Технология программирования

Курс лекций для гр. 3057, 4057

Лекция №6

Содержание

Структуризация моделей программ

Проекты сложных ПП обычно требуют описания несколькими видами моделей, взаимно дополняющими друг друга; таков подход UML

- Многомодельность вообще характерна для описания сложных систем
- Однако смешение понятийных средств разных видов в одной модели недопустимо

Основной принцип методологии проектирования ПП - структуризация моделей, т.е. стандартизация структур, ограничение количества и разнообразия частей и связей - все это для улучшения ясности, понятности проекта и продукта

Все это меры борьбы со сложностью

Одна из целей структуризации – *обозримость* модели
– требует ограничения количества составных частей в модели «психологической константой» 7±2 → возможно несколько уровней абстракции

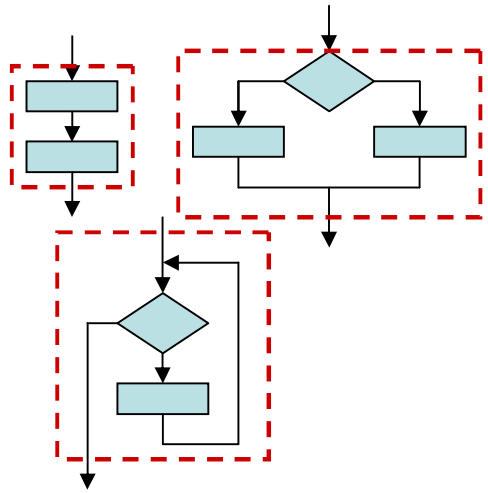
- ✓ Модель любого уровня должна содержать не более десятка составных частей (графическая емкость страницы А4)
- ✓ Каждая из частей может детализироваться на нижележащем уровне абстракции – нисходящая декомпозиция

Принцип иерархического поуровневого описания и принятия решений - общий принцип не только инженерной и научной деятельности, но и хорошего мышления вообще

Структуризация алгоритмических моделей

- это *структурное программирование* в узком смысле; иначе оно называется «программированием без go to»

Начало - статья Дейкстры "GO TO considered to be harmful" (1965):
частые ошибки кодирования - неправильная передача управления - вызваны скачками по тексту программы



Разрешены только три вида элементарных управляющих структур с функциональными блоками (ФБ):

- ✓ Последовательность (составной оператор): S1; S2;
- ✓ Выбор (разветвление): if P then S1 else S2;
- ✓ Повторение (цикл): while P do S;

Каждая из этих стандартных структур имеет, как и ФБ, один вход - один выход
Следовательно, возможны эквивалентные преобразования для структуризации:

- ❖ ФБ → станд. структура (детализация, декомпозиция)
- ❖ станд. структура → ФБ (обобщение, абстрагирование)

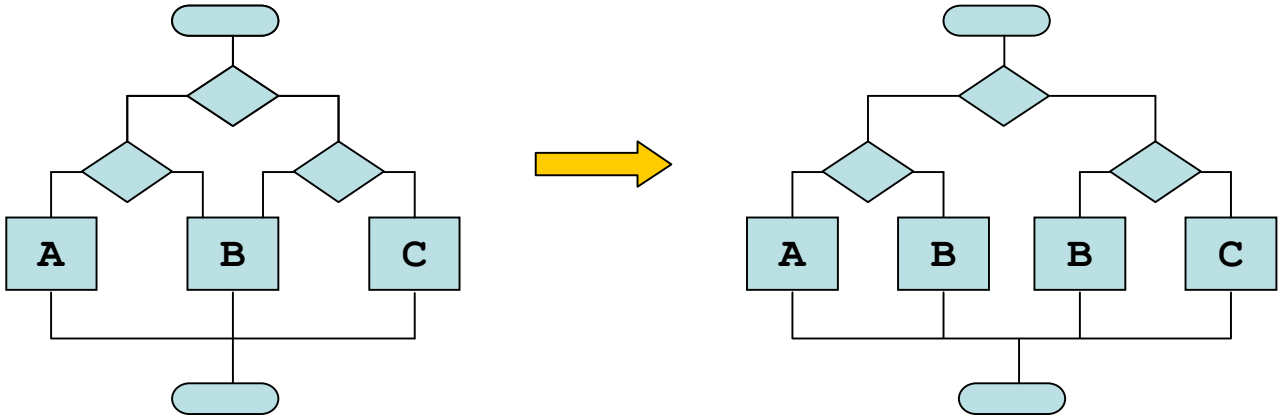
} **обозримые модели на каждом уровне**

Теоретическое обоснование возможности записи *любого* алгоритма с помощью только указанных трех структур - теорема Бома-Якопини (1966)

Структуризация алгоритмических моделей (2)

Запрет на go to часто приводит к потере эффективности

Например, следующее преобразование неструктурной блок-схемы в структурную путем дублирования блока:



Дублирование кода фрагмента B неэффективно по памяти, оформление B как подпрограммы - неэффективно по времени

Существует универсальный метод Ашкрофта-Манна (1971) автоматического преобразования *любой* блок-схемы в структурную (правда, неэффективную и не наглядную)

Программу изначально следует проектировать как структурную !

Структуризация алгоритмических моделей (3)

- Языки программирования делятся на структурные – содержащие конструкции трех стандартных структур управления, и неструктурные
- Все современные языки – структурные (в них список стандартных структур расширен добавлением операторов **case, for и repeat**); некоторые из них вообще не содержат go to (Modula-2, CLU, Java)
- Ассемблеры и первоначальные Фортран и Бейсик – неструктурные
- Чтобы избежать неэффективности, современные языки содержат “структурные” операторы перехода: не на метку, а в точку входа в объемлющий блок (**break**) или точку вызова подпрограммы (**return**)
- Похожим образом оператор возбуждения исключения (в механизме exception в Аде, Java или C++) описывает переход не на метку, а по имени исключительной ситуации – на код ее обработки
- В целом современный подход не рекомендует, но и не запрещает **go to**

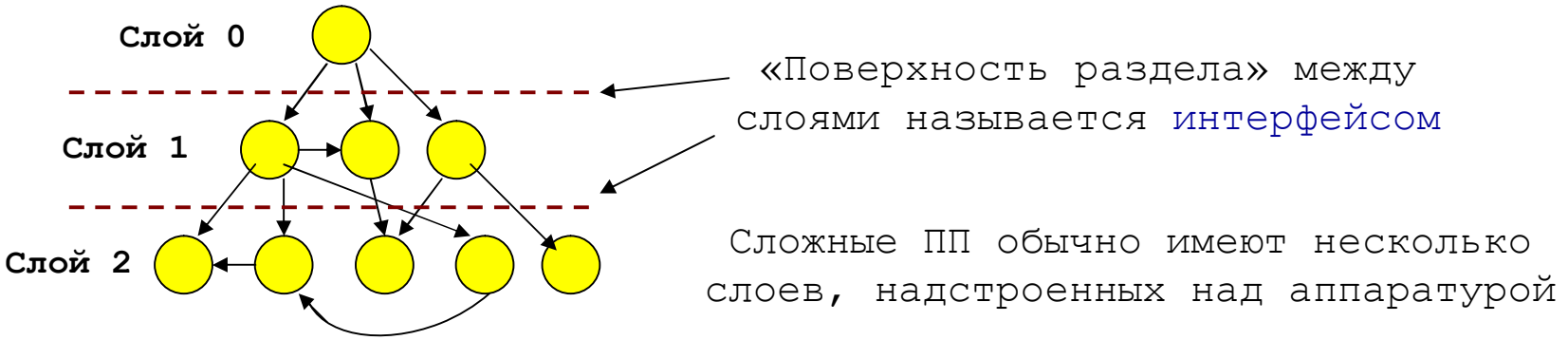
Структуризация функциональной модульной структуры

