Библиотеки программ

С точки зрения системы программирования, библиотеки программ состоят из двух основных компонентов:

- ***файл (или множество файлов) библиотеки***, содержащий объектный код,
- ***файлы описаний*** функций, констант и переменных, составляющих библиотеку.

Описания оформляются на соответствующем входном языке (например, для языка C или C++ это будет набор заголовочных файлов).

Объектный код библиотеки (программа на машинном языке с недоопределенными внешними ссылками) подключается **компоновщиком/редактором связей** к результирующей программе при создании **исполняемого/загрузочного** модуля (абсолютно готовой к выполнению программы в машинных кодах).

Файлы описания служит для информирования компилятора о составе входящих в библиотеку функций. Обрабатывая эти файлы, компилятор получает всю необходимую информацию о составе библиотеки с точки зрения входного языка программирования. Эти файлы предназначены для того, чтобы избавить программиста от необходимости описывать заголовки библиотечных функций, константы и переменные.

В ходе развития систем программирования состав библиотек постоянно расширялся, в них включалось все большее число функций, однако принципы создания и использования библиотек претерпели мало изменений. Наиболее значимое из них заключалось в том, что некоторые базовые библиотеки, без использования которых вообще нельзя написать программу (например, библиотека ввода-вывода), стали подключаться к результирующей программе автоматически.

Пример: одна из первых в мире библиотек **Интерпретирующая Система ИС-2** (1958 год).

Система состояла из Интерпретирующей программы и собственно Библиотеки. Были описаны правила составления и оформления программ библиотеки и правила обращения к ним. Автор – Михаил Романович Шура-Бура – один из пионеров и классиков российского программирования.

Система ИС-2:
- содержала около 30 программ (элементарные функции, ввод/вывод и др.);
- была написана в машинных кодах (на ЭВМ М-20) в условных адресах;
- хранилась на внешних носителях (МЛ/МБ), программы загружались в оперативную память;
- выполнялась в режиме интерпретации.

Сама система занимала 267 ячеек (по 6 байтов), использовала рабочее поле в 277 ячеек.
Эта система – прообраз современных DLL.

Динамически подключаемые библиотеки

Принципиально новые возможности предоставляют современные операционные системы (ОС), которые поддерживают широкий набор различных методов адресации (например, сегментный, страничный, сегментно-страничный способы организации памяти). Это позволяет подключать библиотеки динамически, по ходу выполнения программы.

Динамические библиотеки (**DLL – Dynamic Link Library**) в отличие от традиционных (статических) библиотек могут подключаться к программе не в момент её компоновки, а непосредственно в ходе выполнения, как только программа затребовала ту или иную функцию, находящуюся в библиотеке.

Аргументы в пользу DLL:

- во-первых, DLL дает возможность повторного использования кода (т.е. использование один раз разработанной функции при создании нескольких приложений). К тому же, функции, хранящиеся в библиотеке, которая была разработана в какой-либо среде, могут быть вызваны на выполнение из приложений, разработанных в другой среде (например, библиотека была разработана на Object Pascal в Delphi, а используется в C++ Builder, Visual Basic и др.).
- во-вторых, использование DLL предоставляет возможность использования один раз загруженного в оперативную память кода несколькими приложениями. Естественно, что выделение общих для нескольких приложений данных в DLL может привести к достаточно существенной экономии как дискового пространства, так и оперативной памяти.

- в-третьих, с помощью DLL можно решить проблему локализации – т.е. перевода на другие языки заранее подготовленных и написанных на родном языке разработчика текстов на элементах управления (пункты меню, кнопки, подсказки), сообщений об ошибках и т.д. Одним из путей её решения может явиться создание ресурсов интерфейсов внутри DLL. К примеру, можно создать одно приложение, которое в зависимости от версии динамической библиотеки будет выводить тексты на одном из «известных» системе языков.

Загрузка DLL-библиотеки может быть осуществлена одним из двух способов: ***статическая загрузка*** и ***динамическая загрузка***. Оба метода имеют как преимущества, так и недостатки.

Статическая загрузка означает, что динамическая библиотека загружается автоматически при запуске на выполнение использующего ее приложения. Для того чтобы применить такой способ загрузки, обычно необходимо особым образом описать экспортируемую из динамической библиотеки функцию. DLL автоматически загружается при старте программы, и любые экспортируемые из нее подпрограммы можно использовать точно так же, как если бы они были описаны внутри модулей приложения. Это наиболее легкий способ использования кода, помещенного в DLL.

Смысл динамического метода заключается в том, что библиотека загружается не при старте приложения, а в тот момент, когда это действительно необходимо (т.е., в момент, когда при выполнении приложения происходит обращение к той или иной программе библиотеки). Очевидно, что если функция, описанная в динамической библиотеке, нужна только при 10% запусков программы, то статический метод загрузки оказывается неэффективным. Еще одно преимущество такого способа загрузки DLL - это уменьшение времени старта приложения. В то же время использование данного метода является более хлопотным, чем статическая загрузка (например, в Delphi сначала необходимо воспользоваться функцией Windows API LoadLibrary, затем для получения указателя на экспортируемую процедуру или функцию должна использоваться функция GetProcAddress; после завершения использования библиотека DLL должна быть выгружена с применением FreeLibrary).

