Методы, технологии и инструменты программирования

Технология ПО включает в себя такие понятия, как методы, инструменты, организационные мероприятия направленные на создания ПО.

Технология программирования – технологии разработки программ, которые будут использоваться людьми для решения задач на ЭВМ.

Метод программирования – способ, средство, определяющие стиль написания, откладки и сопровождения программ.

Основные определения

Программа — завершенный продукт, пригодный для запуска своим автором на системе, на которой он был разработан.

Программный продукт —по ГОСТ 7.83 2001 самостоятельное, отчуждаемое произведение представляющее собой публикацию текста программы или программ на языке программирования или в виде исполняемого кода. Такая программа должна быть написана в обобщенном стиле, тщательно оттестирована и сопровождена подробной документацией

Программный комплекс — набор взаимодействующих программ, согласованных по функциям и форматам, точно определенным интерфейсам, и вкупе составляющих полное средство для решения больших задач.

Жизненный цикл программного обеспечения – это весь период его разработки и эксплуатации, начиная с момента возникновения замысла и заканчивая прекращением ее использования.

Методология программирования – совокупность методов, применимых в жизненном цикле программного обеспечения и объединенных общим философским подходом.

В настоящее время широко известны следующие методологии программирования – императивный, объектно-ориентированный, логический, функциональный, быстрого проектирования.

Технология программирования изучает технологические процессы и порядок их прохождения – стадии (с использованием знаний, методов и средств).

Процесс — совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные. Процессы состоят из набора действий, а каждое действие из набора задач. Вертикальное измерение отражает статические аспекты процессов и оперирует такими понятиями, как рабочие процессы, действия, задачи,


ссы
ализ
Проектирование
Программирование
Тестирование
Сопровож
Ста
Проектирование
Программирование
Тестирование
и отладка
Эксплуат
и сопров

Стадия — часть действий по созданию программного обеспечения, ограниченная некоторыми временными рамками и заканчивающаяся выпуском конкретного продукта, определяемого заданными для данной стадии требованиями. Стадии состоят из этапов, которые обычно имеют итерационный характер. Иногда стадии объединяют в более крупные временные рамки, называемые фазами. Итак, горизонтальное измерение представляет время, отражает динамические аспекты процессов и оперирует такими понятиями, как фазы, стадии, этапы, итерации и контрольные точки.

Технологический подход определяется спецификой комбинации стадий и процессов, ориентированной на разные классы программного обеспечения и на особенности коллектива разработчиков.

ПАРАДИГМА

- Парадигма – это система взглядов на явления окружающего мира и представлений о возможных взаимодействиях с ними
- Парадигма программирования – система идей и понятий, определяющих фундаментальный стиль программирования

ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

- императивная
- процедурная
- объектно-ориентированная
- декларативная
  - логическая
  - функциональная
- параллельная
- …

Некоторый язык программирования не обязательно использует только одну парадигму, многие языки поддерживают несколько парадигм, являясь мультипарадигменными

Ни одна парадигма не может быть одинаково эффективной для всех задач, и программисту следует выбирать лучший стиль программирования для решения каждой отдельной задачи

Синтаксис языка описывает систему правил написания различных языковых конструкций, а **семантика языка** программирования определяет смысл этих конструкций. Синтаксис языка программирования может быть описан с помощью НБФ-нотаций.

Семантика языка взаимосвязана с используемой вычислительной моделью. В настоящее время языки программирования в зависимости от применяемой вычислительной модели делятся на четыре основные группы:

Классификация языков программирования

Императивное программирование.

Императивное программирование — это исторически первая методология программирования, которой пользовался каждый программист, программирующий на любом из «массовых» языков программирования – Basic, Pascal, C.

Методология императивного программирования характеризуется принципом последовательного изменения состояния вычислителя пошаговым образом. При этом управление изменениями полностью определено и полностью контролируемо.

Методы и концепции.

Метод *изменения состояний* — заключается в последовательном изменении состояний. Метод поддерживается концепцией алгоритма.

Метод *управления потоком исполнения* — заключается в пошаговом контроле управления. Метод поддерживается концепцией потока исполнения.

Вычислительная модель. Если под вычислителем понимать современный компьютер, то его состоянием будут значения всех ячеек памяти, состояние процессора (в том числе — указатель текущей команды) и всех сопряженных устройств. Единственная структура данных — последовательность ячеек (пар «адрес» - «значение») с линейно упорядоченными адресами.

В качестве математической модели императивное программирование

Основные черты императивных языков:

- использование именованных переменных
- использование оператора присваивания
- использование составных выражений;
- использование подпрограмм
- и др.

Синтаксис и семантика. Языки, поддерживающие данную вычислительную модель, являются как бы средством описания функции переходов между состояниями вычислителя. Основным их синтаксическим понятием является оператор. Первая группа — простые операторы, у которых никакая их часть не является самостоятельным оператором (например, оператор присваивания, оператор безусловного перехода, вызова процедуры и т. п.). Вторая группа — структурные операторы, объединяющие другие операторы в новый, более крупный оператор (например, составной оператор, операторы выбора, цикла и т. п.).

Традиционное *средство структурирования* — подпрограмма (процедура или функция). Подпрограммы имеют параметры и локальные определения и могут быть вызваны рекурсивно. Функции возвращают значения как результат своей работы.

Если в данной методологии требуется решить некоторую задачу для того, чтобы использовать ее результаты при решении следующей задачи, то типичный подход будет таким. Сначала исполняется алгоритм, решающий первую задачу. Результаты его работы сохраняются в специальном месте памяти, которое известно следующему алгоритму, и используются им.

Класс задач. Императивное программирование наиболее пригодно для решения задач, в которых последовательное исполнение каких-либо команд является естественным. Примером здесь может служить управление современными аппаратными средствами. Поскольку практически все современные компьютеры императивны, эта методология позволяет порождать достаточно эффективный исполняемый код. С ростом сложности задачи императивные программы становятся все менее и менее читаемыми.

Программирование и отладка действительно больших программ (например, компиляторов), написанных исключительно на основе методологии императивного программирования, может затянуться на долгие годы.

Структурное программирование

Структурное программирование — методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков. Предложена в 70-х годах XX века Э. Дейкстрой, разработана и дополнена Н. Виртом.

Цель структурного программирования - повышение надежности программ, обеспечение *сопровождения и модификации,* облегчение и ускорение разработки.

Методология структурного императивного программирования — подход, заключающийся в задании хорошей топологии императивных программ, в том числе отказе от использования глобальных данных и оператора безусловного перехода, разработке модулей с сильной связностью и обеспечении их независимости от других модулей.

