Содержание:

Введение

На сегодняшний день редко встречаются программы, которые способны работать абсолютно закрыто, без какой-либо связи с другими программами. Поэтому разработчику программного обеспечения необходимо предусматривать возможность использования в создаваемом приложении других программ, для выполнения определенных функций [4]. Для этого применяется набор стандартизованных запросов (или прикладных программных интерфейсов), определенных для программы, к которой адресован запрос.

Актуальность работы обусловлена необходимостью изучения особенностей прикладных программных интерфейсов для создания качественного программного обеспечения, которое будет успешно решать поставленные задачи, иметь возможность взаимодействовать с другими программами, при этом имея возможность модификации и усовершенствования.

Предмет исследования – прикладные программные интерфейсы.

Цель исследования – изучить особенности программных интерфейсов, их эволюцию, и рассмотреть самые популярные из существующих прикладных программных интерфейсов.

В первой главе даются определения программных интерфейсов, рассматриваются их основные особенности, и некоторые правила их создания.

Во второй главе проведена классификация и обзор существующих на данный момент стандартизированных программных интерфейсов, также рассмотрена их эволюция.

Программные интерфейсы

1.1 Особенности программных интерфейсов

Согласно общему определению **Интерфейс** - это связь двух отдельных сущностей. Интерфейсы бывают: языковые, программные, аппаратные, пользовательские, цифровые и т. д. Например, аппаратный интерфейс (port) - это способ преобразования входных/выходных данных во время объединения компьютера с периферийным оборудованием. В языках программирования - это программа или часть программы, в которой определяются константы, переменные, параметры и структуры данных для передачи другим.

В программировании термин **программный интерфейс приложения, интерфейс прикладного программирования** (англ. ***a****pplication* ***p****rogramming* ***i****nterface*, *API*) - значит полное описание всех возможных способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой.

API определяет функциональность, которую предоставляет программа (модуль, библиотека), при этом API позволяет абстрагироваться от того, как именно эта функциональность реализована.

При этом обычно компоненты образуют иерархию — высокоуровневые компоненты используют API низкоуровневых компонентов, а те, в свою очередь, используют API ещё более низкоуровневых компонентов.

В современном мире большая часть программного обеспечения требует использования других программ для выполнения определенных функций [5]. Для этого применяется набор стандартизованных запросов (или прикладных программных интерфейсов), определенных для программы, к которой адресован запрос. Например, почти каждая программа обращается к API базовой операционной системы для выполнения таких основных функций, как доступ к файловой системе. Хотя существуют и абсолютно закрытые программы, которые вообще не предоставляют никакого API.

Существуют классификации интерфейсов программирования по уровню реализации, например по А.В. Гордееву [1] выделяется три варианта реализации:

- на уровне модулей операционной системы;

- на уровне системы программирования;

- на уровне внешних библиотек процедур и функций.

При реализации функций API на уровне модулей операционной системы ответственность за выполнение функций API несёт ОС. Объектный код, выполняющий функции, или непосредственно входит в состав операционной системы, или содержится в составе динамически загружаемых библиотек.

При реализации функций API на уровне системы программирования эти функции предоставляются пользователю в виде библиотеки функций RTL (Run Time Library) соответствующего языка программирования. RTL перенаправляет системный вызов соответствующим обработчикам программных прерываний, входящим в состав операционной системы.

При реализации функций API с помощью внешних библиотек эти функции предоставляются пользователю в виде библиотеки процедур и функций, созданной сторонним разработчиком.

При создании программного обеспечения, разработчику, скорее всего, придётся включить в приложение прикладные программные интерфейсы. При их создании желательно соблюдение следующих правил:

1) **Независимость от платформы**. Любой клиент должен иметь возможность вызывать API, независимо от того, как API реализован внутренне [6].

2) **Возможность развития**. API должен предусматривать возможность развиваться и расширять набор функций, независимо от приложений, которые могут его использовать. По мере развития API все имеющиеся функции должны быть доступными, чтобы другие приложения могли полноценно их использовать [6].

В большинстве современного программного обеспечения программный интерфейс соответствует какому-либо стандарту, и сопровождается документацией [4,7,8]. Документация облегчает использование программы пользователями, кроме того, для качественной программы должна быть предусмотрена возможность внесения изменений и усовершенствований. Поэтому разработчики приложений и производители должны постоянно думать о том, будут ли их прикладные программные интерфейсы понятны последующим разработчикам. Но, несмотря на это, некоторые производители оставляют свои API недокументированными.