Ограничивается разнообразие связей модулей (подпрограмм, функций, классов) по вызовам, т.е. стандартизуются виды графа вызовов модулей

А) Строгая иерархия: граф вызовов – дерево

- **+** обозримость, понятность кода
- **+** облегчается отладка, т.к. единственная цепочка вызовов для любого модуля
- **–** необходимо дублирование модулей на нижних уровнях иерархии для выполнения одинаковых функций

В) Поуровневая (layered) структуризация: граф – гамак (расширение дерева перекрестными дугами между ярусами) или "слоеный пирог":



Отступление: понятие интерфейса

Интерфейс (И.) – совокупность средств и методов взаимодействия объектов

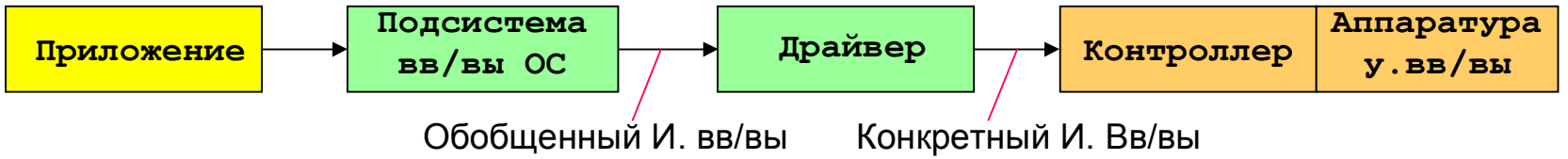
- Взаимодействие – обмен информацией и управление
- Обычно И. несимметричен: одна его сторона – управляющая (заказчик, master, client), другая – подчиненная (поставщик, slave, server)
- Средства взаимодействия – в зависимости от природы взаимодействующих объектов
- Для 3-х видов объектов: аппаратура, программы, пользователи – имеем 4 вида интерфейса:

1. Аппаратный И. – между цифровыми устройствами. Обеспечивает двусторонний обмен сигналами на физическом уровне. Средства: кабели, шины, разъемы, схемы, сигналы. Методы: временны**е** диаграммы сигналов, алгоритмы их обработки. Примеры:

- И. вв/вы – между центральными и внешними устройствами компьютера: RS-232, USB, Unibus, CAMAC и др.
- И. системной шины PCI – между ЦП и быстрыми устройствами типа HD (для PC)

2. Аппаратно-программный И. – между программой и аппаратурой:

- Система команд и прерываний компьютера
- Взаимодействие приложения с внешним устройством при вв/вы:



Понятие интерфейса (2)

3. Программный И. – между программными компонентами

- И. модуля, процедуры, класса, метода, библиотеки, пакета
- В случае программной библиотеки: CALL-интерфейс; современное название – API (Application Programming Interface) - это набор форматов вызова процедур или публичных методов класса (прототипов функций)
- В современных языках И. – это тип; он может передаваться как параметр и наследоваться
- Два значения термина И.:
  - ✓ точное описание (спецификация) И.
  - ✓ его программная реализация (код)

4. И. пользователя (UI) – между пользователем и программой

Диалоговое взаимодействие с помощью средств вв-вы:

- клавиатуры – И. командной строки
- мыши и графических элементов на экране – графический И. (GUI)

Принципы хорошего интерфейса

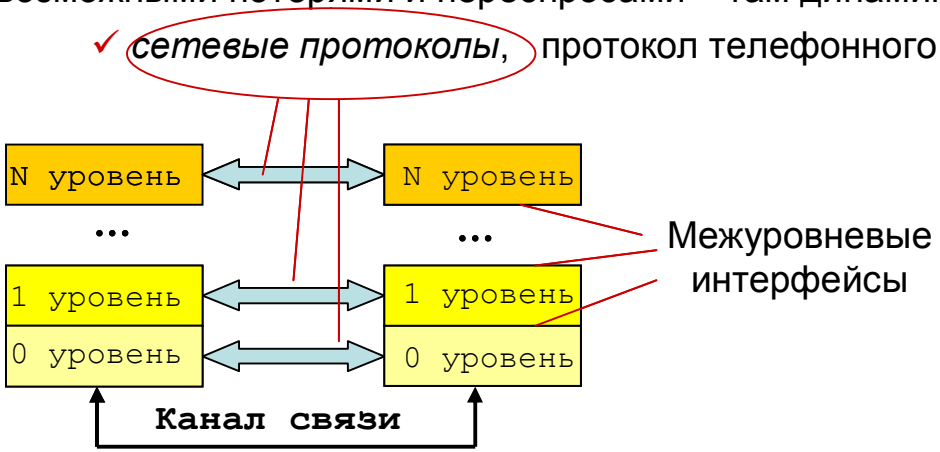
- *Инкапсуляция* внутреннего устройства, изоляция сторон И.
  - Сервер не должен знать, *зачем* он задействуется, клиент – *как* тот действует
  - Основа независимости реализации поставщика при фиксированном интерфейсе
- Экономность, лаконичность И. – минимум обмениваемой информации
  - Все это борьба со сложностью !
- Стандартизация, унификация И. – способствует совместимости, сопрягаемости (interoperability) и повторному использованию компонентов
- Описание И. должно исчерпывающим образом определять функциональность компонента, но не предопределять способ ее реализации
  - Спецификация интерфейса может служить его стандартом

Интерфейс vs. протокол

Протокол – это интерфейс, рассматриваемый как процесс

- И. – спецификация взаимодействия в статике: *что* и *где*
- Протокол – спецификация динамики: *когда* и *как*
  - ✓ т.е., описание последовательности шагов взаимодействия

- В API передачу фактических параметров обычно считают одномоментной
- Если не считать ее одномоментной и описывать то, как заполняется стек параметров при вызове функций – это будет протокол
- Интерфейс аутентификации пользователя – несколько шагов → протокол
- В сетях связи канал обычно требует долгого обмена последовательностями сигналов с возможными потерями и переспросами – там динамика существенна
  - ✓ *сетевые протоколы*, протокол телефонного соединения, факс-протокол, …