Подход базируется на двух основных принципах:

- Последовательная декомпозиция алгоритма решения задачи сверху вниз.
- Использование структурного кодирования.

Методы и концепции, лежащие в основе структурного программирования.

- Метод *алгоритмической декомпозиции сверху вниз* — заключается в пошаговой детализации постановки задачи, начиная с наиболее общей задачи. Данный метод обеспечивает хорошую структурированность. Метод поддерживается концепцией алгоритма.
- Метод *модульной организации частей программы* — заключается в разбиении программы на специальные компоненты, называемые модулями. Метод поддерживается концепцией модуля.
- Метод *структурного кодирования* — заключается в использовании при кодировании трех основных управляющих конструкций. Метки и оператор безусловного перехода являются трудно отслеживаемыми связями, без которых мы хотим обойтись. Метод поддерживается концепцией управления

Теорема о структурировании (Бёма-Джакопини (Boem-Jacopini)): Всякую правильную программу (т.е. программу с одним входом и одним выходом без зацикливаний и недостижимых веток) можно записать с использованием следующих логических структур - последовательность, выбора и повторение цикла

- Следствие 1: *Всякую программу можно привести к форме без оператора goto.*
- Следствие 2: *Любой алгоритм можно реализовать в языке, основанном на трех управляющих конструкциях - последовательность, цикл, повторение.*
- Следствие 3: *Сложность структурированных программ ограничена, даже в случае их неограниченного размера.*

Принципы структурного программирования

- Программирование осуществляется «сверху-вниз»;
- Программа состоит из подпрограмм с одним входом и одним выходом;
- Подпрограмма должно допускать только три основные структуры – последовательное выполнение, ветвление и

Модульное программирование

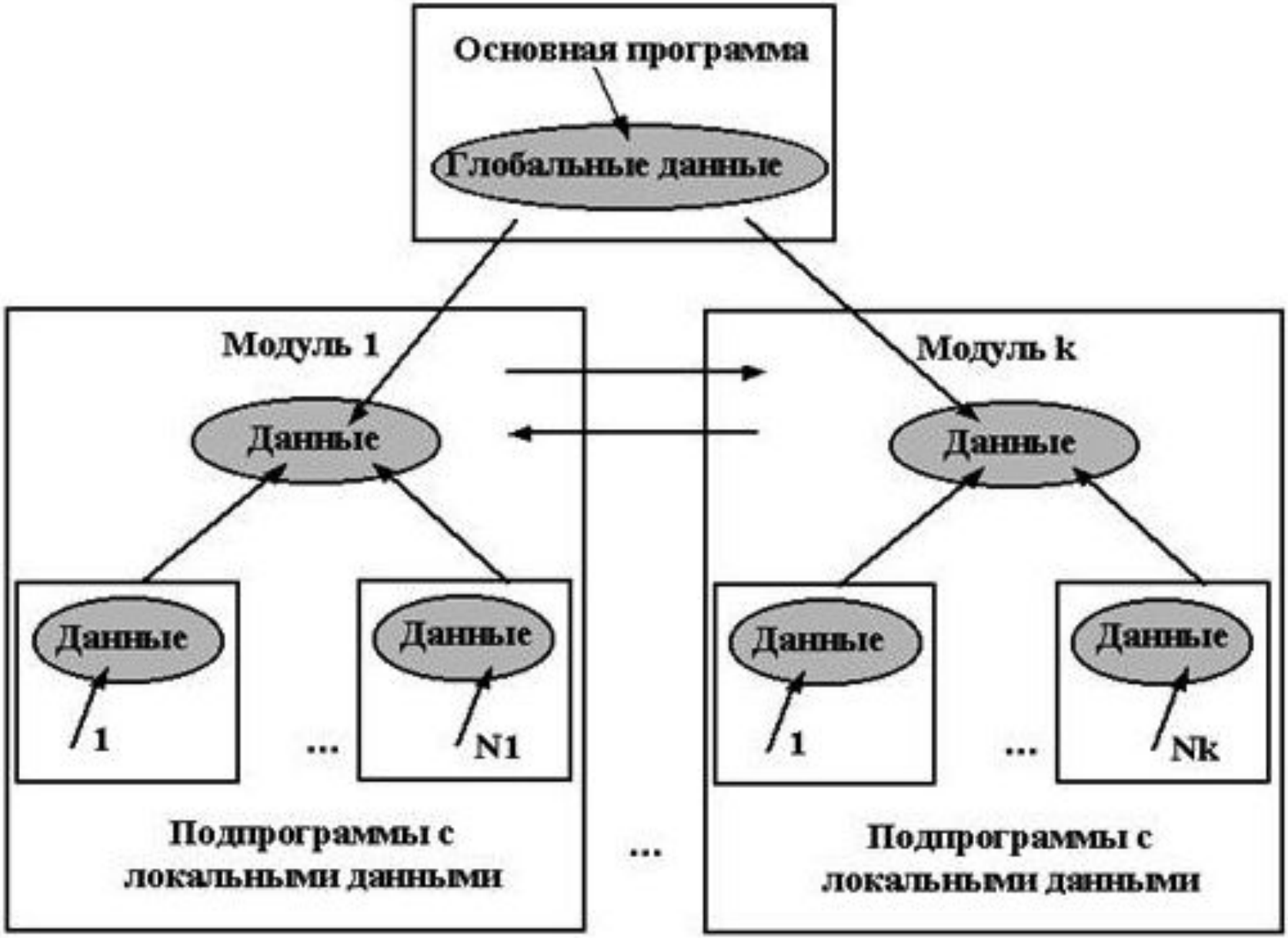
Модульное программирование – это метод разработки программ, при котором программа разбивается на независимые *модули,* причем каждый из них имеет свой контролируемый размер, четкое назначение и детально проработанный интерфейс с внешней средой.

Модуль — это совокупность команд, к которым можно обратиться по имени.

Модуль — это совокупность операторов программы, имеющая граничные элементы и идентификатор.

Функциональная спецификация модуля должна включать:

- синтаксическую спецификацию его входов, которая должна позволять построить на используемом языке программирования синтаксически правильное обращение к нему;
- описание семантики функций, выполняемых модулем по каждому из его входов.



Модули с локальными данными и подпрограммами

Концепции модульного программирования.

- *Принцип утаивания информации Парнаса.* Всякий компонент утаивает единственное проектное решение, т. е. модуль служит для утаивания информации. Подход к разработке программ заключается в том, что сначала формируется список проектных решений, которые особенно трудно принять или которые, скорее всего, будут меняться. Затем определяются отдельные модули, каждый из которых реализует одно из указанных решений.