На данный момент общепринято правило, что любое серьёзное разрабатываемое приложение должно также удовлетворять следующим требованиям [8]:

- расширяемость/масштабируемость*:* это возможность добавления новых функций в программу, или изменения некоторых уже имеющихся при неизменных остальных функциональных частях программы;

- мобильность/переносимость*:* это возможность переноса программ, данных при модернизации или замене аппаратных платформ, и возможность работы с ними специалистов, пользующихся ИТ, без их переподготовки при изменениях программы;

- интероперабельность: способность к взаимодействию с другими программами;

- дружественность к пользователю.

Все эти общепринятые свойства, взятые по отдельности, имелись и в предыдущих поколениях программного обеспечения и средств вычислительной техники. Новый взгляд проявляется в том, что эти черты рассматриваются в совокупности, как взаимосвязанные, и реализуются в комплексе, что вполне естественно, поскольку все указанные выше свойства дополняют друг друга. Только в совокупности возможности открытых систем позволяют решать проблемы современного программного обеспечения.

Для выполнения этих требований для программного обеспечения существуют согласованный набор международных стандартов информационных технологий и профилей функциональных стандартов, которые специфицируют интерфейсы, службы и поддерживающие их форматы, чтобы обеспечить интероперабельность и мобильность приложений.

Например, структура приложения может представляться состоящей из двух взаимодействующих частей:

- функциональной части, включающей прикладные программы, которые реализуют функции прикладной области;

- среды или системной части, обеспечивающей исполнение прикладных программ.

С этим разделением тесно связаны две группы вопросов стандартизации:

- стандарты интерфейсов взаимодействия прикладных программ со средой ИС (Application Program Interface - API);

- стандарты интерфейсов взаимодействия самой ИС с внешней для нее средой (External Environment Interface - EEI).

Эти две группы интерфейсов определяют спецификации внешнего описания среды программного обеспечения - архитектуру с точки зрения конечного пользователя, проектировщика, прикладного программиста, разрабатывающего функциональные части программы.

Спецификации внешних интерфейсов среды программного обеспечения и спецификации интерфейсов взаимодействия между компонентами самой программы - это точные описания всех необходимых функций, служб и форматов определенного интерфейса.

Развитие интерфейсов в языках программирования

На начальном этапе программирования в роли интерфейса выступали операторы обращения к процедурам и функциям программ через формальные параметры. Программы, процедуры и функции записывались в одном языке программирования. Операторы обращения включали имена вызываемых объектов (процедур и функций) и список фактических параметров, задающих значения формальным параметрам и получаемым результатам. Последовательность и число формальных параметров соответствовало фактическим параметрам. Выполнение функции в среде программы на одном языке программирования не вызывало проблем, так как типы данных параметров совпадали.

В случае, когда один из элементов (программа, процедура или функция) были записаны на разных языках программирования и, кроме того, располагались на разных компьютерах, возникали проблемы неоднородности типов данных в этих языках программирования, структур памяти платформ компьютеров и операционных сред, где они выполнялись. Понятие интерфейса, как самостоятельного объекта, сформировалось в связи со сборкой или объединением разноязыковых программ и модулей в монолитную систему на больших ЭВМ.

