаний типов, классов и подпрограмм, термин «модуль» стал использоваться и в смысле *автономно компилируемый набор программных ресурсов.*

Данные, модуль может получать и/или возвращать через общие области памяти или параметры. Первоначально к модулям предъявлялись следующие требования:

- отдельная компиляция;
- одна точка входа;
- одна точка выхода;
- соответствие принципу вертикального управления;
- возможность вызова других модулей;
- небольшой размер (до 50-60 операторов языка);
- независимость от истории вызовов;
- выполнение одной функции.

Уменьшение зависимости модулей улучшает технологичность проекта. Степень независимости модулей оценивают двумя критериями: сцеплением и связностью.

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

Различают пять типов сцепления модулей:

- по данным;
- по образцу;
- по управлению;
- по общей области данных;
- по содержимому.

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

В случае *сцепления по содержимому* один модуль содержит обращения к внутренним компонентам другого, что полностью противоречит блочно-иерархическому подходу. Отдельный модуль в этом случае уже не является блоком («черным ящиком»): его содержимое должно учитываться в процессе разработки другого модуля.

Современные универсальные языки процедурного программирования, например, Pascal, данного типа сцепления в явном виде не поддерживают, но для языков низкого уровня, например Ассемблера, такой вид сцепления остается возможным.

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

Виды связности (в порядке убывания уровня):

- функциональная;
- последовательная;
- информационная (коммуникативная);
- процедурная;
- временная;
- логическая;
- случайная.

При хорошо продуманной декомпозиции модули верхних уровней иерархии имеют функциональную или последовательную связность функций и данных.

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

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

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

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

В качестве средства улучшения технологических характеристик библиотек ресурсов в настоящее время широко используют разделение тела модуля на интерфейсную часть и область реализации (секции Interface и Implementation - в Pascal, h и cpp-файлы в C++ и в Java).

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

Нисходящая и восходящая разработка программного обеспечения

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

- восходящий;
- нисходящий.

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

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

Для тестирования и отладки компонентов проектируют и реализуют специальные тестирующие программы.

Подход имеет следующие недостатки:

- увеличение вероятности несогласованности компонентов вследствие неполноты спецификаций;
- наличие издержек на проектирование и реализацию тестирующих программ, которые нельзя преобразовать в компоненты;
- позднее проектирование интерфейса, а соответственно невозможность продемонстрировать его заказчику для уточнения спецификаций и т. д.

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

Нисходящий подход предполагает, что проектирование и последующая реализация компонентов выполняется «сверху-вниз», т. е. вначале проектируют компоненты верхних уровней иерархии, затем следующих и так далее до самых нижних уровней.

В той же последовательности выполняют и реализацию компонентов. При этом в процессе программирования компоненты нижних, еще не реализованных уровней заменяют специально разработанными отладочными модулями - «заглушками», что позволяет тестировать и отлаживать уже реализованную часть.

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

Иерархический метод предполагает выполнение разработки строго по уровням. Исключения допускаются при наличии зависимости по данным, т. е. если обнаруживается, что некоторый модуль использует результаты другого, то его рекомендуется программировать после этого модуля.

Основной проблемой данного метода является большое количество достаточно сложных заглушек. Кроме того, при использовании данного метода основная масса модулей разрабатывается и реализуется в конце работы над проектом, что затрудняет распределение человеческих ресурсов.

Операционный метод связывает последовательность выполнения при запуске программы. Применение метода усложняется тем, что порядок выполнения модулей может зависеть от данных. Кроме того, модули вывода результатов, несмотря на то, что они вызываются последними, должны разрабатываться одними из первых, чтобы не проектировать сложную заглушку, обеспечивающую вывод результатов при тестировании.

С точки зрения распределения человеческих ресурсов сложным является начало работ, пока не закончены все модули, находящиеся на так называемом *критическом* пути.

Комбинированный метод учитывает следующие факторы, влияющие на последовательность разработки:

- достижимость модуля - наличие всех модулей в цепочке вызова данного модуля;
- зависимость по данным - модули, формирующие некоторые данные, должны создаваться раньше обрабатывающих;
- обеспечение возможности выдачи результатов - модули вывода результатов должны создаваться раньше обрабатывающих;
- готовность вспомогательных модулей - вспомогательные модули, например, модули закрытия файлов, завершения программы, должны создаваться раньше обрабатывающих;
- наличие необходимых ресурсов.