- *Аксиома модульности Коуэна.* Модуль — независимая программная единица, служащая для выполнения некоторой определенной функции программы и для связи с остальной частью программы. Программная единица должна удовлетворять следующим условиям:

✔ блочность организации, т. е. возможность вызвать программную единицу из блоков любой степени вложенности;

✔ синтаксическая обособленность, т. е. выделение модуля в тексте синтаксическими элементами;

✔ семантическая независимость, т. е. независимость от места, где программная единица вызвана;

✔ общность данных, т. е. наличие собственных данных, сохраняющихся при каждом обращении;

✔ полнота определения, т. е. самостоятельность программной единицы

- *Сборочное программирование* Цейтина. Модули — это программные кирпичи, из которых строится программа. Существуют три основные предпосылки к модульному программированию:

✔ стремление к выделению независимой единицы программного знания. В идеальном случае всякая идея (алгоритм) должна быть оформлена в виде модуля;

✔ потребность организационного расчленения крупных разработок;

✔ возможность параллельного исполнения модулей (в контексте параллельного программирования).

Разновидности модулей.

1) *"Маленькие" (функциональные)* модули, реализующие, как правило, одну какую-либо определенную функцию. Основным и простейшим модулем практически во всех языках программирования является процедура или функция.

2) *"Средние" (информационные)* модули, реализующие, как правило, несколько операций или функций над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Примеры "средних" модулей в языках программирования:

a)задачи в языке программирования Ada;

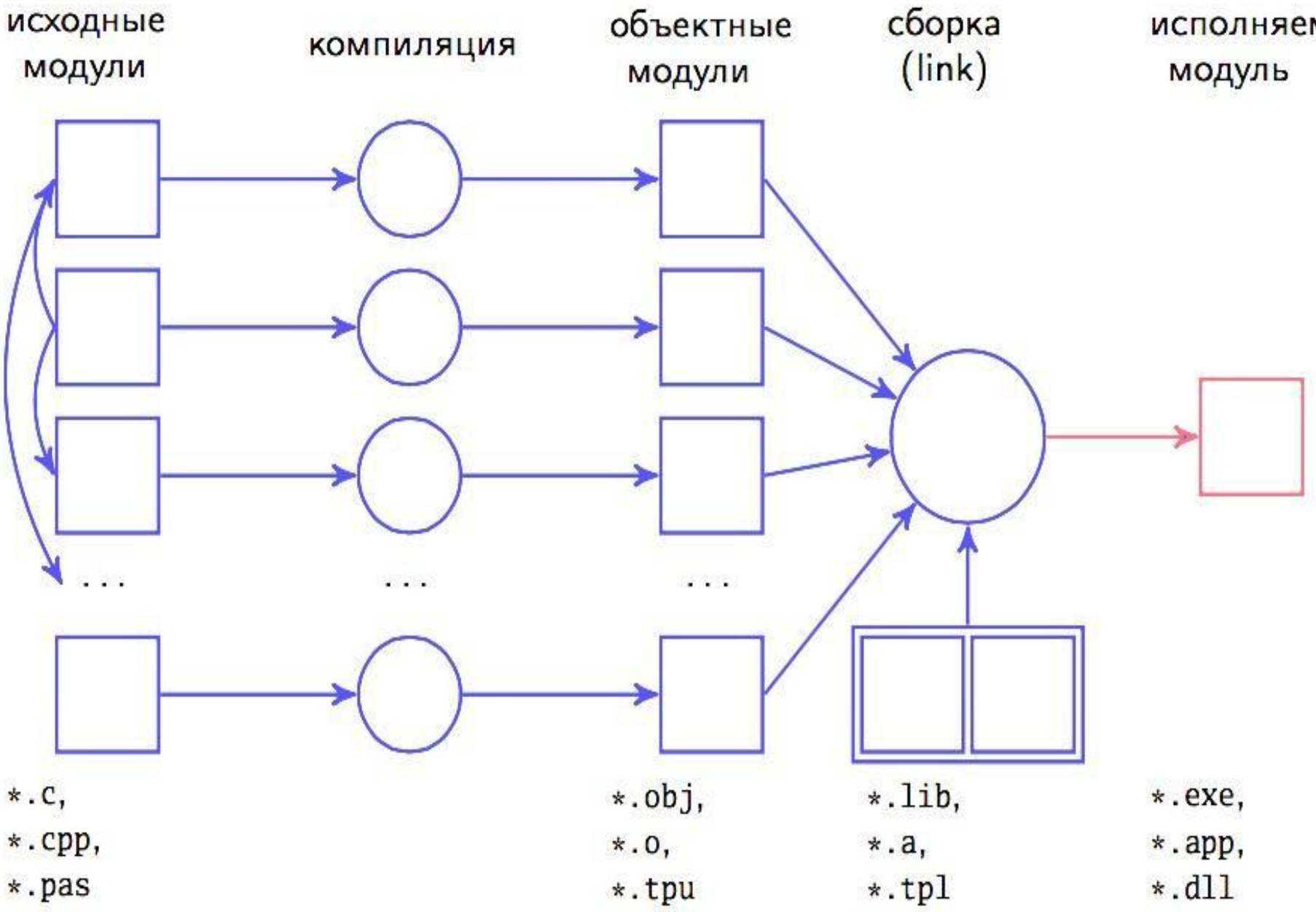
b)кластер в языке программирования CLU;

c)классы в языках программирования C++ и Java.

3) *"Большие" (логические) модули*, объединяющие набор "средних" или "маленьких" модулей. Примеры "больших" модулей в языках программирования: a)модуль в языке программирования Modula-2;

b)пакеты в языках программирования Ada и Java.