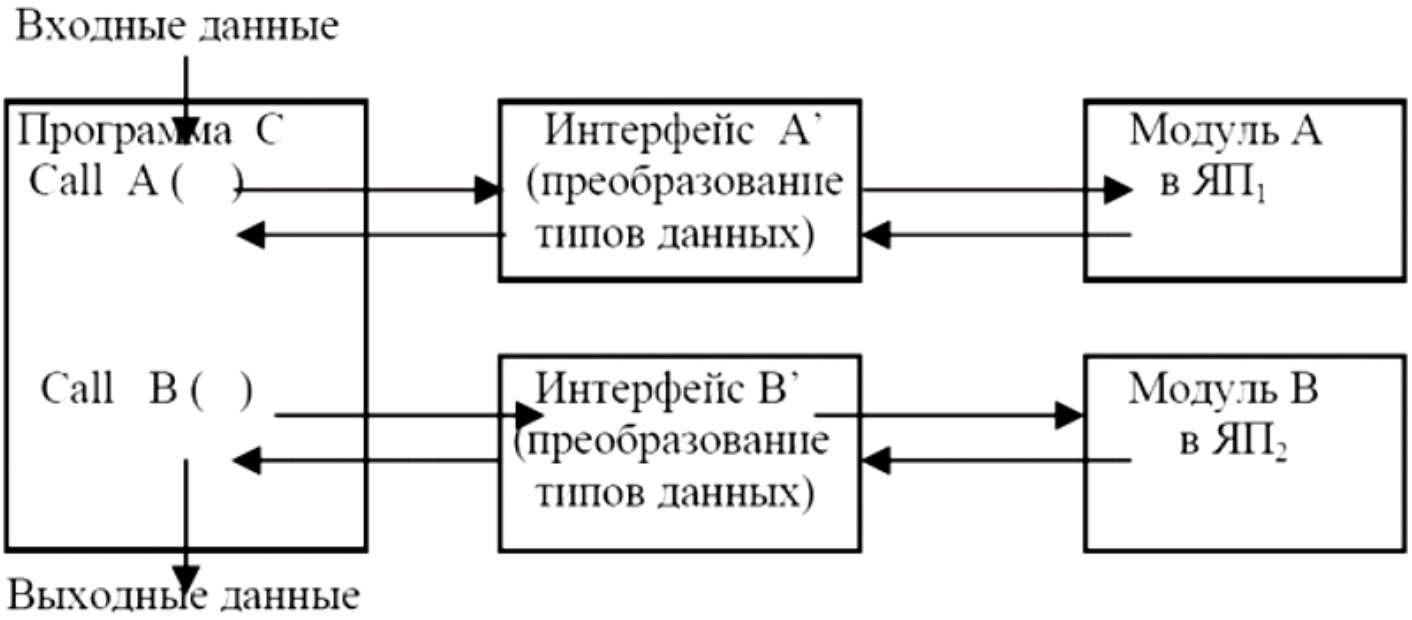

Рисунок. 1. Схема вызова модулей A и B из C через интерфейсы A'и B'

Интерфейс играл роль посредника между вызываемым и вызывающим модулями. В нем давалось описание формальных и фактических параметров, производилась проверка соответствия передаваемых параметров (количества и порядка расположения), а также их типов данных. Если типы данных параметров оказывались не релевантными (например, передается целое, а результат функции - вещественное или наоборот), то производилось прямое и обратное их преобразование с учетом структуры памяти компьютеров.

На рис. 1 приведена схема программы С, в которой содержатся два вызова – CallA() и CallB(), с параметрами, которые через интерфейсные модули-посредники A’ и B’ производят преобразование данных и их передачу модулям A и B. После выполнения A и B результаты преобразуются обратно к виду программы С.

В современных средах используется объектно-ориентированный подход [4-8] ООП.

В ООП главным элементом является класс, включающий множество объектов с одинаковыми свойствами, операциями и отношениями. Класс имеет внутреннее (реализацию) и внешнее представление – интерфейс.

Структуру представления класса и интерфейса можно определить так:

Класс	
Внешнее представление	**Внутреннее представление**
Интерфейсные операции: публичные, доступные всем клиентам: защищенные, доступные классу п подклассу: приватные, доступные классу.	Реализация операций класса, определение поведения.

Рисунок 2. Структура представления класса и интерфейса

Интерфейс содержит множество операций, описывающих его поведение. Класс может поддерживать несколько интерфейсов, каждый из которых содержит операции и сигналы, используются для задания услуг класса или программного компонента. Интерфейс именует множество операций или определяет их сигнатуру и результирующие действия. Если интерфейс реализуется с помощью класса, то он наследует все его операции.

Одни и те же операции могут появляться в различных интерфейсах. Если их сигнатуры совпадают, то они задают одну и ту же операцию, соответствующую поведению системы. Класс может реализовывать другой класс через интерфейс.

Операции и сигналы могут быть связаны отношениями обобщения. Интерфейс-потомок включает в себя все операции и сигналы своих предков и может добавлять собственные путем наследования всех операций прямого предка, т.е. его реализацию можно рассматривать как наследование поведения.

Таким образом, можно сделать вывод, что программный интерфейс определяет функциональность, которую предоставляет программа (модуль, библиотека), при этом API позволяет абстрагироваться от того, как именно эта функциональность реализована.