Широкий набор динамических библиотек поддерживается всеми современными ОС. Как правило, они содержат системные функции ОС и общедоступные функции программного интерфейса (API – application program interface). Кроме того, многие независимые разработчики предоставляют для различных систем программирования свои динамические библиотеки как отдельные товары на рынке средств разработки прикладных программ.

Использование библиотек, в том числе и динамических, накладывает определенные обязательства как на разработчика программ, так и на создателя самой библиотеки. Разработчик прикладной программы должен использовать библиотеку, опираясь только на доступный ему интерфейс, не привлекая никаких знаний о внутреннем устройстве библиотеки (даже если они ему стали каким-то образом известны); создатель библиотеки в случае её модификации никаким образом не должен модифицировать доступный пользователю интерфейс.

Распространение динамически подключаемых библиотек и ресурсов прикладных программ привело к ситуации, когда большинство прикладных программ стало представлять собой не единый программный модуль, а набор сложным образом взаимосвязанных между собой компонентов.

Функциональное, системное и информационное наполнение Пакетов Прикладных Программ

Пакет прикладных программ – комплекс взаимосвязанных прикладных программ (библиотека) и системных средств для решения некоторого класса задач прикладной области.

Функциональное наполнение ППП – комплекс взаимосвязанных прикладных программ для решения типовых задач некоторой прикладной области (множество файлов библиотеки, содержащих объектный код).

Системное наполнение ППП – комплекс взаимосвязанных системных средств для решения некоторого класса характерных задач этой прикладной области (выбор походящих программ библиотеки и их компоновка, интерфейс с пользователем и др.).

Информационное наполнение ППП – описание технологии работы с конкретным ППП как для пользователя, так и для лиц, поддерживающих ППП (HELP, инструкции пользователю, описание языка и сценария общения, инструкции по составлению новых программ и включению их в ППП).

Можно говорить и о:

внутреннем информационном наполнении ППП – аналоге файлов описания из библиотек. Оно служит для информирования компилятора о составе входящих в библиотеку функций.

Четыре поколения языков общения с машиной. Языки 4GL

В компьютерных науках выделяют ***Четыре поколения языков общения с машиной*** **(Generation Languages – GL)**:

1GL - машинные языки
2GL - автокоды (языки ассемблера)
3GL - языки программирования высокого уровня
4GL - простейшие языки запросов информационных систем,
языки запросов, ЯОД, ЯМД (баз данных),
языки генераторов отчетов,
языки поддержки принятия решений (статистический / прогнозный анализ),
языки генераторов приложений (RPG - Report Program Generator)

Считается, что переход к каждому новому поколению повышает производительность работы с компьютером примерно на порядок (в 10 раз).

Качественное отличие языков 4GL от языков предыдущих поколений – заключается в том, что эти языки служат ***не для описания алгоритмов***, а ***для спецификации решаемых пользователем задач*** (описывается, ***что*** нужно получить, но не описывается, ***как*** это получить).
Языки 4GL – должны обеспечивать дружественный (интеллектуальный) интерфейс.
Функциональные требования к 4GL:
интегрированное множество функций,
язык очень высокого уровня,
совместимость с языками 3GL,
дружественный интерфейс,
средства анализа и моделирования данных,
высокая эффективность при сильной нагрузке,
поддержка стандартных СУБД

Интеллектуализация ЭВМ

Интеллектуальный интерфейс – совокупность программных и аппаратных средств, позволяющая конечному пользователю решать на компьютере характерные для его повседневной деятельности задачи без помощи посредников-программистов.

В идеале система должна иметь «модель мира задачи», над которой работают система и пользователь и которая близка модели этого мира в уме пользователя.