Различаются три основных вида модулей, соответствующие основным этапам получения готовой программы: исходный, объектный, загрузочный. ***Исходный модуль*** **содержит команды выбранного языка программирования. В дальнейшем он подлежит компиляции. В результате компиляции получаем объектный модуль.** ***Объектный модуль*** **состоит из текста на машинном языке и содержит, кроме машинных команд и констант, управляющие словари, необходимые для последующей загрузки и настройки программы в оперативной памяти. Объектные модули имеют стандартный формат независимо от исходного языка программирования. Специальной программой – редактором – они объединяются в загрузочный модуль.** ***Загрузочный модуль*** **– программа, состоящая из текста на машинном языке, и управляющих словарей, которые необходимы для его загрузки в оперативную память для последующего исполнения. Компиляторы, редакторы и**


исходные модули
компиляция
объектные модули
сборка (link)
исполняемый модуль
. . .
. . .
. . .
*.c,
*.cpp,
*.pas
*.obj,
*.o,
*.tpu
*.lib,
*.a,
*.tpl
*.exe,
*.app,
*.dll

Исходный модуль включает, как правило, указатель начала модуля, тело модуля, указатель конца модуля. *Указатель начала модуля* содержит, по меньшей мере, имя модуля, по которому к нему обращаются. *Тело модуля* состоит из любых допустимых команд языка программирования, за исключением указателей начала и конца модуля. *Указатель конца модуля* ограничивает текст модуля. Иногда он может отсутствовать. Тогда тело модуля ограничивается последней командой, либо указателем начала следующего модуля.
Со временем, когда основные требования структурного подхода стали поддерживаться языками программирования, и под модулем стали понимать отдельно компилируемую библиотеку ресурсов, требование независимости модулей стало основным.

Метод проектирования, который можно называть "модульным", должен удовлетворять пяти основным требованиям:

- **Декомпозиции (decomposability).**
- **Композиции (composability).**
- **Понятности (understandability).**
- **Непрерывности (continuity).**
- **Защищенности (protection).**

Метод проектирования удовлетворяет критерию Декомпозиции, если он помогает разложить задачу на несколько менее сложных подзадач, объединяемых простой структурой, и настолько независимых, что в дальнейшем можно отдельно продолжить работу над каждой из них.

Метод удовлетворяет критерию Модульной Композиции, если он обеспечивает разработку элементов программного продукта, свободно объединяемых между собой для получения новых систем, быть может, в среде, отличающейся от той, для которой эти элементы первоначально разрабатывались.

Метод удовлетворяет критерию Модульной Понятности, если он помогает получить такую программу, читая которую можно понять содержание каждого модуля, не зная текста остальных, или, в худшем случае, ознакомившись лишь с некоторыми из них.

Метод удовлетворяет критерию Модульной Непрерывности, если незначительное изменение спецификаций разработанной системы приведет к изменению одного или небольшого числа модулей.

Метод удовлетворяет критерию Модульной Защищенности, если он

Из рассмотренных критериев следуют пять правил, которые должны соблюдаться, чтобы обеспечить модульность:

- **Прямое отображение (Direct Mapping).**
- **Минимум интерфейсов (Few Interfaces).**
- **Слабая связность интерфейсов (Small interfaces - weak coupling).**
- **Явные интерфейсы (Explicit Interfaces).**
- **Скрытие информации (инкапсуляция) (Information Hiding).**

Модульная структура, создаваемая в процессе конструирования ПО, должна оставаться совместимой с модульной структурой, создаваемой в процессе моделирования проблемной области.

Каждый модуль должен поддерживать связь с возможно меньшим числом других модулей.

Если два модуля общаются между собой, то они должны обмениваться как можно меньшим объемом информации.

Всякое общение двух модулей A и B между собой должно быть очевидным и отражаться в тексте A и/или B.

Разработчик каждого модуля должен выбрать некоторое подмножество свойств модуля в качестве официальной информации о модуле,

Характеристики модуля (Майерс, 1980)

- Размер модуля
- Связность (прочность) модуля
- Сцепление модуля с другими модулями
- Рутинность (независимость от предыдущих обращений) модуля

1) размера модуля;

В модуле должно быть 7 (+/-2) конструкций (например, операторов для функций или функций для пакета).

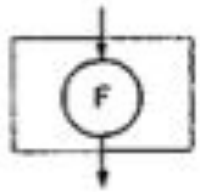
Модуль (функция) не должен превышать 60 строк. В результате его можно поместить на одну страницу распечатки или легко просмотреть на экране монитора.

2) прочности (связности) модуля;

Связность (прочность) модуля (cohesion) — мера независимости его частей. Чем выше связность модуля — тем лучше, тем больше связей по отношению к оставшейся части программы он упрятывает в себе.

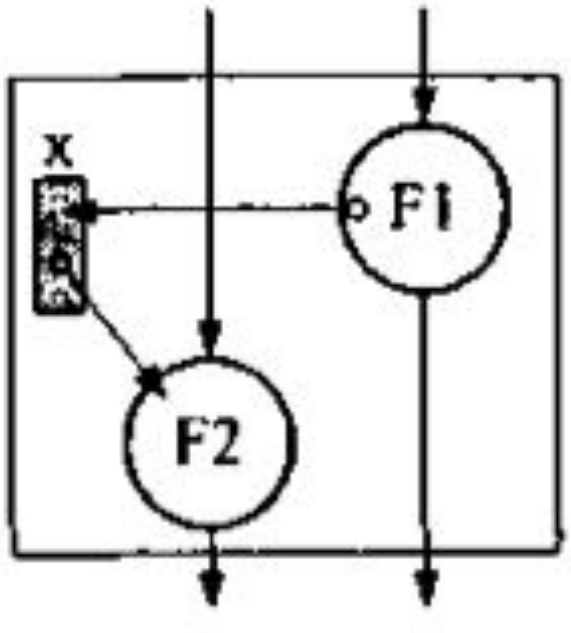
Типы связности:

- *Функциональная* связность. Модуль с функциональной связностью реализует одну какую-либо определенную функцию и не может быть разбит на 2 модуля с те типами связностей.



- *Последовательная* связность. Модуль с такой связностью может быть разбит на последовательные части, выполняющие независимые функции, но совместно реализующие единственную функцию. Например, один и тот же модуль может быть использован сначала для оценки, а затем для обработки данных.

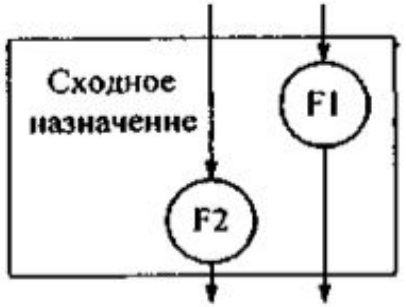
- *Информационная* (коммуникативная) связность. Модуль с информационной связностью — это модуль, который выполняет несколько операций или функций над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Эта информационная связность применяется для реализации абстрактных типов данных.



