**УВАЖАЕМЫЕ СТУДЕНТЫ! Законспектируйте в своей рабочей тетради по дисциплине приведенную лекцию (объемом 4-5 страницы), ответьте *письменно* на контрольные вопросы.**

**Результаты работы, фотоотчет, предоставить преподавателю на e-mail: r.bigangel@gmail.com до 23.01.2023.**

**При возникновении вопросов по приведенному материалу обращаться по следующему номеру телефона: (072)111-37-59, (Viber, WhatsApp), vk.com: https://vk.com/daykini**

***ВНИМАНИЕ!!! При отправке работы, не забывайте указывать ФИО студента, наименование дисциплины, дата проведения занятия (по расписанию).***

# Лекция 31

## Тема: Структурное и событийно-ориентированное программирование.

С момента появления первых ЭВМ возникла потребность написания большого количества программ, и эта потребность увеличивалась с каждым годом. Начали складываться методы и принципы создания программных продуктов, из которых постепенно сформировалась традиционная для 60-х – 70-х годов технология программирования «снизу-вверх», суть которой заключалась в следующем: сначала создавались программные модули нижнего уровня, из которых далее формировались модули более высоких уровней. На формирование этой технологии оказало влияние то, что в то время потребителем программы становился ограниченный круг лиц (часто сами разработчики), поэтому вопросы, связанные с дальнейшим сопровождением программы, не принимались во внимание при оценке качества программы. Зато основным критерием качества считалась её эффективность в смысле экономии ресурсов ЭВМ, поскольку тогда эти ресурсы были весьма ограничены. Программа должна была занимать минимум ОЗУ и выполняться за кратчайшее время. Тело программы было очень запутанным, и исправлять скрытые ошибки и вносить изменения в данные программы было очень трудно. При таком проектировании основные трудности концентрировались на заключительных этапах разработки больших проектов.

С развитием средств вычислительной техники ситуация кардинально изменилась: количество потребителей программ резко возросло, и узким местом стали не вычислительные, а человеческие ресурсы, необходимые при создании и сопровождении программ. При этом сопровождение программ стало стоить в несколько раз дороже. Неудовлетворенность традиционной технологией и осознание новых критериев заставило искать новые технологические принципы. Эти принципы были найдены и успешно внедрены в практику корпорацией IBM в начале 70-х годов. С тех пор начала прочно утверждаться технология структурного программирования.

**Цели структурного программирования:**

- избавиться от плохой структуры программы;

- создавать программы, которые можно было бы понимать, сопровождать и модифицировать без участия автора.

**Технология структурного программирования состоит из двух частей:**

- нисходящая разработка;
- структурное программирование.

**Нисходящая разработка.**

Нисходящую разработку можно рассматривать как процесс, состоящий из трех этапов:

- проектирование;
- планирование;
- реализация.

**Законы Мэрфи.**

Все оказывается сложнее, чем кажется.

Все тянется дольше, чем можно ожидать.

Все оказывается дороже, чем планировалось.

Если что-то может испортиться, оно обязательно испортится.

Комментарий Каллагена: «Мэрфи был оптимистом».

Эти законы хорошо описывают проблемы традиционной восходящей технологии программирования.

**Цели нисходящей разработки:**

- уменьшить сложность программ;
- уменьшить время разработки;
- позволить как можно раньше обнаружить ошибки.

**Проектирование программы**

Одним из принципов проектирования программ является модульность программ. Модуль – это программа, которую можно вызвать из любого другого модуля и можно отдельно откомпилировать.

Разбиение программ на модули характеризуется следующими преимуществами:

- сокращение сроков написания программы, так как проектирование модулей можно поручить разным программистам;
- можно создавать библиотеки наиболее часто употребляемых функций;
- упрощается загрузка больших программ в ОЗУ – не требуется сегментация всей программы, так как это достигается естественным образом при разбиении на модули;
- для руководства легче наблюдать за продвижением проекта;
- облегчается тестирование;
- проще проектирование и последующие изменения программы.