2. Существующие стандарты программных интерфейсов

Приведём наиболее распространённые API:

Операционных систем:

Amiga ROM Kernel, Cocoa, Linux Kernel API, OS/2 API, POSIX, Windows API;

Графических интерфейсов:

DirectDraw/Direct3D (часть DirectX), GDI, GDI+, GTK+, SFML, Motif,

OpenGL, OpenVG, Qt, SDL, Tk, Vulkan, X11, wxWidgets, Zune;

Звуковых интерфейсов:

DirectMusic/DirectSound (часть DirectX), OpenAL

2.1. API операционных систем

Практически все операционные системы (UNIX, Windows, OS X и т. д.) имеют API, с помощью которого программисты могут создавать приложения для этой операционной системы.

Для разработчиков приложений все свойства конкретной операционной системы выражаются в свойствах её API. Поэтому операционные системы с различной внутренней организацией, но с одинаковым набором функций API представляются для них одной и той же ОС [2,3]. Это упрощает стандартизацию операционных систем, и обеспечивает переносимость приложений между внутренне различными ОС, соответствующими определенному стандарту на API. Например, общий стандарт API UNIX, одним из которых является стандарт Posix, позволяет говорить о некоторой обобщенной операционной системе UNIX, хотя многочисленные версии этой ОС от разных производителей иногда существенно отличаются внутренней организацией [2].

Приложения выполняют обращения к функциям API с помощью **системных вызовов**. Способ реализации системных вызовов зависит от структурной организации ОС, которая, в свою очередь, тесно связана с особенностями аппаратной платформы. Кроме того, он зависит от языка программирования. Например, в ОС UNIX системный вызов похож на вызов подпрограммы [2,3].

Операционная система должна обеспечивать удобный интерфейс не только для прикладных программ, но и для человека, работающего за терминалом - пользователя, администратора ОС или программиста.

Современные ОС поддерживают развитые функции пользовательского интерфейса двух типов: *алфавитно-цифровой и графический.*

В области программного обеспечения общие стандартные API для стандартной функциональности играют важную роль, так как они гарантируют, что все программы, использующие общий API, будут одинаково хорошо работать, или хотя бы более-менее обычным образом. В случае API графических интерфейсов это значит, что программы будут иметь похожий пользовательский интерфейс. Это даст возможность быстрого освоения новых программных продуктов.

С другой стороны, отличия в API различных операционных систем существенно затрудняют перенос приложений между платформами. Существуют различные методы обхода этой сложности — написание «промежуточных» API (**API графических интерфейсов** wxWidgets, GTK и т. п.), написание библиотек, которые отображают системные вызовы одной ОС в системные вызовы другой ОС (такие среды исполнения, как Wine, cygwin и т. п.), введение **стандартов кодирования** в языках программирования (например, стандартная библиотека языка C), написание **интерпретируемых языков**, реализуемых на разных платформах (sh, Python, Perl, PHP, Tcl, Java и т. д.).

Также в распоряжении программиста часто находится несколько различных API, позволяющих добиться одного и того же результата. При этом каждый API обычно реализован с использованием API программных компонент более низкого уровня абстракции.

Небольшой пример: для того, чтобы увидеть в браузере строку «Hello, world!», нужно лишь создать HTML-документ с минимальным заголовком, и с простейшим телом, содержащим данную строку. При открытии браузером этого документа, программа-браузер передаёт имя файла (или уже открытый дескриптор файла) библиотеке, обрабатывающей HTML-документы, а та при помощи API операционной системы прочитает этот файл, разберётся в его устройстве, затем последовательно вызовет через API библиотеки стандартных графических примитивов операции типа «очистить окошко», «написать „Hello, world!“ выбранным шрифтом». При выполнении этих операций библиотека графических примитивов обратится к библиотеке оконного интерфейса с соответствующими запросами, а уже эта библиотека обратится к API операционной системы, чтобы записать данные в буфер видеокарты.

В этом примере на каждом из уровней существует несколько возможных альтернативных API: мы могли бы писать исходный документ не на HTML, а на LaTeX, для отображения могли бы использовать различные браузеры. При этом различные браузеры используют различные HTML-библиотеки, всё это может быть собрано с использованием различных библиотек примитивов, и на различных операционных системах.

Таким образом, проявляются основные сложности при использовании современных многоуровневых системах API:

- Сложность портирования программного кода с одной системы API на другую, например, при смене операционной системы;

- Потеря функциональности при переходе с более низкого уровня на более высокий. Каждый уровень API создаётся для облегчения выполнения некоторого стандартного набора операций. Но при этом или затрудняется, или становится принципиально невозможным выполнение некоторых других операций, которые предоставляет более низкий уровень API.