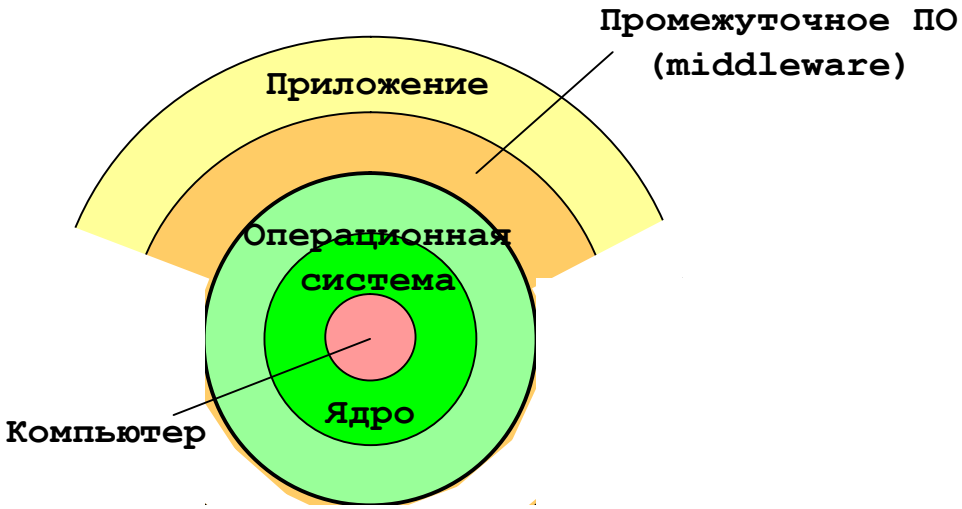


В распределенных системах протоколы могут описывать многостороннее взаимодействие (напр., передачу широковещательных сообщений), а не только двустороннее, как интерфейс

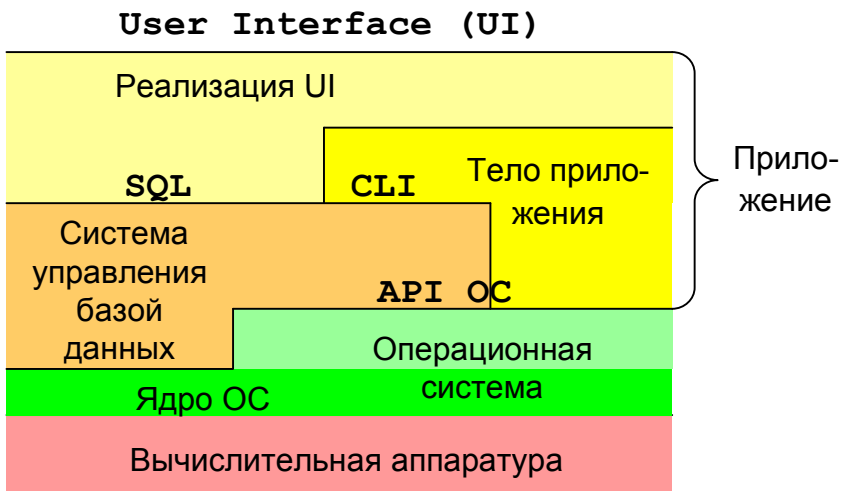
Многоуровневая структура протоколов сети

Поуровневая структуризация

(продолжение темы слайда 7)



Вложенные виртуальные машины



CLI – Call Level Interface = API для СУБД
SQL – язык запросов к базе данных

Цели поуровневой структуризации

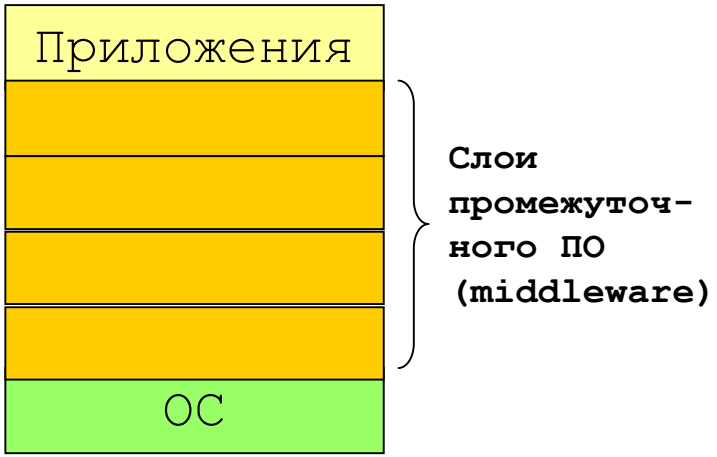
- Функциональная декомпозиция - способ борьбы со сложностью
- Независимость реализации слоя при фиксированном его интерфейсе, что способствует:
  - распараллеливанию работ при разработке ПП
  - независимой модификации реализации слоя
  - совместимости, сопрягаемости и переносимости слоев

➢ С целью переносимости ПП машинно-зависимые части кода выделяют в специальный (нижний) уровень.

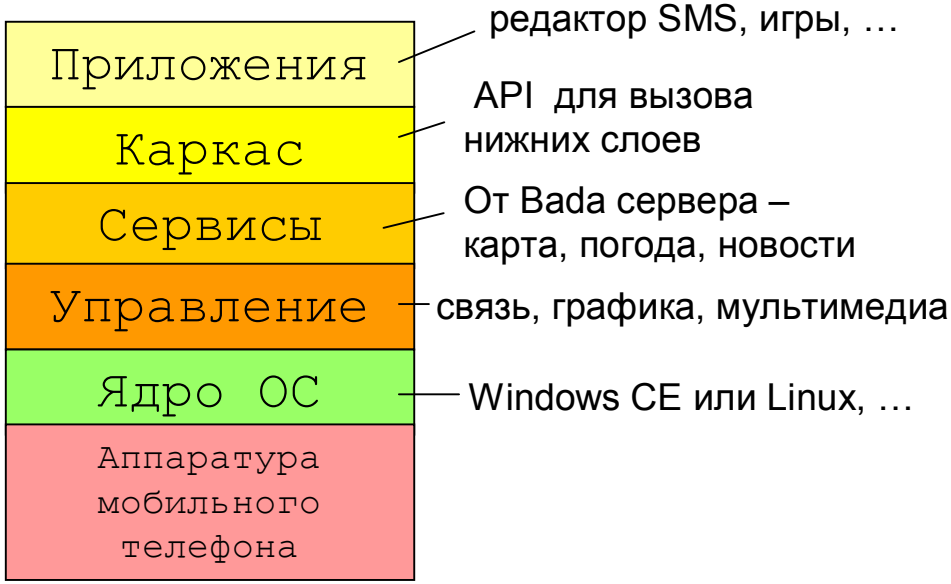
✓ Только 10% кода Unix написано на ассемблере конкретной машины, остальное – на Си