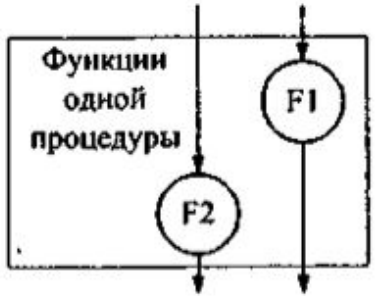
Следует избегать

- **Временной связности**
  - когда объединяются действия, связанные со временем (например, действия, которые должны быть выполнены в один и тот же момент времени)

- **Логической связности**
  - когда в модуль объединяются действия по признаку их некоторого подобия (например, функции для проверки корректности входных данных для всей программы)



- **Случайной связности**
  - когда действия объединяются произвольным образом
- **Процедурной связности**
  - когда действия сгруппированы вместе только потому, что они выполняются в течение одной и той же части процесса



3) сцепления модуля с другими модулями;

Сцепление **(coupling)** — мера относительной независимости модуля от других модулей. Независимые модули могут быть модифицированы без переделки других модулей. Чем слабее сцепление модуля, тем лучше. Рассмотрим различные типы сцепления.

Независимые **модули** — это идеальный случай. Модули ничего не знают друг о друге. Организовать взаимодействие таких модулей можно, зная их интерфейс и соответствующим образом перенаправив выходные данные одного модуля на вход другого. Достичь такого сцепления сложно, да и не нужно, поскольку сцепление по данным (параметрическое сцепление) является достаточно хорошим.

Сцепление *по данным* (параметрическое) — это сцепление, когда данные передаются модулю, как значения его параметров, либо как результат его обращения к другому модулю для вычисления некоторой функции. Этот вид сцепления реализуется в языках программирования при обращении к функциям (процедурам). Две разновидности этого сцепления определяются характером данным.

- Сцепление по простым элементам данных.

Не рекомендуется использовать

- *Сцепление по управлению* – это сцепление, в котором один модуль управляет решениями внутри другого с помощью передачи флагов, переключателей и т.п.
  В этом случае один модуль должен достаточно хорошо знать структуру вызывающего модуля
- *Сцепление по внешним ссылкам* – возникает, когда у одного модуля есть доступ к данным другого
- *Сцепление по кодам* – возникает, когда коды инструкций модулей перемежаются друг с другом (внутренняя область одного модуля доступна другому)

4) рутинности (идемпотентность, независимость от предыдущих обращений) модуля.

Рутинность — это независимость модуля от предыдущих обращений к нему (от предыстории). Будем называть модуль рутинным, если результат его работы зависит только от количества переданных параметров (а не от количества обращений).

Модуль должен быть рутинным в большинстве случаев, но есть и случаи, когда модуль должен сохранять историю. В выборе степени рутинности модуля пользуются тремя рекомендациями.

- В большинстве случаев делаем модуль рутинным, т. е. независимым от предыдущих обращений.
- Зависящие от предыстории модули следует использовать только в тех случаях, когда это необходимо для сцепления по данным.
- В спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость, чтобы

Свойства модуля

- *На модуль можно ссылаться с помощью имени модуля.*
- *Модуль должен иметь один вход и один выход*
- *Модуль должен быть сравнительно невелик*
- *Возможность сепаратной компиляции*
- *Модуль может вызвать другой модуль или сам себя*
- *Модуль должен возвращать управление тому, кто его вызвал*
- *Модуль не должен сохранять историю*

Преимущества модульного программирования

- ***Функциональные компоненты модульной программы могут быть написаны и отлажены порознь***
- ***Модульную программу проще проектировать, легче сопровождать и модифицировать***
- ***Становится проще процедура загрузки в оперативную память боль-шой программы, требующей сегментации***

Недостатки модульного программирования

- ***Может увеличиться время компиляции и загрузки.***
- ***Может увеличиться время исполнения программы.***
- ***Может возрасти объем требуемой памяти.***
- ***Организация межмодульного взаимодействия может оказаться довольно сложной.***

Стандартные модули

- Разработка и использование стандартных библиотечных программ является одним из путей построения модульного программирования
- Преимущества стандартных модулей:
  1) стандартные модули экономят время программирования;
- 2) они также могут экономить память компьютера и выполняться максимально быстро;

  3) использование стандартных модулей защищает от ошибок программирования.

- Недостатки:
  - Нужный стандартный модуль иногда бывает трудно найти. Еще труднее – подробную документацию к нему
  - Стандартный модуль может оказаться более универсальным, чем это нужно пользователю
  - Стандартный модуль может быть написан на другом языке

Каждый программист решает самостоятельно использовать ему стандартные модули или разрабатывать свой собственный.

Подпрограммы (функции)

- Подпрограммы также являются средством для построения модульных программ
- Не всякая подпрограмма является модулем. Модуль должен удовлетворять перечисленным выше характеристикам и свойствам.

Библиотеки ресурсов.

Различают библиотеки ресурсов двух типов:

библиотеки подпрограмм и библиотеки классов. Библиотеки подпрограмм реализуют функции, близкие по назначению, например, библиотека графического вывода информации. 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