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

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

Нисходящий подход обычно используют и при объектно-ориентированном программировании.

Нисходящий подход обеспечивает:

- максимально полное определение спецификаций проектируемого компонента и согласованность компонентов между собой;
- раннее определение интерфейса пользователя, демонстрация которого заказчику позволяет уточнить требования к создаваемому программному обеспечению;
- возможность нисходящего тестирования и комплексной отладки.

Эффективными считают программы, требующие минимального времени выполнения и/или минимального объема оперативной памяти. Особые требования к эффективности ПО предъявляют при наличии ограничений (на время реакции системы, на объем оперативной памяти и т. п.). В случаях, когда обеспечение эффективности *не требует серьезных временных и трудовых затрат, а также не приводит к существенному ухудшению технологических свойств,* необходимо это требование иметь в виду.

Частично проблему эффективности программ решают за программиста компиляторы. Средства оптимизации, используемые компиляторами, делят на две группы:

- *машинно-зависимые*, т. е. ориентированные на конкретный машинный язык, выполняют оптимизацию кодов на уровне машинных команд, например, исключение лишних пересылок, использование более эффективных команд и т. п.;
- *машинно-независимые* выполняют оптимизацию на уровне входного языка, например, вынесение вычислений константных (независящих от индекса цикла) выражений из циклов и т. п.

Принятие мер по экономии памяти предполагает, что в каких-то случаях эта память неэкономно использовалась. Учитывая, что анализировать имеет смысл только операции размещения данных, существенно влияющие на характеристику эффективности, следует обращать особое внимание на выделение памяти под данные структурных типов (массивов, записей, объектов и т. п.).

Прежде всего при наличии ограничений на использование памяти следует выбирать алгоритмы обработки, *не требующие дублирования исходных данных структурных типов в процессе обработки.* Примером могут служить алгоритмы сортировки массивов, выполняющие операцию в заданном массиве, например, сортировка методом «пузырька».

Если в программе необходимы большие массивы, используемые ограниченное время, то их можно размещать в динамической памяти и удалять при завершении обработки.

Также следует помнить, что при передаче структурных данных в подпрограмму «по значению» копии этих данных размещаются в стеке. Избежать копирования иногда удается, если передавать данные «по ссылке», но как неизменяемые (описанные const). В последнем случае в стеке размещается только адрес данных, например:

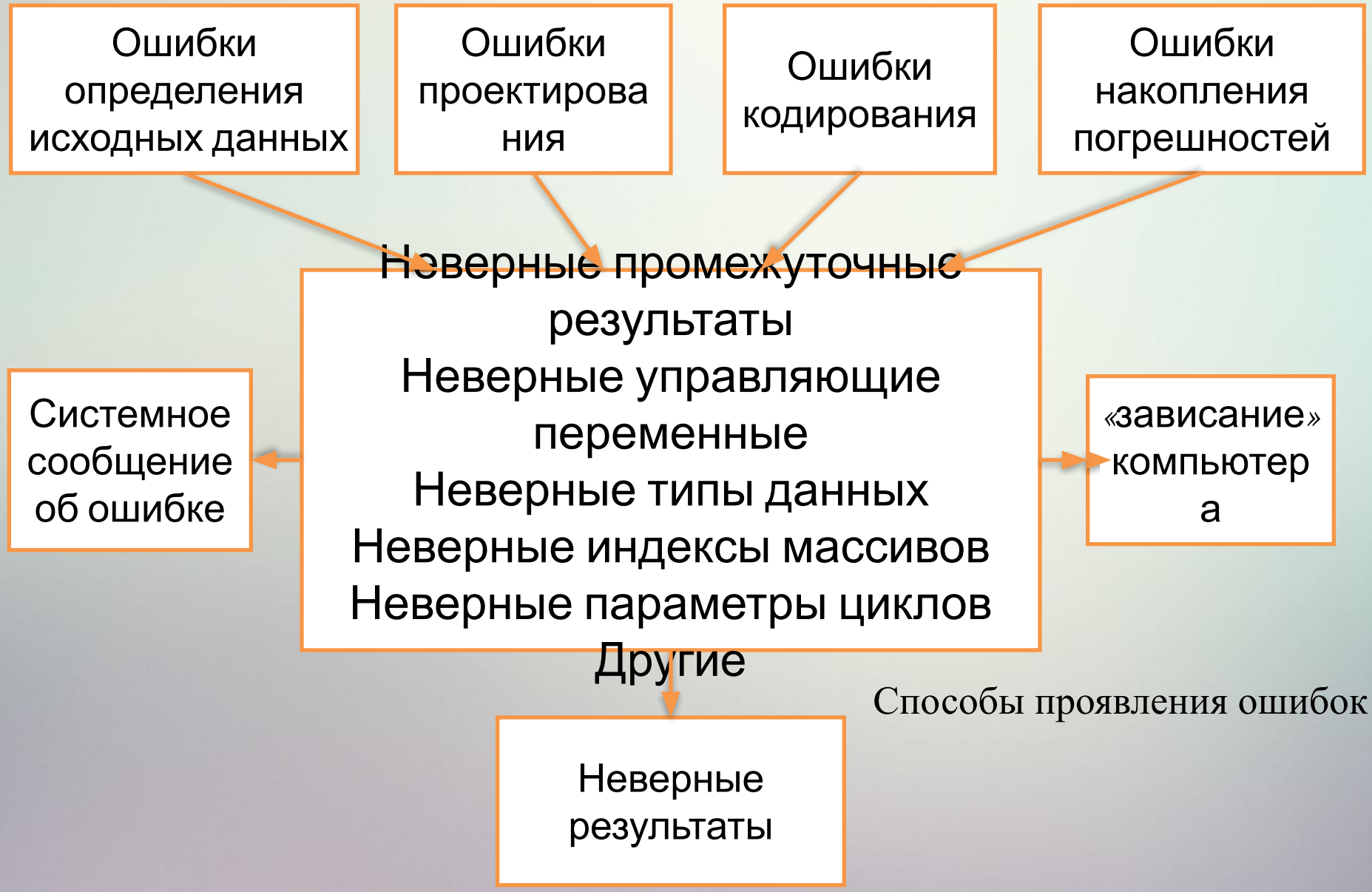
```
Type Mas.4iv = array [I.. 100] of real;


function Summa (Const a:Massiv; ...)...
```