➢ Строгая спецификация интерфейса компонентов – основа их стандартизации

Промежуточное ПО



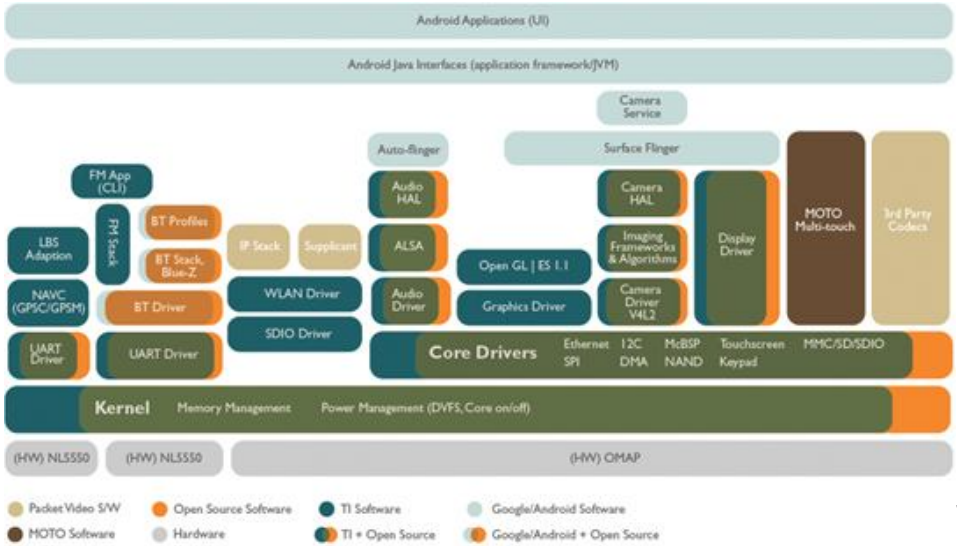
Пример: платформа Samsung Bada

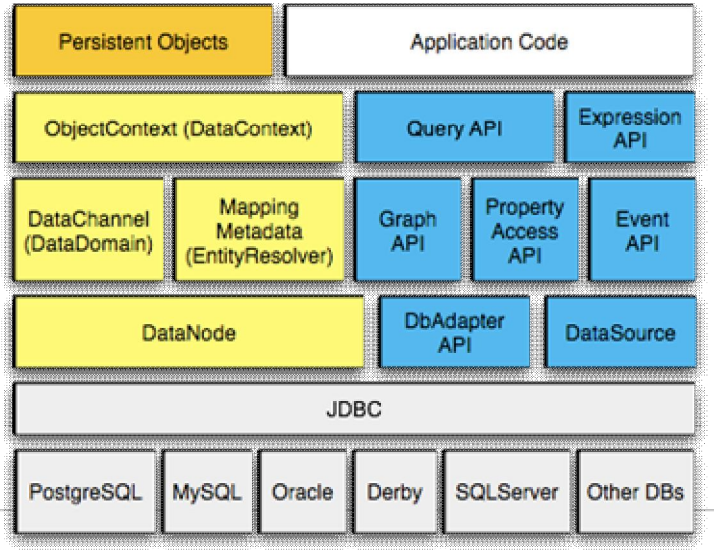


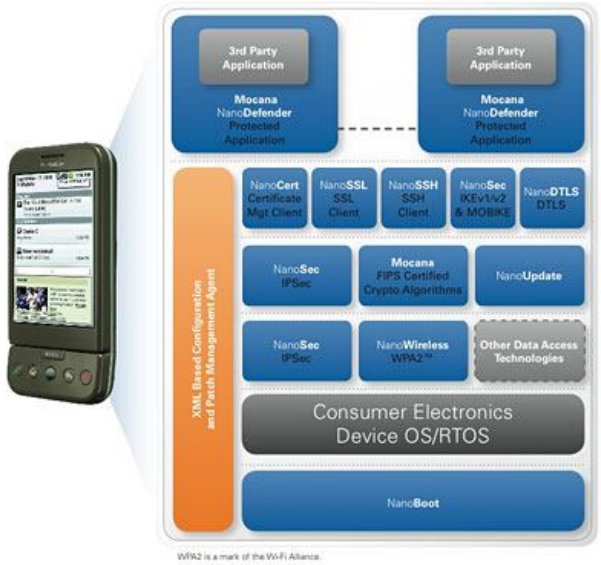
- Цели многослойной структуры - на слайде 12
- Тенденция: интерфейсы слоев (или их частей) становятся индустриальными стандартами различных функциональностей
  - ✓ Примеры: OpenGL, DirectX - графика, H264 - видео

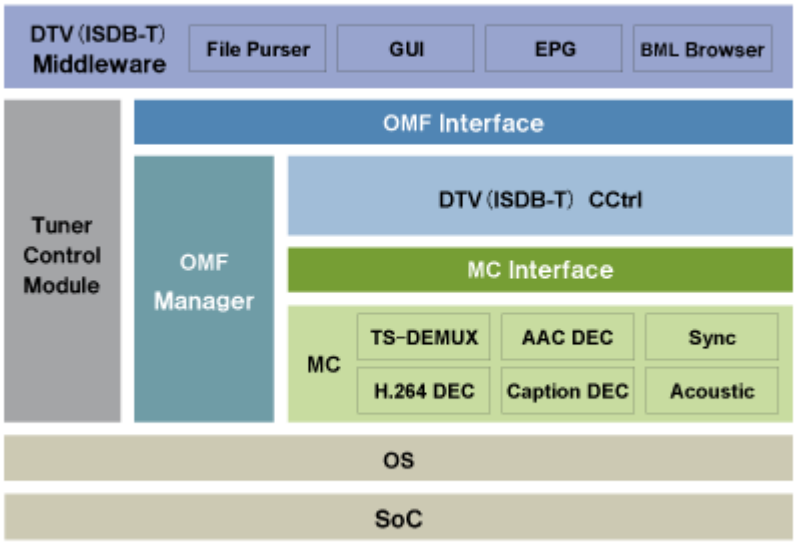
Многослойная структура ПП

(примеры – не для экзамена)



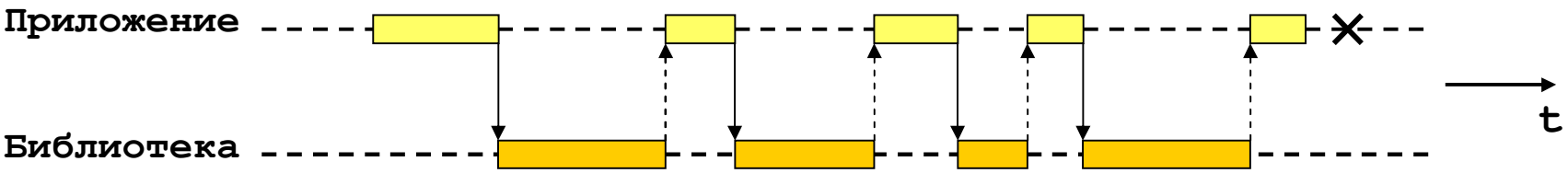




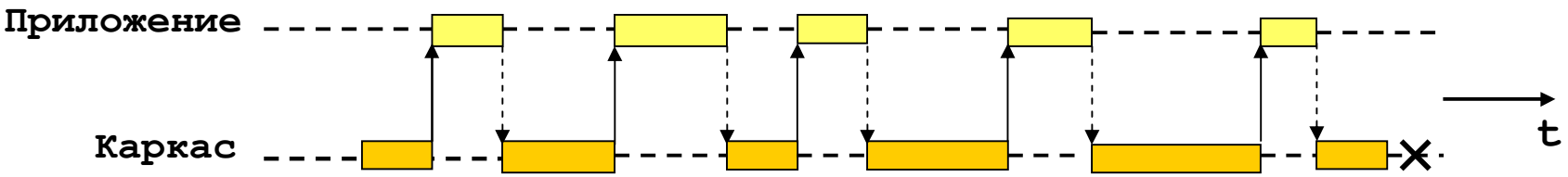


5.0

Каркас (Framework) vs. библиотека



Аналогия: программа -калькулятор



Аналогия: мастер установки программ (wizard)