**Наряду с этим имеются и недостатки:**

- может увеличиться время выполнения программы;
- может возрасти размер памяти, требуемый программе;
- может увеличиться время компиляции и загрузки;

- при некорректном разбиении программ на модули могут возникнуть проблемы межмодульного взаимодействия (особенно при внесении изменений).

Для современной техники эти недостатки несущественны и окупаются за счет сокращения сроков разработки и сопровождения.

**Разбиение программы на модули нужно производить по возможности так, чтобы модули имели следующие желательные свойства:**

- модуль должен иметь один вход и один выход и возвращать управление тому, кто его вызвал;
- размер модуля должен быть небольшим (10-100 операндов), при этом его легче читать и тестировать;
- модуль не должен сохранять историю своих вызовов для управления своим функционированием;
- модуль должен по возможности реализовывать одну функцию преобразования исходных данных в результат, это позволяет выделять часто употребляемые модули и объединять их в библиотеки;
- по возможности принятие решений в модулях нужно организовать так, чтобы эти решения прямо влияли только на выполнение вызываемых модулей;
- модуль должен быть по возможности независимым от других модулей, для этого важно ослабить связи между модулями.

**Модули могут быть связаны:**

- по содержимому, если один модуль ссылается на операторы другого модуля, используя абсолютное смещение, при этом даже перекомпиляция другим компилятором может привести к ошибке;
- по общей памяти, если два модуля ссылаются на одинаковые абсолютные адреса, при этом изменение размера одного элемента общей области памяти может привести к ошибкам в других модулях. Глобальные данные затрудняют чтение программ, поэтому количество глобальных данных надо стремиться уменьшать;
- по управлению – когда один модуль при вызове другого модуля может модифицировать его действия (это может возникнуть, если вызываемый модуль реализует несколько функций, при этом, если исправлять действие одной из функций, можно испортить и выполнение остальных);
- по данным, нужно стремиться передавать меньше данных в модули.

Нисходящее проектирование программы основано на идее уровней абстракции.

Абстрагирование – это процесс обобщения, при котором внимание концентрируется на сходстве явлений и предметов и они объединяются в группы на основе этого сходства.

Уровни абстракции определяют уровни модулей в программе. На этапе проектирования строится схема иерархии, отображающая эти уровни, их функции и взаимодействие модулей разных уровней. От блок схемы она отличается тем, что не показывает логику принятия решений или точный порядок исполнения.

Разработка схемы иерархии.

Чтобы создать схему иерархии, то есть спроектировать модульную организацию системы, нужно начинать с вершины и продвигаться вниз.

**Пример.** Диалоговая информационная система, в которой обрабатываются запросы пользователя, приведена на рисунке 1.

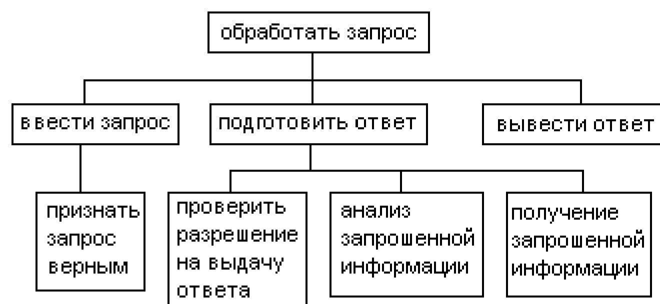


Рис. 1

Линии связи показывают подчиненность модулей. Каждый модуль активизируется вышестоящим и, закончив работу, возвращается ему управление, то есть передача управления происходит только по вертикальным связям. Таким образом, схемы иерархии строятся с использованием принципа вертикального управления, который предполагает следующие правила:

- модуль должен возвратить управление тому, кто его вызвал (исключение – аварийное завершение модуля);
- модуль может вызвать другой модуль уровнем ниже, но не может вызвать модуль своего уровня или выше (но может рекурсивно вызвать сам себя). Это упрощает межмодульный обмен данными. Если нужно вызвать модуль, находящийся несколькими уровнями ниже, то в этом случае в схеме иерархии его нужно дополнительно поместить на следующий уровень и везде специально отметить. Это означает, что модуль пишется один раз, а используется на разных уровнях иерархии;
- принятие основных решений нужно выносить на максимально высокий уровень (обычно в головной модуль, который является кратким конспектом всей программы);
- модуль низшего уровня не должен принимать решения за модули высшего уровня, то есть модуль не должен производить действий, непосредственно изменяющих порядок работы программы.