Для уменьшения времени выполнения необходимо анализировать циклические участки программы с большим количеством повторений. При их написании необходимо по возможности:

- избегать «длинных» операций умножения и деления, заменяя их сложением, вычитанием и сдвигами;
- выносить вычисление константных;
- минимизировать преобразования типов в выражениях;
- оптимизировать запись условных выражений - исключать лишние проверки;
- исключать многократные обращения к элементам массивов по индексам - первый раз прочитав из памяти элемент массива, следует запомнить его в скалярной переменной и использовать в нужных местах;
- избегать использования различных типов в выражении и т. п.

Программирование «с защитой от ошибок»

Ошибка программирования, которая не обнаруживается на этапах компиляции и компоновки программы, может проявиться тремя способами: привести к выдаче системного сообщения об ошибке, «зависанию» компьютера и получению неверных результатов.



Способы проявления ошибок

Программирование, при котором применяют специальные приемы раннего обнаружения и нейтрализации ошибок, названо *защитным* или *программированием с защитой от ошибок.*

Детальный анализ ошибок и их возможных ранних проявлений показывает, что целесообразно проверять:

- правильность выполнения операций ввода-вывода;
- допустимость промежуточных результатов (значений управляющих переменных, значений индексов, типов данных, значений числовых аргументов и т. д.).

Принято различать:

- *ошибки передачи* - аппаратные средства, например, вследствие неисправности, искажают данные;
- *ошибки преобразования* - программа неверно преобразует исходные данные из входного формата во внутренний;
- *ошибки перезаписи* - пользователь ошибается при вводе данных, например, вводит лишний или другой символ;
- *ошибки данных* - пользователь вводит неверные данные. Ошибки передачи обычно контролируются аппаратно.

Обнаружить и устранить ошибки перезаписи можно только, если пользователь вводит избыточные данные, например, контрольные суммы. Если это не желательно, то следует по возможности проверять вводимые данные, хотя бы контролировать интервалы возможных значений, которые обычно определены в техническом задании, и выводить введенные данные для проверки пользователю.

Неверные данные обычно может обнаружить только пользователь.

Проверки промежуточных результатов позволяют снизить вероятность позднего проявления некоторых ошибок кодирования и проектирования. Для того чтобы такая проверка была возможной, необходимо, чтобы в программе использовались переменные, для которых существуют ограничения любого происхождения, например, связанные с сущностью моделируемых процессов.