Каркас (framework) – набор классов с общей (родовой) функциональностью, которая может быть переопределена (специализирована) в приложении

В отличие от обычной библиотеки, каркас:

- задает поток управления сам (а не приложение)
- имеет поведение по умолчанию

Примеры: .NET Framework, Qt (аналог MFC, но кроссплатформенный)

Понятие модуля

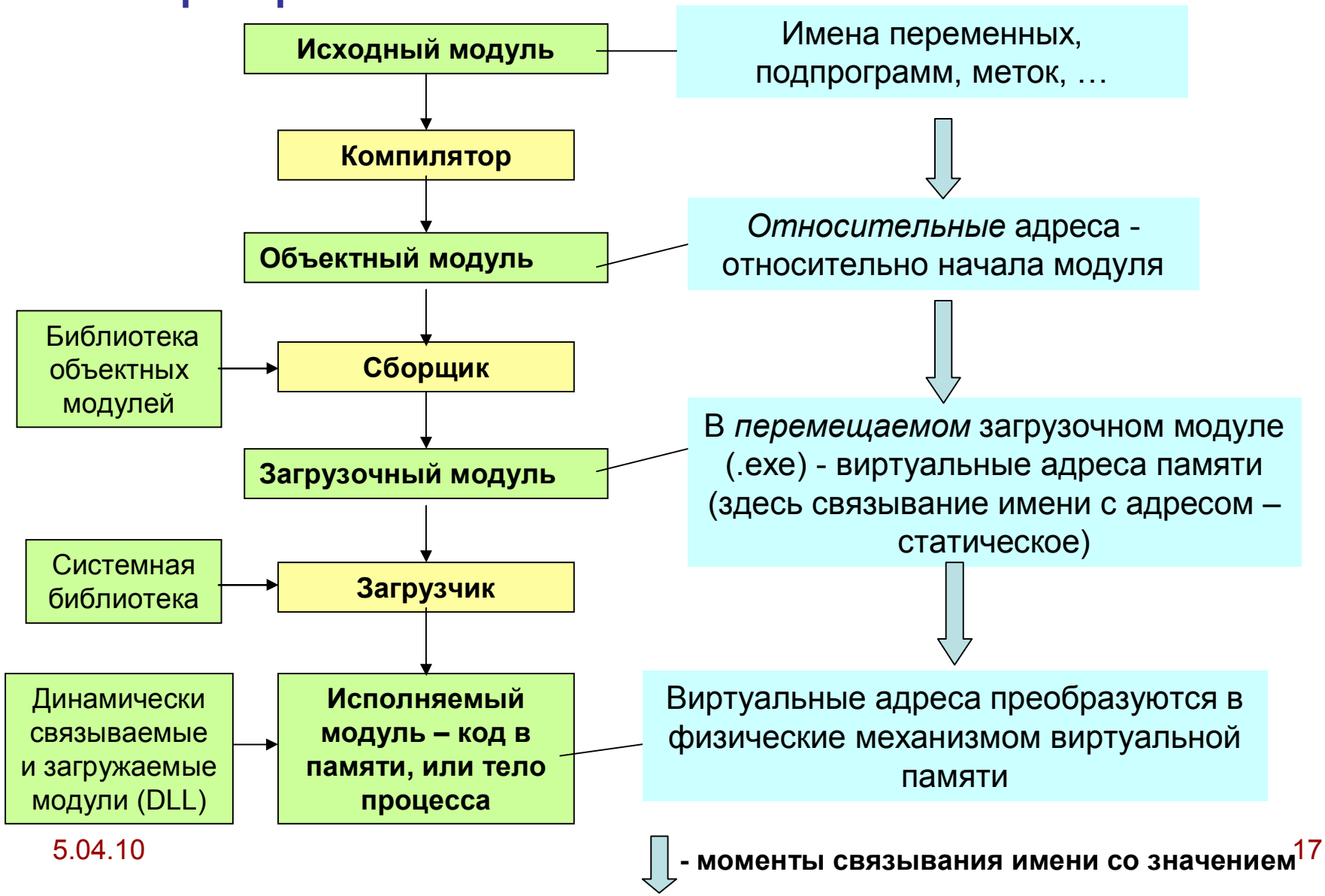
Модуль – это элемент, стандартным образом стыкуемый с другими модулями для построения составной конструкции

- Это общетехнический термин; синонимы: блок, узел
- Программный модуль - это единица компиляции и сборки
- В классических языках программирования это подпрограмма (процедура, функция) или головная программа
- *Независимая* компиляция исходных модулей и *библиотеки* объектных модулей – основа повторного их использования

История развития концепции программного модуля

Подпрограмма	-	Алгол, Фортран
Абстрактный тип данных (АТД)	-	CLU, C (файл), АДА (пакет)
Объект класса	-	SmallTalk, C++, Eiffel, Java, C#
COM-объект	-	Windows
Сборка	-	.NET

Схема подготовки и выполнения модульных программ



Структуризация функциональной модели совместно с моделью данных

Идея новых видов модульности в современных языках: объединять данные и обрабатывающие их процедуры в единый модуль и скрывать видимость всех элементов, кроме интерфейсных

В ранних, классических языках (Алгол) - три способа структуризации данных:

- **типизация** - средство разграничения видов данных и привязки к ним возможных операций - удобное средство контроля при компиляции и в период выполнения
- **агрегатирование** - составные типы (массивы, записи) - сокращение пространства имен, группирование и индексация однородных объектов

} улучшается понятность и надежность программы

- **локализация** - ограничение видимости локальных данных внутри процедур (для сокращения пространства имен и предотвращения конфликта имен)

  - подходит для сокрытия излишней информации, но только для промежуточных, короткоживущих данных (локальные переменные перестают существовать после выхода из процедуры или блока !)

Долговременные данные, общие для нескольких процедур, приходится делать глобальными для всех процедур, но их недостаток - опасность неправильного использования