**Преимущества вертикального управления:**

- логика программы становится понятней;
- программу проще изменять и дополнять, в ней нет перекрестных связей;
- программирование и проверка вначале модулей высшего уровня, а затем низшего позволяет быстрее обнаруживать логические ошибки.

Модули низшего уровня детализируются только после написания всех модулей высших уровней. Рекомендации:

- при разбиении на модули лучше чуть перестараться, чем недостаточно продвинуться, так как объединять модули легче;
- нельзя рассредоточивать реализацию одной функции в разных модулях;
- аргументы функции следует передавать явно, а не через общую память, количество аргументов нужно стремиться уменьшать.

Процесс разделения на функции заканчивается, когда все модули полностью спроектированы и дальнейшее деление может привести к рассредоточению

функций. Задача разбиения на модули не имеет единственного решения. При разбиении программы на модули нужно провести внешнее проектирование каждого модуля, то есть определить его внешние характеристики. Эта информация выражается в виде внешних спецификаций модуля, которые содержат описания всех сведений, необходимых вызывающим его модулям. Здесь не описывается логика работы модуля. Внешние спецификации должны содержать следующие сведения:

- имя модуля;
- функция, выполняемая модулем;
- список параметров – число и порядок передачи модулю параметров;
- входные параметры – точное описание всех входных параметров (формат, размер, единицы измерений, диапазоны значений);
- выходные параметры – точное описание данных, которые возвращает модуль, здесь же указывается функциональная связь между входными и выходными параметрами, а также выходные данные, которые получаются при неверных входных данных;
- внешние эффекты – дается описание всех внешних событий, происходящих при работе модуля;
- использование внешних областей памяти и глобальных объектов.

Внешние спецификации определяют межмодульные связи. Их изменение приводит, как правило, к изменению вызывающих модулей. Лучше всего эти спецификации помещать в виде начального комментария в тексте модуля.

**Конец этапа проектирования характеризуется следующими условиями:**

- известно необходимое число модулей и их спецификации;
- связи между модулями отображены схемой иерархии;
- вся работа проверена пользователями на точность и полноту.

**Планирование.**

Планирование включает в себя две задачи: планирование порядка разработки модуля и планирование тестов. На этом этапе выбирается последовательность программирования и тестирования модулей.

Планирование порядка разработки модулей.

Существуют несколько подходов, но наиболее распространены иерархический и операционный подходы.

Иерархический подход. При этом подходе порядок программирования и тестирования модулей определяется их расположением в схеме иерархии. Сначала программируются и тестируются все модули одного уровня, после чего происходит переход на уровень ниже. При тестировании вызовы модулей нижних уровней заменяются заглушками. Заглушка – это упрощенная схема будущего модуля, содержащая все необходимое для тестирования модуля более высокого уровня и эмулирующая для модуля высокого уровня поведение модуля низкого уровня. Используя иерархический подход, не нужно забывать о двух вещах:

- необходимость использовать в модуле данные, подготовленные или измененные другим модулем, то есть зависимость по данным, может затруднить программирование и тестирование при иерархическом подходе;
- слепое следование поуровневому подходу приводит к реализации основной массы модулей в конце проекта.

Операционный подход. При этом подходе модули разрабатываются в порядке их выполнения при запуске готовой программы. Так как порядок исполнения обычно меняется с изменением входных данных, то здесь также остается свобода для выбора порядка разработки. Преимущество операционного подхода в том, что минимизируются трудности, вызванные зависимостью по данным. Недостаток операционного подхода в том, что при такой последовательности разработчики модулей «выстраиваются в очередь», то есть нерационально используется рабочая сила.