Рассмотрим наиболее используемые на сегодняшний день стандарты API операционных систем:

Windows API - общее наименование целого набора базовых функций интерфейсов программирования приложений операционных систем семейств Windows. Является самым прямым способом взаимодействия приложений с Windows.

Windows API спроектирован для использования в языке Си для написания прикладных программ, предназначенных для работы под управлением операционной системы MS Windows. Работа через Windows API — это наиболее близкий к операционной системе способ взаимодействия с ней из прикладных программ. Более низкий уровень доступа, необходимый только для драйверов устройств, в текущих версиях *Windows* предоставляется через Windows Driver Model.

Windows API представляет собой множество функций, структур данных и числовых констант, следующих соглашениям языка Си. В то же время, конвенция вызова функций отличается от cdecl, принятой для языка C: Windows API использует stdcall (winapi). Все языки программирования, способные вызывать такие функции и оперировать такими типами данных в программах, исполняемых в среде Windows, могут пользоваться этим API. В частности, это языки C++, Pascal, Visual Basic и

многие другие. [9]

Windows API состоит из нескольких тысяч вызываемых функций, которые разбиты на следующие основные категории:

1) Базовые службы (Base Services).

2) Службы компонентов (Component Services).

3) Службы пользовательского интерфейса (User Interface Services).

4) Графические и мультимедийные службы (Graphics and Multimedia Services).

5) Обмен сообщениями и совместная работа (Messaging and Collaboration).

6) Сеть (Networking).

7) Веб-службы (Web Services).

Описание Windows API можно найти в документации по набору инструментальных средств разработки программного обеспечения - Windows Software Development Kit (SDK). Эта документация доступна на веб-сайте www.msdn.microsoft.com. Она также включена со всеми уровнями подписки в сеть Microsoft Developer Network (MSDN), предназначенную для разработчиков.

POSIX (Portable operating system interfaces) - Интерфейсы переносимых операционных систем – это группа стандартов, созданная специалистами США под эгидой IEEE (Institute of Electrotechnical and Electronics Engineers), обеспечивающих аффективную по трудоемкости переносимость прикладных программ между различными аппаратными и операционными платформами.

Данный набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой (системный API), библиотеку языка С и набор приложений и их интерфейсов. Стандарт создан для обеспечения совместимости различных UNIX-подобных операционных систем и переносимости прикладных программ на уровне исходного кода, но может быть использован и для не-Unix систем.

Данная группа стандартов сосредоточили проблему переноса программ на унификации интерфейсов операционных систем ЭВМ с различными прикладными программами, а также с окружающей средой: с пользователями, базами данных, средствами коммуникации. Эти стандарты не ориентированы на определенную,

конкретную архитектуру ЭВМ, однако предполагают использование современной операционной среды Unix System V как стандарта де-факто, а также международных стандартов на языки программирования и стандартов верхних уровней взаимосвязи открытых систем (ВОС — ISO 07498). В совокупности они образуют нормативную базу открытых компьютерных систем - OCS, обеспечивающих разработку переносимых программных средств и баз данных.

При формировании концепции стандартов POSIX были поставлены следующие задачи:

- содействовать облегчению переноса кода прикладных программ на иные платформы;

- способствовать определению и унификации интерфейсов заранее, а не в процессе их реализации;

- учитывать все главные, созданные ранее и используемые прикладные программы;

- определять необходимый минимум интерфейсов для ускорения создания, одобрения и утверждения документов;

- развивать стандарты в направлении обеспечения коммуникационных сетей, распределенной обработки данных и защиты информации;

-рекомендовать ограничивать использование бинарного (объектного) кода для приложений в простых системах.

Все стандарты POSIX имеют рекомендательный характер. Они не должны служить препятствием для переноса объектного кода, ограничивать или ухудшать исполнение приложений при стандартизированных интерфейсах и ограничивать формирование новых унифицированных интерфейсов по мере необходимости.

Cocoa (в пер. с англ. — какао) — объектно-ориентированный API для операционной системы macOS производства компании Apple. Это один из пяти основных API, доступных в Mac OS X, — - Cocoa, Carbon, Toolbox (для работы старых приложений Mac OS 9), POSIX и Java. Такие языки, как Perl, Python и Ruby, не считаются основными, так как на них пока что пишется не так много серьёзных приложений для Mac OS X.