Концепция абстрактных типов данных

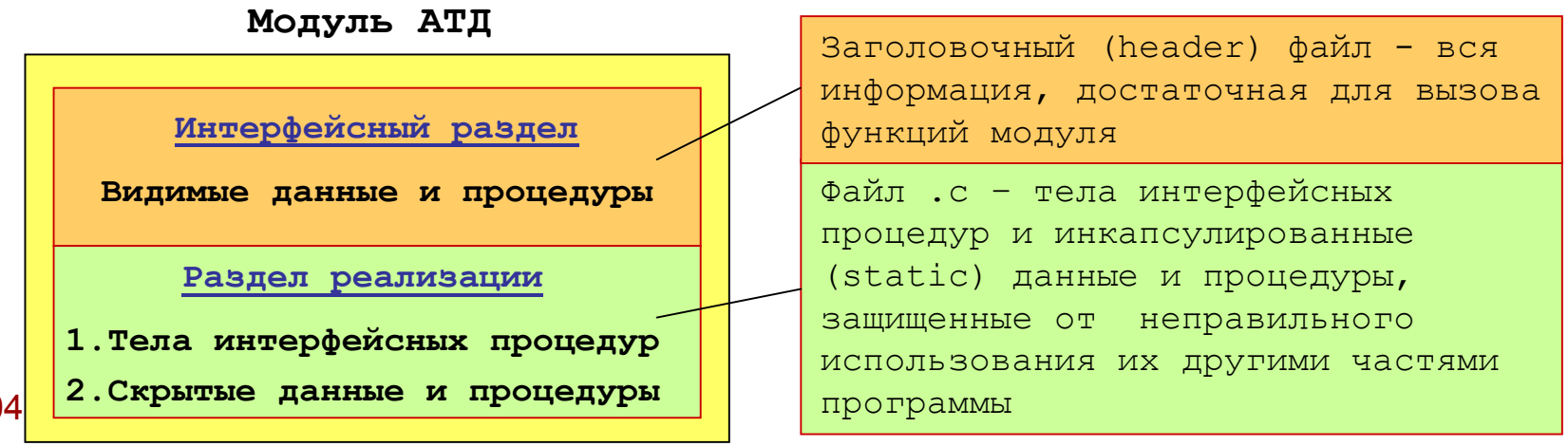
Тип данных – это множество значений данных вместе с множеством операций, определенных на них

- В ранних языках были только стандартные типы данных (integer, float, char и пр.)
- В Паскале и С появились типы, задаваемые программистом:
  а) синонимы стандартных типов (операторы Type, typedef),
  б) перечислимые типы и диапазоны (Type, enum), где задается свое множество именуемых значений (отображаемых на подмножество целых констант)

```
Type street light = {red, yellow, green}
```

В обоих случаях определяются производные типы, *наследующие* некоторые операции базового типа

- Естественное развитие: дать возможность программисту определять *свои операции*, описывая их процедурами; при этом запретив любые другие операции путем *инкапсуляции*



От АТД к классам

Большое преимущество АТД – возможность *раздельной* компиляции разделов интерфейса и реализации (.h и .c файлов в случае языка С)

Но АТД, моделируемый файлом С – не полноценный тип данных:

- он не поддерживается компилятором в смысле статического контроля
- он не передается функциям как параметр
- приходится вручную кодировать процедуры создания и уничтожения экземпляра объекта

В объектно-ориентированных (ОО) и языках простой класс – это развитый АТД - полноценный тип данных

Однако простым классам недостает гибкости: если в другом контексте тип нужно повторно использовать с небольшими изменениями, то его приходится заново описывать

Решение - *наследование* и *полиморфизм*: общие свойства объектов воплощаются в базовом классе, а специфические свойства (данные и операции) добавляются или переопределяются в производных классах

Технологические достоинства ОО–программирования vs. процедурного

- Ясность, понятность, обозримость проекта и кода
- Лучше модифицируемость кода
- Выше надежность (безошибочность)
- Повторное использование кодов *внутри* библиотек классов лучше, чем в библиотеках подпрограмм

Почему?

Технологические проблемы повторного использования ОО-программ

Проблема ООП - трудности *обновления* программ, распространяемых в виде библиотек классов

Для классов C++ возможны два варианта:

А. Распространение библиотек классов в виде исходного кода. Недостатки:

— нарушение интеллектуальных прав разработчика

— необходимость перекомпиляции *всех* приложений, использующих библиотеку, при ее обновлении

— соблазн модификации кода, адаптации под конкретное приложение, и тогда обновления разработчика становятся невозможными

Б. Упаковка класса в динамически связываемую библиотеку (DLL)

Однако и здесь во многих случаях остается необходимость перекомпиляции из-за:

- отсутствия двоичного стандарта для C++: различные компиляторы по-разному реализуют отдельные языковые особенности C++ и некоторые вопросы компоновки
- если при изменении реализации класса был изменен состав (и/или порядок) данных класса, то клиент не будет работать с новой версией DLL без перекомпиляции т.к. именно клиент отводит память под все данные экземпляра класса

  ❖ Это *проблема версий* DLL**:** установив новую версию DLL, мы можем погубить все приложения, которые были скомпилированы для работы со старой версией – The hell of DLL = Ад DLL

От классов к объектным компонентам

Компонентная технология имеет целью такую *сборку* любых программ из готовых компонентов, которая исключает специфически программистские операции – компиляцию и сборку

Степень готовности таких компонентов к выполнению выше, чем у библиотечных модулей или классов; они могут задействоваться через сеть или скачиваться из Интернета для немедленного выполнения

Появляется возможность *глобального* повторного использования компонентов непосредственно пользователями, а не программистами

Две современные технологии: .NET (развитие COM / DCOM) и Java

COM (Component Object Model) – “улучшенный C++” (1996)

ПП - один или несколько *объектов COM*, каждый из которых поддерживает один или *несколько* интерфейсов для:

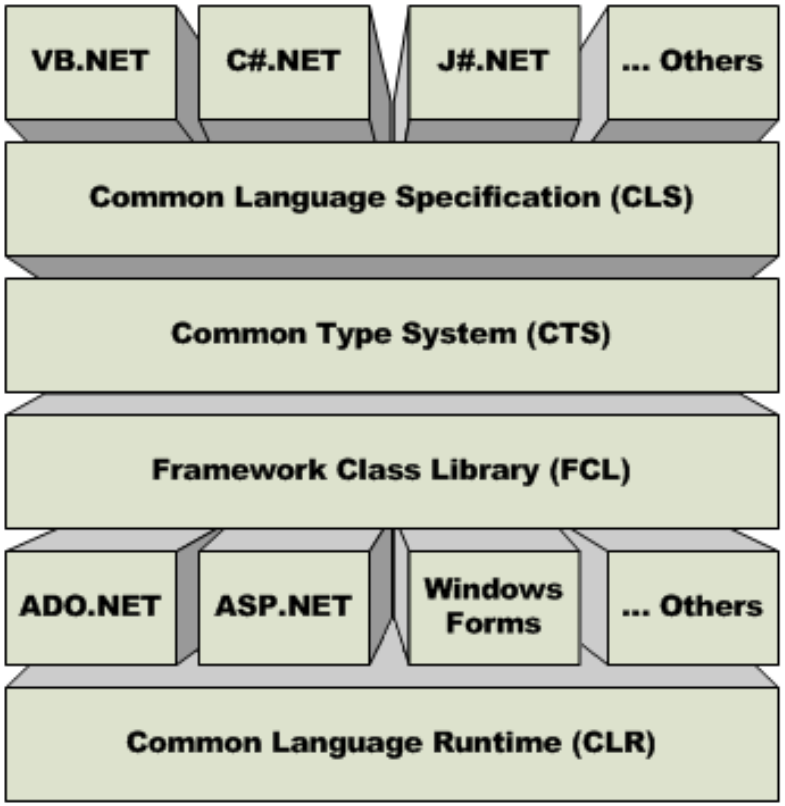
- возможности определять дифференцированный доступ для разных клиентов
- совместимости сверху вниз при расширениях (решение проблемы Hell of DLL):
  - ✓ доступ к новому сервису определяется через новый интерфейс для новых клиентов
  - ✓ старые клиенты продолжают работать со старыми интерфейсами, не зная о существовании новых (перекомпиляция приложений не нужна !)

```
Всякий COM-объект - экземпляр некоторого класса; класс в COM - это
конкретная реализация набора интерфейсов; различные реализации одного и
того же набора интерфейсов означают полиморфизм
```