Таким образом, в чистом виде ни иерархический, ни операционный подходы не дают приемлемого результата. Наилучший подход – комбинированный, где взвешиваются достоинства каждого подхода. При этом, чтобы не выйти за рамки нисходящей технологии, перед программированием каждого модуля нужно проверять следующие условия:

- должен существовать путь управления к этому модулю, то есть такая цепочка уже разработанных модулей, через которую управление может быть передано новому модулю (критерий управления);
- должны быть доступны значения всех данных, необходимых для работы модуля, которые выдаются либо уже разработанными модулями, либо заглушками (критерий доступности данных).

Кроме этого, при определении порядка разработки модулей следует учитывать следующие рекомендации:

- обеспечить готовность вспомогательных модулей в первую очередь;
- следует начинать проектирование со сложных модулей, так как они могут оказаться сложнее и потребовать для разработки больше времени, кроме того, ранее программирование обнаружит ошибки в определении режимов работы модулей;
- обработка исключительных ситуаций, связанных с неправильными данными, должна программироваться в первую очередь.

На практике большинство этих принципов противоречат друг другу. Поэтому такое планирование требует времени и способностей. При поиске наилучшей последовательности лучше всего начинать с операционного подхода. При этом легко убедиться, что критерий управления и критерий доступности данных выполняются для каждого модуля. При этом часто нужно пытаться мысленно выполнить программу. Это может помочь выявить недостатки полученной модульной структуры и документации на модули (в документации должна быть указана зависимость модуля по данным).

Пример планирования для схемы на рисунке 1. Последовательность разрабатываемых модулей.

1. «Обработать запрос» – головной модуль.
2. «Вывести ответ» – необходим при тестировании.
3. «Подготовить ответ», «Анализ», «Получение информации» – как наиболее сложные.
4. «Ввести запрос», «Признать», «Проверить» – как наиболее простые.

**Планирование тестов.**

Вместе с планированием порядка, в котором следует разрабатывать модули, нужно планировать разработку тестов для всего проекта. Тесты необходимо готовить до начала программирования в силу следующих причин:

- устраняется возможность подгонки тестов к уже написанной программе (уменьшается возможность внесения одних и тех же ошибок в программу и тест);
- лучше понимаются ограничения на входные данные (сопряжение модулей зависит от передаваемых данных);
- при планировании и разработке тестов программист вынужден представить себе окончательный результат, получаемый программой, что позволяет на ранних этапах проектирования увидеть неточности спецификаций и вовремя принять решения;
- при таком подходе обеспечивается непрерывность создания программ: тесты определяют входные данные, которые используются при разработке модуля и при его проверке.

В некоторых случаях для тестирования создаются специальные программы – генераторы тестов.

Тесты не должен готовить тот человек, который пишет программу, чтобы исключить взаимовлияние этих процессов. Тестер должен быть специалистом высокой квалификации, знакомый с областью применения программы. Нужно привлекать к процессу тестирования пользователя программы: он может указать различные ситуации, возникающие при эксплуатации, и помочь при определении ограничений на данные.

**Реализация.**

Этап реализации включает:

- фактическую подготовку тестов;
- программирование и тестирование модулей.

Модули нужно добавлять по одному, а все возникающие несоответствия стараться обнаружить и исправить до перехода на следующий уровень.

**Структурное программирование.**

Общепринятого точного определения структурного программирования нет. Как концепция оно привлекло внимание после того, как Корадо Бойман и Джузеппе Якопини в 1966г. было доказано, что программу для решения любой логической задачи можно составить только из трех структур: следование, ветвление и повторение (цикл). Далее было указано, что неограниченное использование оператора goto нарушает соответствие между текстом и логикой работы программы (то есть программа плохо читается).