Методы разработки структуры программы

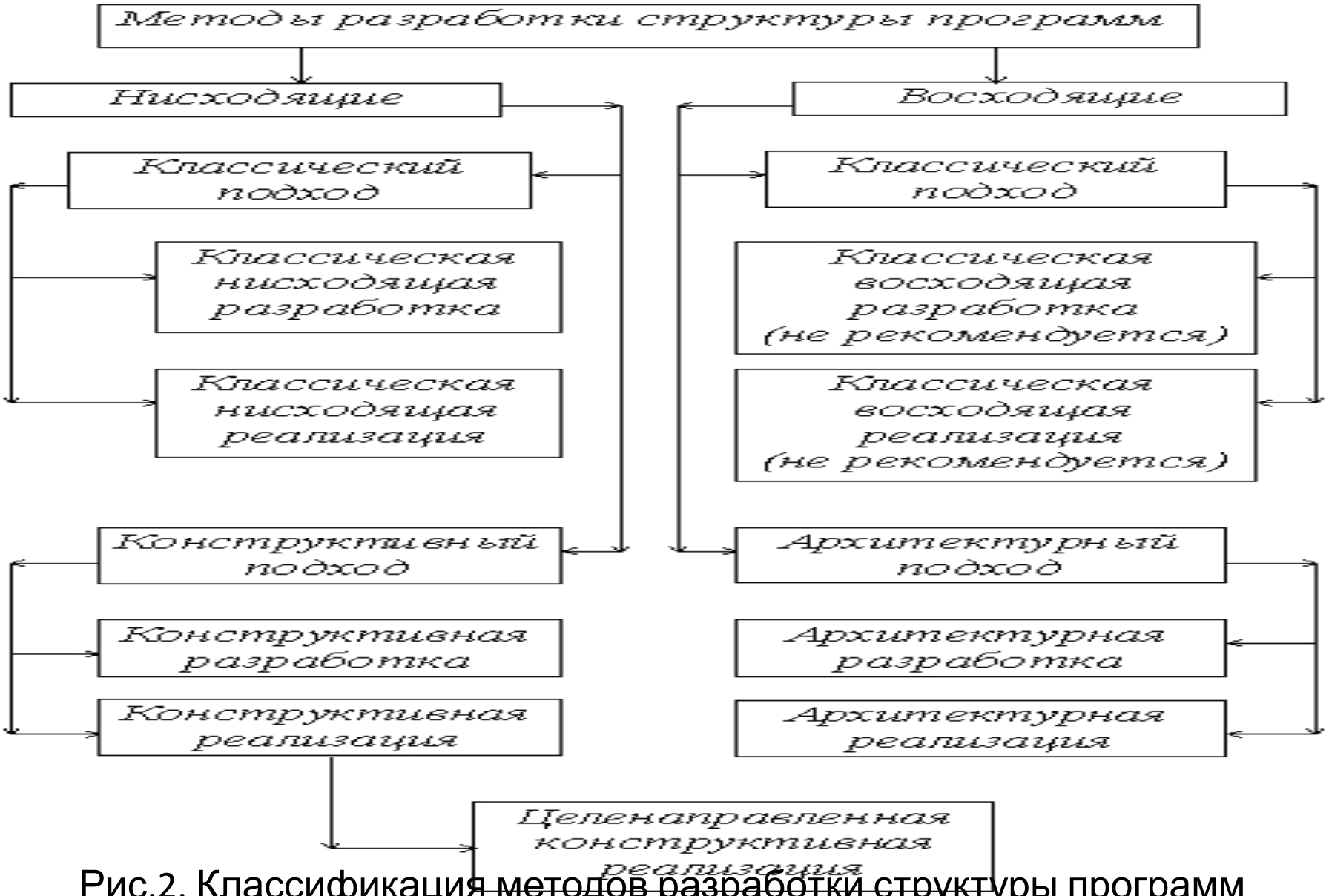


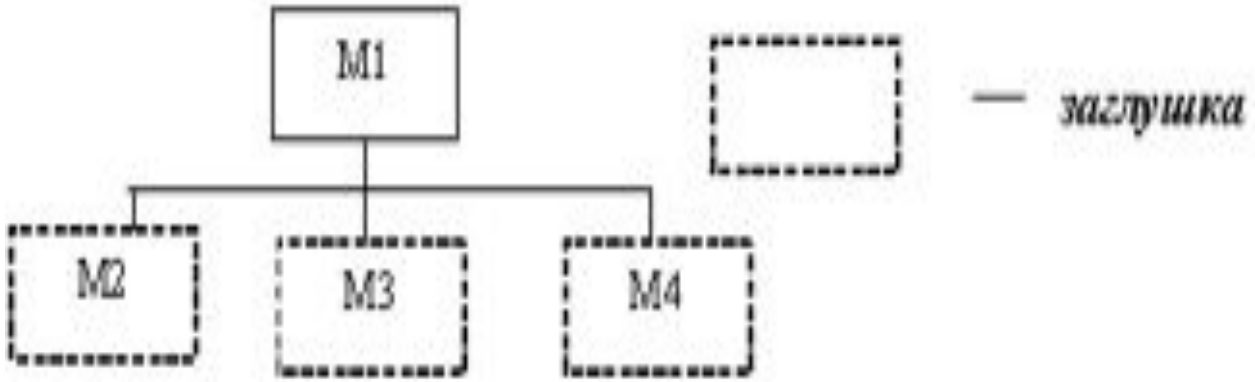
Рис.2. Классификация методов разработки структуры программ

Нисходящее и восходящее программирование.

При разработке модульных программ применяются два метода проектирования – нисходящее и восходящее.

При **нисходящем проектировании** разработка программного комплекса идет сверху вниз.

На первом этапе разработки кодируется, тестируется и отлаживается головной модуль, который отвечает за логику работы всего программного комплекса. Остальные модули заменяются заглушками, имитирующими работу этих модулей. Применение заглушек необходимо для того, чтобы на самом раннем этапе проектирования можно было проверить работоспособность головного модуля. На последних этапах проектирования все заглушки постепенно заменяются рабочими модулями.


M1
M2
M3
M4
— заглушка

При **восходящем проектировании** разработка идет снизу вверх. На первом этапе разрабатываются модули самого низкого уровня . На следующем этапе к ним подключаются модули более высокого уровня и проверяется их работоспособность. На завершающем этапе проектирования разрабатывается головной модуль, отвечающий за логику работы всего программного комплекса. 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.

Методы нисходящего и восходящего программирования имеют свои преимущества и недостатки.

Недостатки нисходящего проектирования:

- Необходимость заглушек.
- До самого последнего этапа проектирования неясен размер программного комплекса и его эксплутационные характеристики, за которые, как правило, отвечают модули самого низкого уровня.

Преимущество нисходящего проектирования – на самом начальном этапе проектирования отлаживается головной модуль (логика программы).

Преимущество восходящего программирования – не нужно писать заглушки.

Недостаток восходящего программирования – головной модуль разрабатывается на завершающем этапе проектирования, что порой приводит к необходимости дорабатывать модули более низких уровней.

На практике применяются оба метода. Метод нисходящего проектирования чаще всего применяется при разработке нового программного комплекса, а метод восходящего проектирования – при

Архитектурный подход к разработке программы представляет собой модификацию восходящей разработки, при которой модульная структура программы формируется в процессе программирования модуля. Но при этом ставится существенно другая цель разработки: повышение уровня используемого языка программирования, а не разработка конкретной программы.

Метод объектно-ориентированного программирования

Методология ООП использует **метод объектной декомпозиции**, согласно которому структура системы (статическая составляющая) описывается в терминах *объектов и связей* между ними, а поведение системы (динамическая составляющая) - в терминах обмена сообщениями между объектами. Сообщения могут быть как реакцией на события, вызываемые как внешними факторами, так и порождаемые самими объектами.

Объектно-ориентированные программы называют «программами, управляемыми от событий», в отличие от традиционных программ, называемых «программам, управляемыми от данных».

Основные **методы и концепции ООП**

Метод объектно-ориентированной декомпозиции – заключается в выделении объектов и связей между ними. Метод поддерживается концепциями инкапсуляции, наследования и полиморфизма.

Метод абстрактных типов данных – метод, лежащий в основе инкапсуляции. Поддерживается концепцией абстрактных типов данных.