Приложения, использующие Cocoa, обычно разрабатываются с помощью среды разработки Apple Xcode с использованием языков программирования: C, Objective-C

и Swift. Однако, среда Cocoa также доступна и при разработке на других языках, таких как Ruby, Python и Perl с помощью связующих библиотек (MacRuby, PyObjC и CamelBones соответственно). Также можно писать Cocoa-программы на Objective-C в обычном текстовом редакторе и вручную компилировать их с помощью GCC или make-сценариев для GNUstep.

С точки зрения конечного пользователя, Cocoa-приложения - это приложения, написанные с использованием программной среды Cocoa. Такие приложения обычно имеют характерный внешний вид, поскольку эта среда во многом упрощает поддержку принципов «человечного интерфейса» Apple (Apple Human Interface Guidelines).

Одной из особенностей среды Cocoa является механизм для управления динамически выделяемой памятью. В классе NSObject, от которого порождается большинство классов Cocoa, как стандартных, так и пользовательских, для управления памятью реализован механизм подсчёта ссылок (reference counting). Когда количество ссылок достигает нуля, объект удаляется, и занимавшаяся им память освобождается (высвобождение памяти для объектов Objective-C - это то же, что и вызов деструктора у объектов C++. Метод dealloc делает примерно то же самое, что и деструктор в C++. Его вызов не гарантируется.).

Кроме подсчёта ссылок, программисты могут воспользоваться автоматически высвобождаемыми пулами (autorelease pools). они обычно создаются и высвобождаются в начале и в конце цикла сообщений, гарантируя, что выполнение программы выйдет за пределы блока, в котором объекты были зарегистрированы для автоматического высвобождения. Это означает, что приложение выполняется предсказуемо, и освобождение памяти происходит прозрачно для пользователя, в то время как при использовании автоматического сборщика мусора в большинстве случаев программа неожиданно перестаёт реагировать на действия пользователя при его запуске.

Cocoa состоит в основном из двух библиотек объектов Objective-C, называемых фреймворками (Framework). Фреймворки – это, примерно то же, что и динамические библиотеки. Они представляют собой скомпилированные объекты, загружаемые в адресное пространство программы во время исполнения, но помимо этого фреймворки включают ресурсы, заголовочные файлы и документацию. Cocoa также включает систему контроля версий, предупреждающую проблемы, встречающиеся в Microsoft Windows (так называемый «DLL hell»).

Ключевой элемент архитектуры Cocoa — это модель представлений (views). Внешне она организована как обычный фреймворк, но реализована с использованием PDF для всех операций рисования, предоставляемых Quartz. Это позволяет программисту рисовать всё, что угодно, используя команды языка, похожего на PostScript. Кроме того, это автоматически предоставляет возможность вывода любого представления на печать. Поскольку Cocoa обрабатывает обрезку, прокрутку, масштабирование и прочие типичные задачи отображения графики, программист освобождается от необходимости реализовывать базовую инфраструктуру и может сконцентрироваться на уникальных аспектах разрабатываемого приложения.

В архитектуре Cocoa строго соблюдены принципы MVC:

Известная как «парадигма модель-представление-поведение» (MVC), эта концепция предусматривает разделение приложения на три набора взаимодействующих между собой классов. Классы модели представляют данные, такие как документы, файлы настроек или объекты в памяти. Представления, как следует из названия, отображают данные (зачастую визуально). Классы поведения содержат логику, связывающую модели с соответствующими представлениями, и обеспечивают их синхронизацию.

2.2 API графических интерфейсов

Большинству интерфейсов популярных ныне операционных систем свойственно интуитивно-понятное графическое оформление с использованием визуальных эффектов, однако так было не всегда. Первые GUI современному пользователю показались бы достаточно примитивными, хотя в смысле функциональности, по тем временам многие были довольно удобными и качественно справлялись со стоящими перед пользователем задачами.

Первым созданным графическим интерфейсом традиционно считают GUI, разработанный в рамках проекта **Xerox Alto** - первого персонального компьютера, который был создан в *1973* году. Графическая оболочка *Xerox Alto* была очень проста, но в ней уже тогда имелись меню, кнопки и примитивные окна. Присутствовал также курсор мыши, с присущими ему функциями выделения, копирования и вставки (Рис. 3).