Можно так определить структурное программирование: это программирование, ориентированное на общение с людьми, а не с машиной. Для этого программа должна удовлетворять следующим требованиям:

- текст программы должен быть композицией трех основных элементов: следование, ветвление и цикл;

- употребление goto следует избегать всюду, где это возможно. Наихудшее применение goto – переход на оператор, расположенный выше в тексте программы;
- при написании программы желательно придерживаться следующих рекомендаций: используйте осмысленные имена для объектов; избегайте схожих имен; избегайте использования промежуточных переменных; во избежание неоднозначности употребляйте скобки в выражениях; по возможности избегайте специфических особенностей языка; не игнорируйте предупреждающие сообщения компилятора; игнорируйте все предложения по повышению эффективности, пока программа не будет правильной; используйте комментарии для записи спецификаций в начале модуля; комментируйте объявления переменных (для чего они предназначены); помните особенности внутреннего представления данных в ЭВМ и диапазоны значений типов переменных;
- текст программы нужно структурировать, то есть писать так, чтобы можно было легко выделять блоки программы;
- каждый модуль должен иметь один вход и один выход.

Защитное программирование (защита от дурака) – это контроль входных данных с целью обнаружения ошибок в них. Оно используется для следующих целей:

- устойчивость программы в целом;
- худшее, что может сделать модуль, – это получить неверные входные данные и выдать неверный, но правдоподобный результат.

Защитное программирование требует разумного подхода, так как, доведенное до крайности, оно может чрезвычайно усложнить программу.

**Тестирование программного обеспечения.**

Тестирование – это процесс выполнения программы с намерением найти ошибки. Если цель – показать отсутствие ошибок, то их будет найдено немного, если цель – показать наличие ошибок, то их будет найдено значительно больше. Отладка – это процесс установления точной природы уже найденной ошибки и ее исправление. Результат тестирования является исходными данными для отладки. Фундаментальные принципы тестирования отражены в следующих аксиомах тестирования.

1. Хорош тот тест, для которого высока вероятность обнаружить ошибку, а не тот, который демонстрирует правильную работу программы.
2. Одна из самых сложных проблем при тестировании – решить, когда надо закончить, то есть как выбрать минимум тестов, дающих максимальную отдачу.
3. Невозможно тестировать свою собственную программу. Тестирование должно быть разрушительным процессом.
4. Необходимая часть любого теста – описание ожидаемых выходных данных и результатов, то есть ожидаемые результаты тестирования нужно определять заранее, до написания программы.
5. Нужно избегать не воспроизводимых тестов, то есть не задавать в качестве входных воздействий заранее не спланированные значения, которые часто

невозможно повторить при нахождении ошибок. Тесты следует документировать.
6. Нужно готовить тесты как для правильных, так и для неправильных входных данных.
7. Нужно детально изучать результаты каждого теста.
8. По мере роста числа ошибок, обнаруженных в программе, растет относительная вероятность существования в ней необнаруженных ошибок.
9. Тестирование нужно поручать самым способным программистам.
10. Никогда нельзя изменять программу с целью облегчения ее тестирования.

Существует два похода к тестированию.

1. Предполагает составлять тесты только на основе внешних спецификаций.
2. Предполагает составлять тесты, изучая только логику программы.

**Проектирование тестов.**

1. Руководствуясь внешними спецификациями, нужно подготовить тест для каждой ситуации, вызывающей внешние эффекты, для каждой границы областей допустимых значений входных и выходных данных, для всех недопустимых условий. Если входных параметров немного и они имеют мало различных значений, имеет смысл перебрать все входные комбинации. Если входные аргументы ограничены некоторым диапазоном значений, то проверяют работу на границах области допустимых значений и в нескольких внутренних точках. Многие программы имеют функциональные границы. (Например, модуль выполняет сортировку чисел. Функциональные границы: массив пуст или содержит один элемент, массив уже отсортирован, все элементы массива – одинаковые).
2. Нужно проверить текст программы и добавить тесты с целью охвата всех условных переходов.
3. Для каждого цикла необходимо составить тесты соответствующие однократному выполнению тела цикла, максимальному числу итераций, обходу тела цикла.
4. По тесту программы необходимо выявить чувствительность к особым значениям входных данных и добавить тесты.