Технология Java

- Java – кроссплатформенный интерпретируемый ОО язык, ориентированный на Интернет
- Более простой и надежный язык, чем C/C++
- Программы на Java транслируются в промежуточную форму - байт-код, выполняемый интерпретатором - виртуальной машиной Java (JVM)
  - Байт-код – «обобщенный» машинный язык - на полпути между исходным текстом и машинным кодом конкретной машинной архитектуры
  - Интерпретируемая программа обычно выполняется существенно медленнее скомпилированной
  - Потери производительности уменьшает технология JIT (Just In Time): трансляция байт-кода в машинный код непосредственно во время работы программы с возможностью сохранения версий класса в машинном коде
- Компоненты, готовые к выполнению – это *апплеты* на машине Web-клиента и *сервлеты* на сервере
- Существует три варианта технологии:
  - J2EE – для систем уровня предприятия
  - J2SE – для настольных систем
  - J2ME – для мобильных устройств

Каркас .NET



Главная часть технологии .NET - языковая среда .NET Framework - платформа для корпоративной и Web-разработки и общеязыковая среда выполнения CLR (Common Language Runtime)

Сборки в .NET

Новый вид компонента: *сборка (assembly)* –“логический модуль” приложения - коллекция типов (в частности, классов) и ресурсов (файлов данных)

Это основная единица управления версиями, повторного использования, установки и запроса/предоставления прав

- Вся информация, нужная для этих действий, описана в манифесте сборки, поэтому в отличие от DLL, сборки являются самодостаточными, т.е. не зависящими от системного реестра (registry)
- Клиент или различные клиенты могут работать с различными версиями одного и того же компонента – *доменами,* (а не только с различными его интерфейсами, как в COM)
- Исключаются ошибки работы с реестром, когда установка или удаление одного приложения может нарушить работу других приложений

Манифест – это метаданные, описывающие приложение. Его состав:

- ✓ Идентификация: name + version + culture (локализация)
- ✓ Список файлов
- ✓ Ссылки на другие приложения
- ✓ Экспортируемые типы и ресурсы
- ✓ Права доступа

Другие технологические возможности .NET

- Многоязыковая поддержка: компоненты, написанные на разных языках, транслируются на промежуточный язык MSIL, связываются в единый исполняемый файл переносимого формата .PE (используя общую систему типов CTS) и затем выполняются общей средой CLR
- В одном процессе можно запустить несколько доменов приложений с таким же уровнем изоляции, какой обеспечивают отдельные процессы
  - ✓ без дополнительных издержек на межпроцессные вызовы или переключение между процессами, что повышает масштабируемость серверов
- Модель XML Web-сервисов для взаимодействия программ, исполняющихся на разных платформах, по протоколу SOAP (Simple Object Access Protocol)
  - ✓ для распространения продуктов путем их аренды через Интернет
- Возможность программирования *аспектов* на декларативном языке (см. слайд 27)

Аспектно-ориентированное программирование (АОП)

- локализация кода *сквозной функциональности* в одном модуле

Сквозная, или перекрестная (crosscutting) функциональность – та, реализация которой разбросана по различным модулям программы

- Обычно это *служебная* функциональность: контекстно-зависимая обработка исключений, проверка прав доступа, и т.п.
- Ее код состоит из похожих друг на друга фрагментов в сотнях разных мест, в коде многих классов, что приводит к рассредоточенному и запутанному коду.
- Совокупность таких фрагментов называется «аспект»

(Аспект – это точка зрения, с которой рассматривается какое-либо понятие)

Идея АОП: написать заготовку нужных операторов один раз ("определить аспект"), один раз определить правило модификации аспекта в соответствии с контекстом и правило нахождения мест в тексте программы, куда вставить эти операторы

- Система АОП автоматически вставит нужные операторы везде, где требуется, подобно макроподстановкам; этот финальный процесс называется вплетением (weaving)
- Экономия трудозатрат, повышение надежности, улучшение читабельности, удобство модификации и повторного использования очевидны

Языки АОП

В .NET основной код компонента (класса) пишется на C++, VB и т.п., а аспект - на специальном декларативном (описательном, непроцедурном) языке

- ✓При вплетении выполняется автоматическое построение оптимизированного для выполнения кода (напр., на C++).

Элементы АОП вводятся в языки программирования: Python, Ruby, Java, …

AspectJ – расширение Java; его основные понятия:

- **Точка соединения** (JoinPoint) - определенная точка выполнения программы: вызов метода, точка обращения к членам класса, исполнение блоков обработчиков исключений и т. д.
- **Срез** (PointCut) - набор точек выполнения программы, заданный определенным критерием.
- **Применение** (Advice) - фрагмент кода, выполняющийся до, после или вместо точки соединения
- **Введение** (Introduction) - метод изменения статической структуры класса путем введения новых полей и методов

АОП - в русле современной многоязыковой тенденции, когда логика обработки данных реализуется в компонентах на ОО-языке, а последовательность выполнения компонентов и служебные функции – на языке сценариев

Языки сценариев (script languages)

- предназначены для небольших программ, включающих вызов модулей на других языках

Script = сценарий, рукопись; программа на языке сценариев

- Скриптовая программа:
  - обычно интерпретируется, а не компилируется
  - кроме вызовов модулей, содержит простые вычисления и обработку текстов
- Достоинства:
  - быстрота разработки и модификации
    - не нужно компиляции и сборки
  - наличие большого числа библиотек стандартных программ
- Области применения: системное администрирование, UI (в том числе Web-UI), автоматизация рутинных действий пользователя, программирование каркасов,…
- Языки: Shell, bash, Perl, Tcl, Python, PHP, JavaScript, Rubi, Lua, …

Это еще одно средство «сборочного» программирования

Уровни и стратегии проектирования и инкрементной интеграции

- Архитектура ПП:
  - ➢ состав из частей (подсистем и модулей) и интерфейсы основных модулей
  - ➢ событийная структура системы (структура поведения)
- Детальный проект:
  - ➢ интерфейсы всех модулей
  - ➢ конструкция скрытой части модулей: описание инкапсулированных данных и функций
  - ➢ алгоритмическая и/или событийная структура отдельных модулей

Все это ДО начала кодирования очередного инкремента !