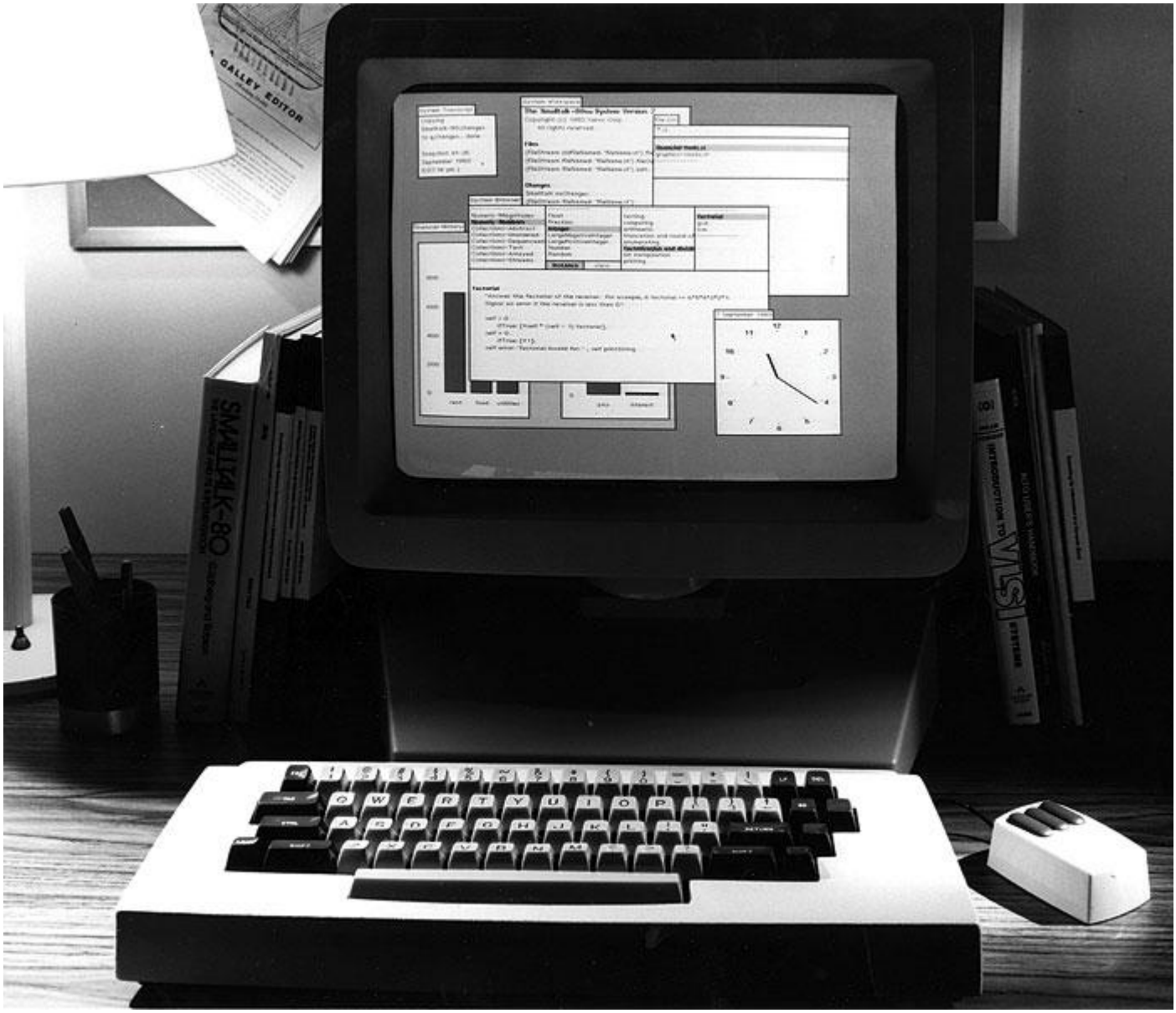

Рисунок 3. Xerox – первый настоящий GUI

В 1981 году появляется новая система под названием **Xerox Star**, основанная на той же Xerox Alto, но с более совершенным функционалом и графическим интерфейсом. Это удивительно, но рабочий стол Xerox Star мало чем отличался от нынешних десктопов, если, конечно, не брать в расчёт визуальные эффекты.