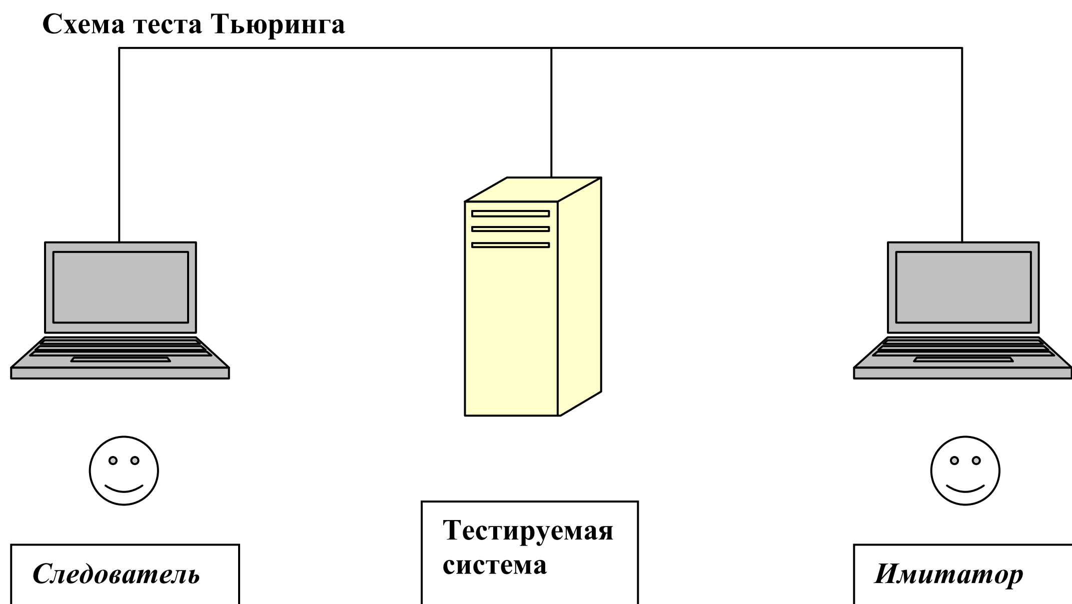
Тест Тьюринга

Автор – один из основоположников кибернетики Алан Тьюринг (США). Описан в Журнале Mind в 1950 году.

За терминалом работает ***Следователь***. Его терминал связан с терминалом, за которым работает ***Имитатор***, и с компьютером, на котором установлена ***Система Искусственного интеллекта (Система ИИ)***. Следователь обращается к своему «собеседнику» с вопросами, предлагает решить задачи. Кто отвечает ему (Имитатор или Система ИИ), он не знает. Это определяется по датчику случайных чисел. Если в течение достаточно длительного времени Следователь не может отличить ответы человека (Имитатора) от ответов машины (Системы ИИ), то машину «можно считать разумной».

Несмотря на условность и неформальность теста Тьюринга, он:
- дает объективное понятие об интеллекте (задан стандарт для определения разумности / интеллектуальности);
- позволяет оставаться на функциональном уровне (не нужно знать, какие механизмы использует Система ИИ;
- может использоваться для тестирования / аттестации систем ИИ.

Схема теста Тьюринга



Понятие об автоматическом синтезе программ

Автоматический синтез программ – автоматическое/автоматизированное построение программы по описанию ее назначения или действия, называемому обычно ***спецификацией программы***.
Термины: ***спецификация программы*** и ***спецификация задачи*** – весьма близки и иногда считаются синонимами. Мы будем считать, что ***спецификация программы*** может содержать информацию об особенностях алгоритма (о том, ***как*** получить результат), а спецификация задачи «говорит» только о том, ***что*** (какой результат) нужно получить.
Пример: спецификация задачи вычисления площади круга, вписанного в квадрат с диагональю 5 (в системе ПРИЗ).

```
ПРОГРАММА'  ПРИМЕР;
      ПО'  ГЕОМ;
      ПУСТЬ'     К1: КВАДРАТ ДИАГОНАЛЬ  =  5;
                 К2: КРУГ ДИАМЕТР  =  К1.СТОРОНА;
      ДЕЙСТВИЯ'
      НА'ПРИМЕР  ВЫЧИСЛИТЬ' К2.ПЛОЩАДЬ;
```

Рассматриваемые в задаче объекты ***круг*** и ***квадрат*** описаны в семантической модели геометрии (**ГЕОМ**). Отношение, связывающее элементы этих объектов – сторона квадрата равна диаметру круга. Величина диагонали квадрата равна 5, искомая величина – площадь круга.
По такой спецификации система ПРИЗ стрит программу (на языке Фортран) для вычисления площади круга.

Основные подходы к синтезу программ

Дедуктивный синтез:
Построить программу по заданному отношению между входными и выходными параметрами.

Пусть задано x, удовлетворяющее условию $P(x)$ – входные условия; по x нужно вычислить z, удовлетворяющее выходным условиям программы $R(x, z)$.

Пример:
Написать программу, выдающую следующий результат:
a – если входной параметр x делится на 7,
b – если входной параметр x не делится на 7 и делится на 9,
c – в остальных случаях.

Введем отношения: ***D(x, y)*** – ***x* делится на *y***, ***R(x, z)*** – **при входном *x* результат есть** *z*.
Формальная спецификация программы (в некотором специальном виде) такова:
$\neg D(x, 7) \vee R(x, a)$
$D(x, 7) \vee \neg D(x, 9) \vee R(x, b)$
$D(x, 7) \vee D(x, 9) \vee R(x, c)$
Синтез программы сводится к доказательству возможности выполнения заданных условий и «извлечению» текста программы из доказательства.

Индуктивный синтез:
Построить программу по примерам (следу) ее работы.

Пример:

По следу:	может быть построена такая программа:
X [1] → Y	1. X [1] → Y
X [2] + Y → Y	2. 2 → K
X [3] + Y → Y	3. X [K] + Y → Y
X [4] + Y → Y	4. K + 1 → K
ОСТАНОВ	5. если K=размерности массива⇒ОСТАНОВ
	иначе переход на команду № 3

Трансформационный синтез:
Построить программу для решения некоторой задачи на основе сходной задачи и готовой программы для ее решения.

Дано: спецификация *S*; программа, решающая соответствующую задачу *P*; спецификация *S'*. Нужно построить программу *P'*.

Частный случай – оптимизация программы (рекурсия → цикл). Здесь *S* = *S'*.