Метод пересылки сообщений – заключается в описании поведения

Вычислительная модель чистого ООП поддерживает только одну операцию – *посылку сообщения объекту.* Сообщения могут иметь параметры, являющиеся объектами. Само сообщение тоже является объектом.

Объект имеет набор обработчиков сообщений (набор методов). У объекта есть поля – персональные переменные данного объекта, значениями которых являются ссылки на другие объекты. В одном из полей объекта хранится ссылка на объект-предок, которому переадресуются все сообщения, не обработанные данным объектом. Структуры, описывающие обработку и переадресацию сообщений, обычно выделяются в отдельный объект, называемый классом данного объекта. Сам объект называется экземпляром указанного класса.

Синтаксис и семантика

В синтаксисе чистых объектно-ориентированных языков все может быть записано в форме посылки сообщений объектам. Класс в объектно-ориентированных языках описывает структуру и функционирование множества объектов с подобными характеристиками, атрибутами и поведением. Объект принадлежит к некоторому классу и обладает своим собственным внутренним состоянием. Методы — функциональные свойства, которые можно активизировать.

У каждого объекта есть ссылка на класс, к которому он относится. При приеме сообщения объект обращается к классу для обработки данного сообщения. Сообщение может быть передано вверх по иерархии наследования, если сам класс не располагает методом для его обработки. Если обработчик событий для сообщения выбирается динамически, то методы, реализующие обработчиков событий, принято называть виртуальными.

Естественным средством структурирования в данной методологии являются классы. Классы определяют, какие поля и методы экземпляра доступны извне, как обрабатывать отдельные сообщения и т. п. В чистых объектно-ориентированных языках извне доступны только методы, а доступ к данным объекта возможен только через его методы.

Взаимодействие задач в данной методологии осуществляется при помощи обмена сообщениями между объектами, реализующими данные задачи.

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

Декларативное программирование

- Программа на декларативном языке состоит из двух компонент: условия задачи (которую иногда называют «базой данных») и целевого запроса
- Для декларативного программирования необходимо наличие «решателя» (называемого обычно интерпретатором), который «знает», как выполнить целевой запрос, исходя из условий, представленных в «базе данных»

Декларативные языкеи программирования:

- Логическое
- функционвльное

ОБЛАСТИ ПРИМЕНЕНИЯ ДЕКЛАРАТИВНЫХ ЯЗЫКОВ

- Реализация обработки типов данных, имеющих рекурсивную природу: списков, деревьев, графов и сводящихся к ним структур
- Такого рода задачи характерны для обработки символьной информации, то есть для создания трансляторов и решения задач искусственного интеллекта: обработки естественного языка, трансформации и автоматического синтеза программ, аналитического преобразования формальных текстов и др.
- Создание систем искусственного интеллекта
- Разработка экспертных систем и оболочек экспертных систем
- Создание систем помощи принятия решений
- Разработка систем обработки естественного языка
- Построение планов действий роботов
- …

Логическое программирование

Согласно логическому подходу к программированию, программа представляет собой совокупность правил или логических высказываний. Кроме того, в программе допустимы *логические причинно-следственные связи*, в частности, на основе операции импликации.

Таким образом, языки логического программирования базируются на классической логике и применимы для систем логического вывода, в частности, для так называемых экспертных систем. На языках логического программирования естественно формализуется логика поведения, и они применимы для описаний правил принятия решений, например, в системах, ориентированных на поддержку бизнеса.

Важным преимуществом такого подхода является достаточно высокий уровень машинной независимости, а также возможность откатов – возвращения к предыдущей подцели при отрицательном результате анализа одного из вариантов в процессе поиска решения (скажем, очередного хода при игре в шахматы), что избавляет от необходимости поиска решения путем полного перебора вариантов и увеличивает эффективность реализации.

Недостатком логического подхода в концептуальном плане является специфичность класса решаемых задач.

Другой недостаток практического характера состоит в сложности эффективной реализации для принятия решений в реальном времени, скажем, для систем жизнеобеспечения.

Нелинейность структуры программы является особенностью декларативного подхода и представляет собой оригинальную особенность, а не объективный недостаток.

В качестве примеров языков логического программирования можно привести Prolog (название возникло от слов PROgramming in LOGic) и Mercury

Функциональное программирование

Функциональный стиль программирования основан на аргументной постановке задачи абстракции математической функции и аналитическом методе построения программ подобно математическим преобразованиям.

Функциональный стиль программирования характеризуется тем, что результат в нем выражается в терминах функций, применяемых к другим функциям, то есть функций, передающих результаты другим функциям.

Аппликация - операция применения функции к аргументу. Этот стиль программирования не использует оператор присваивания. Основными средствами функционального программирования являются композиция и рекурсия.

Важнейшей характеристикой функционального подхода - всякая программа, разработанная на языке функционального программирования, может рассматриваться как функция, аргументы которой, возможно, также являются функциями.

Функциональный подход породил целое семейство языков, родоначальником которых, как уже отмечалось, стал язык программирования LISP. Позднее, в 70-х годах, был разработан первоначальный вариант языка ML, который впоследствии развился, в частности, в SML , язык Haskell, а также ряд других языков.

Важным преимуществом реализации языков функционального программирования является автоматизированное динамическое распределение памяти компьютера для хранения данных. При этом программист избавляется от необходимости контролировать данные, а если потребуется, может запустить функцию **«сборки мусора»** – очистки памяти от тех данных, которые больше не понадобятся программе.

Сложные программы при функциональном подходе строятся посредством агрегирования функций. При этом текст программы представляет собой функцию, некоторые аргументы которой можно также рассматривать как функции. Таким образом, повторное использование кода сводится к вызову ранее описанной функции, структура которой, в

Функцией в языке программирования называется конструкция этого языка, описывающая правила преобразования аргумента (так называемого фактического параметра) в результат.

Для формализации понятия «функция» была построена математическая теория, известная под названием ламбда-исчисления

Под **конверсией** понимается преобразование объектов исчисления (а в программировании – функций и данных) из одной формы в другую. Исходной задачей в математике было стремление к упрощению формы выражений. В программировании именно эта задача не является столь существенной использование ламбда-исчисления как исходной формализации может способствовать упрощению вида программы, т.е. вести к оптимизации программного кода. конверсии обеспечивают переход к вновь введенным обозначениям и, таким образом, позволяют представлять предметную область в более компактном либо более

Близость к математической формализации и изначальная функциональная ориентированность послужили причиной следующих преимуществ *функционального подхода:*

- *простота тестирования и верификации программного кода* на основе возможности построения строгого математического доказательства корректности программ;
- *унификация представления программы и данных* (данные могут быть инкапсулированы в программу как аргументы функций, означивание или вычисление значения функции может производиться по мере необходимости);
- *безопасная типизация:* недопустимые операции с данными исключены;
- *динамическая типизация:* возможно обнаружение ошибок типизации во время выполнения (отсутствие этого свойства в ранних языках функционального программирования может приводить к переполнению оперативной памяти компьютера);
- *независимость программной реализации от машинного представления данных и системной архитектуры программы* (программист сосредоточен на деталях реализации, а не на особенностях машинного представления данных).