Нисходящая (top-down) стратегия: начинать проектирование с основных модулей, затем – производные и специфические компоненты

Восходящая (bottom-up) стратегия – наоборот, снизу-вверх – обобщая частные случаи

Когда что предпочтительнее? - см. Макконнелл. «Совершенный код», глава 29

Заключение

- Структуризация моделей программ – мера борьбы со сложностью
- Структурное программирование (без go to) способствует ясности проекта и кода
- Поуровневая структуризация функциональной модели приводит к многослойной структуре ПП; границы слоев – это их интерфейсы
- Стандартизация интерфейсов – основа совместимости, сопрягаемости и переносимости ПП
- Развитие модулей как строительных элементов ПП - от подпрограмм до сборок - идет в направлении облегчения их повторного использования и распространения

Следующая лекция – 12 апреля

Дополнительные вопросы

1. Предложите идею алгоритма автоматического преобразования любой блок-схемы в структурную, т.е. решающего ту же задачу, что и метод Ашкрофта-Манна.
2. Почему язык ассемблера не структурен в принципе ? Каким образом делается его расширение - превращение в структурный язык?
3. Почему язык программирования нельзя считать интерфейсом, а входной язык системы (среды) программирования - можно ?
4. Интерфейс командной строки в последние годы повсеместно сменился на графический UI. Назовите другие современные и перспективные формы UI.
5. Почему излишнее знание сторон интерфейса друг о друге вредно ?
6. Почему промышленные стандарты являются и двигателем и тормозом технического прогресса одновременно?
7. В чем преимущества динамического связывания модулей (DLL) перед статическим? (слайд 17)
8. Что называется *строгой* типизацией ?
9. Чем похожи и чем отличаются инкапсулированные данные (`static` в C/C++) от глобальных и локальных переменных?
10. Каково технологическое преимущество раздельной компиляции интерфейса и реализации модулей ATD?
11. Объясните технологические достоинства ОО–программирования vs. процедурного (вопрос на слайде 20)
12. Почему программы на интерпретируемых языках (как Java или PHP) обычно выполняются медленнее, чем на компилируемых (как C/C++) ?

Непроцедурные модели описания поведения программ

Процедурная = алгоритмическая модель, она описывает последовательность шагов преобразования данных

Непроцедурные модели описывают структуру и свойства программы как процесса: состояния, реакции на воздействия, временны**ы**е свойства

Простую реакцию на входные воздействия удобно специфицировать *конечными функциями* (КФ), область задания которых - конечное множество значений (а область существования – не обязательно конечное множество)

Простейшая форма задания КФ – таблица с числом входов, равным мощности области задания

Так обычно представляются эмпирические зависимости, например: K = F (V):

Скорость V, км/ч	5	10	25	50
Коэффициент трения K	0,0212	0,0231	0,0257	0,0276

Спецификация поведения программ логическими функциями

Частный случай КФ – логическая функция; ей соответствует таблица истинности

Например, логическая функция двух переменных F(X, Y):

X: Концевой выключатель сработал	F	F	T	T
Y: Реле времени сработало	F	T	F	T
F(X, Y): Сигнал «Включить лампочку»	F	F	T	F

Обобщение – многозначная логика: мощность области значений больше двух

Соответствующая таблица называется ***таблицей решений*** (ТР)

Аргументы называются условиями, значения функции – решениями

Пример: спецификация программы контроля параметров насоса с электроприводом

Арументы - условия	Температура T > 60 C	N	Y	~	N	Y	~
	Скорость вращения < 50 об/с	N	N	Y	N	N	~
	Давление P < 1,6 атм	N	Y	~	Y	N	~
	Напряжение < 90 в	N	N	N	N	N	Y
Функция	**Решение**	O	A	A	O	W	A

Y – yes
N – No
~ – Y или N
O – норма
W – предупреждение
A – авария

Конечный функциональный преобразователь

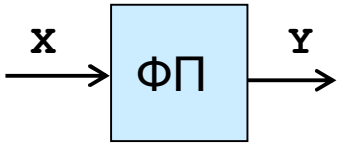
Достоинства табличной формы задания поведения программ:

- наглядность, понятность непрофессионалам
- таблица одновременно служит планом тестирования

➢ Таблицы решений удобны для проектирования программ промышленной автоматики, решающих задачи логического управления, диагностики неисправностей

➢ Существуют текстовые нотации для ТР (языки ТР) и трансляторы с них на инструментальные языки программирования (с оптимизацией: минимизация числа проверок в программе)

Все приведенные модели – варианты табличного задания дискретной кибернетической модели – конечного функционального преобразователя (ФП):



X={x1,... xn} – множество входных сигналов

Y={y1,... ym} – множество выходных сигналов

Выходной сигнал – функция входного: **f: X → Y**

Синхронный ФП срабатывает в регулярные дискретные моменты времени – *такты*

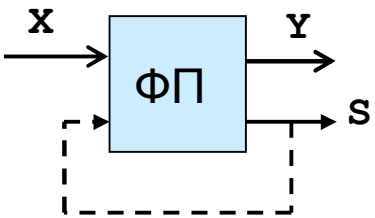
Синхронный ФП традиционно используется для описания работы *комбинационных* цифровых логических схем, т.е. схем без памяти

Понятие конечного автомата

Конечный автомат (КА, FSM - Finite State Machine) – это ФП с памятью, описывающий более сложное поведение:

- преобразование входа в выход зависит от текущего состояния автомата; состояние изменяется тоже как функция входов

- Синхронные КА традиционно служат моделями цифровых устройств
- Для описания программ интересна асинхронная модель КА: входные сигналы поступают в произвольные моменты времени



КА – это шестерка:

$X=\{x1,\ldots,xn\}$ – конечное непустое множество входных сигналов

$Y=\{y1,\ldots,ym\}$ - конечное множество выходных сигналов

$S=\{s1,\ldots,yk\}$ – конечное множество состояний

$S^0 \in S$ – начальное состояние

$Fy: X \times S \rightarrow Y$ – функция выходов

$Fs: X \times S \rightarrow S$ – функция переходов

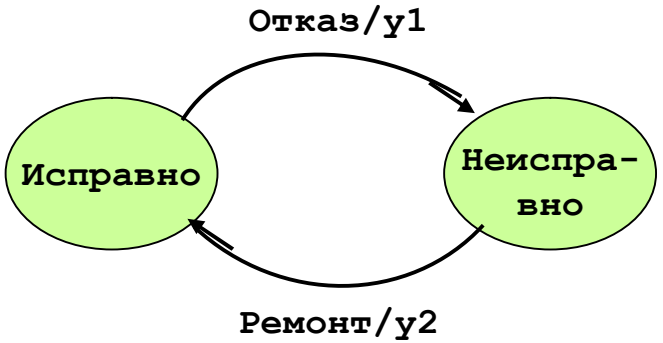
- Состояние – это:
  - Ситуация в жизненном цикле объекта, во время которой он выполняет определенную деятельность или ожидает какого-либо события
  - Фундаментальное понятие дискретных динамических систем
- Переход в новое состояние происходит мгновенно

Спецификация моделей КА

Два способа задания КА

- Табличное (как для любых конечных функций) представление функций выходов и переходов
- Диаграммой состояний и переходов (State-Transition Diagram)
  - Вершины графа – состояния, дуги - переходы

	s1: Исправно	s2: Неисправно
x1: Отказ	s2 / y1	-
x2: Ремонт	-	s1 / y2



y1 – Включить ламп. «Авария»

y2 – Выключить ламп. «Авария»

Разнообразие моделей КА и их графических нотаций:

- где описывать действия, связанные с переходом?
  - связывать их с переходом или с состоянием?
- как описывать дополнительные условия, влияющие на переход?