К достоинствам относятся:

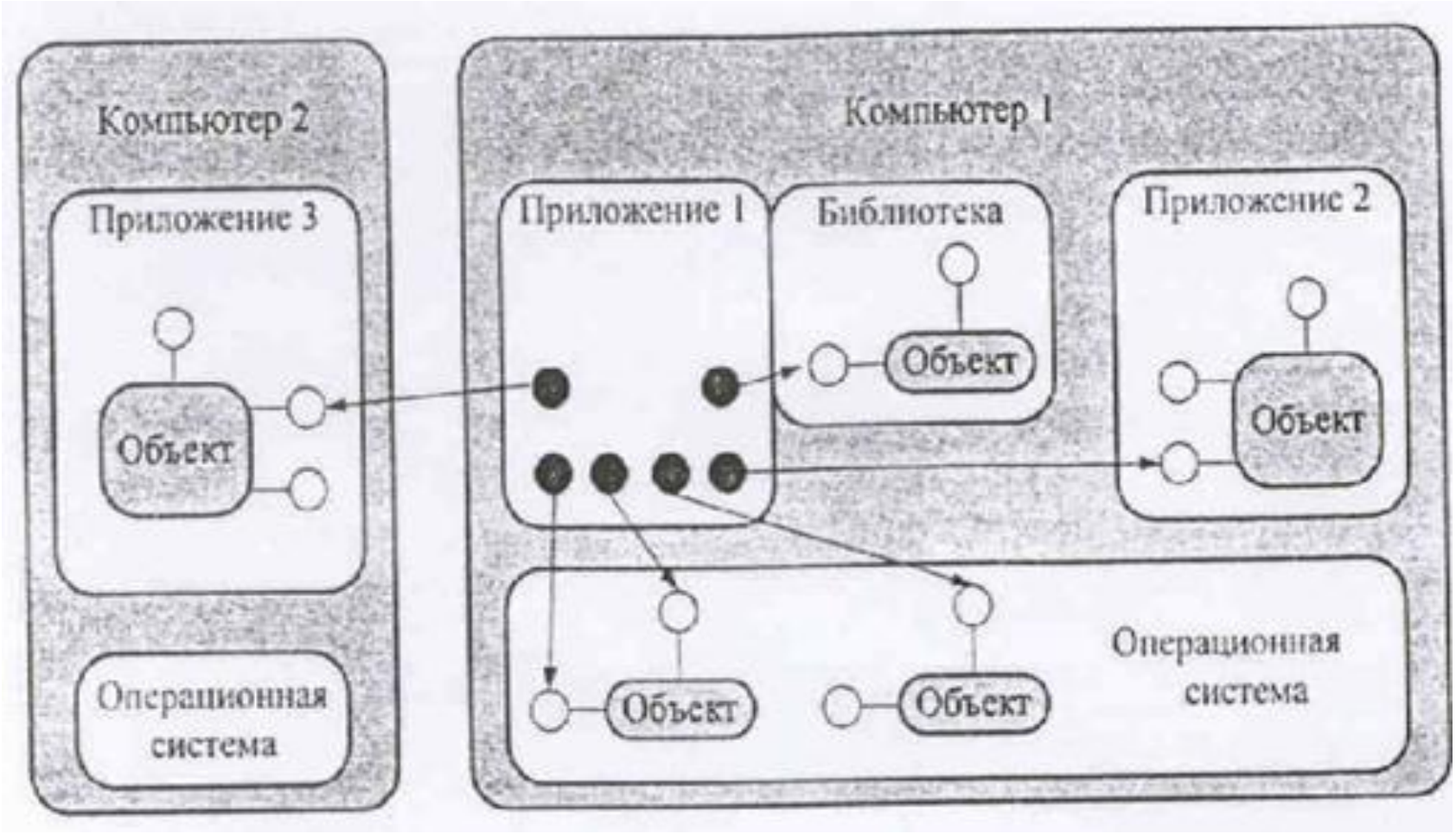
- безопасность программного кода, т.е. гарантия отсутствия переполнения памяти (в случае корректно написанной программы) и, соответственно, защиты от потенциальной неустойчивости работы системы посредством искусственного создания переполнения (такие языки программирования как «классический» C и C++ потенциально небезопасны);
- *статическая типизация*: все ошибки несоответствия типов выявляются уже на стадии контроля соответствия типов в ходе трансляции (а не во время выполнения программы, как в LISP и Scheme);
- *выводимость типов* (нет необходимости явно указывать тип каждого выражения, при этом результирующий программный код становится более удобочитаемым, его легче хранить и повторно

Компонентный подход и CASE - технологии

Компонентный подход (с середины 90-х годов XX до нашего времени) предполагает построение программного обеспечения из отдельных компонентов физически отдельно существующих частей программного обеспечения, которые взаимодействуют между собой через *стандартизованные двоичные интерфейсы*. В отличие от обычных объектов объекты-компоненты можно собрать в динамически вызываемые библиотеки или исполняемые файлы, распространять в двоичном виде (без исходных текстов) и использовать в любом языке программирования, поддерживающем соответствующую технологию.

Компонентный подход лежит в основе технологий, разработанных на базе COM (Component Object Model – компонентная модель объектов), и технологии создания распределенных приложений CORBA (Common Object Request Broker Architecture – общая архитектура с посредником обработки запросов объектов). Эти технологии используют сходные принципы и различаются лишь особенностями их реализации.

Технология COM фирмы Microsoft является развитием технологии OLE (Object Linking and Embedding – связывание и внедрение объектов), которая используется в Windows. Технология COM определяет *общую парадигму взаимодействия программ любых типов*: библиотек, приложений, операционной системы. Т.е. позволяет одной части программного обеспечения использовать функции (службы), предоставляемые другой, независимо от того, функционируют ли эти части в пределах одного процесса, в разных процессах на одном компьютере или на разных компьютерах (рис. 6). Модификация COM, обеспечивающая передачу вызовов между компьютерами, называется DCOM (Distributed COM – распределенная COM).



Взаимодействие программных компонентов различных типов