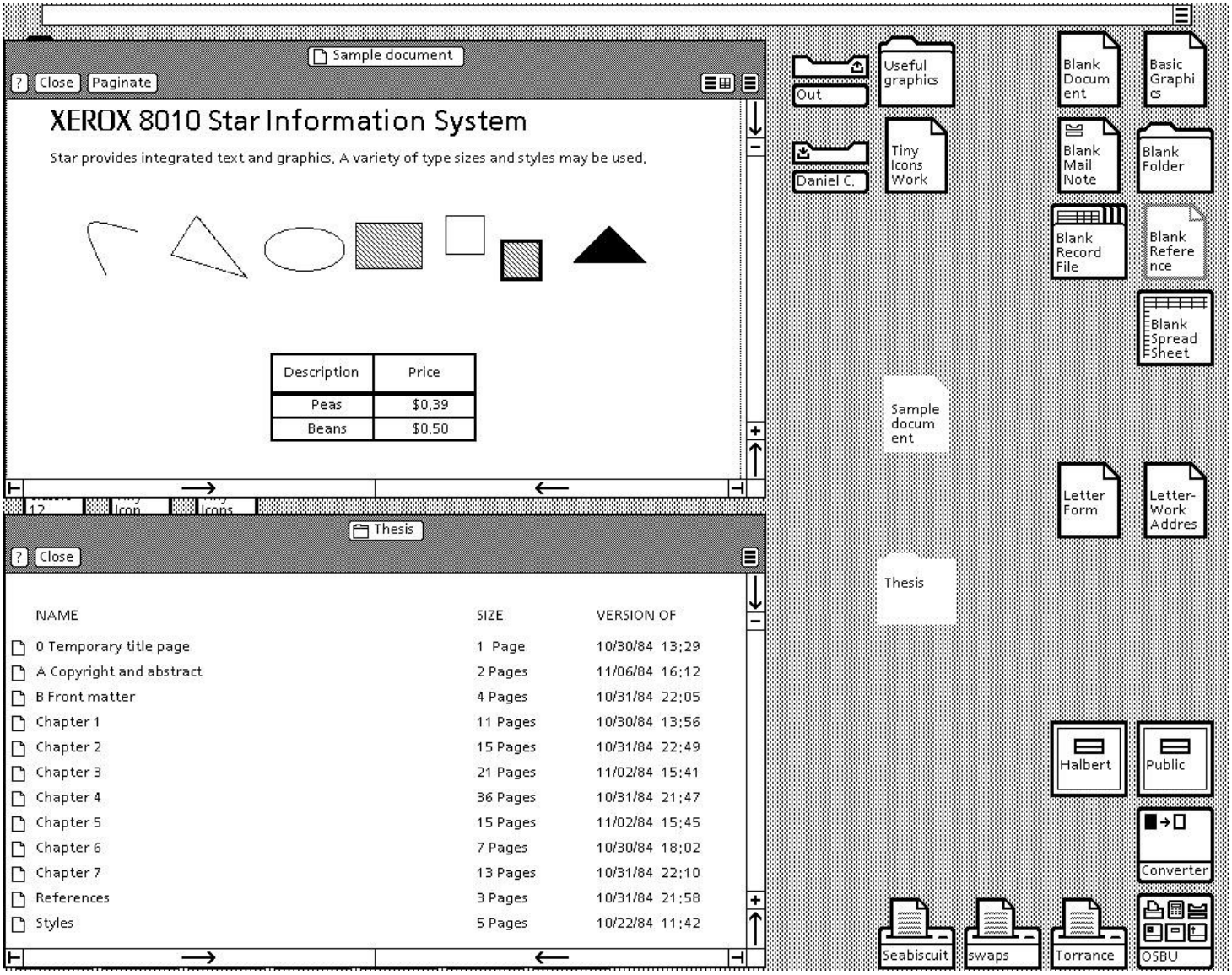

Рисунок 4. Графический интерфейс Xerox Star

В его основе лежит тот же принцип использования ярлыков для запуска файлов и перехода по каталогам файловой системы.

Также в начале 80-х годов свои разработки миру представили компании **Apple** и **Microsoft**. Понимая всё значение GUI, но, не имея достаточно времени для создания оригинальных оболочек для своих систем, разработчики обеих компаний позаимствовали идеи Xerox Lab, что впоследствии даже привело к конфликту между Стивом Джобсом и Биллом Гейтсом. Джобс обвинил Гейтса в плагиате, что тот, якобы, скопировал интерфейс с Macintosh.

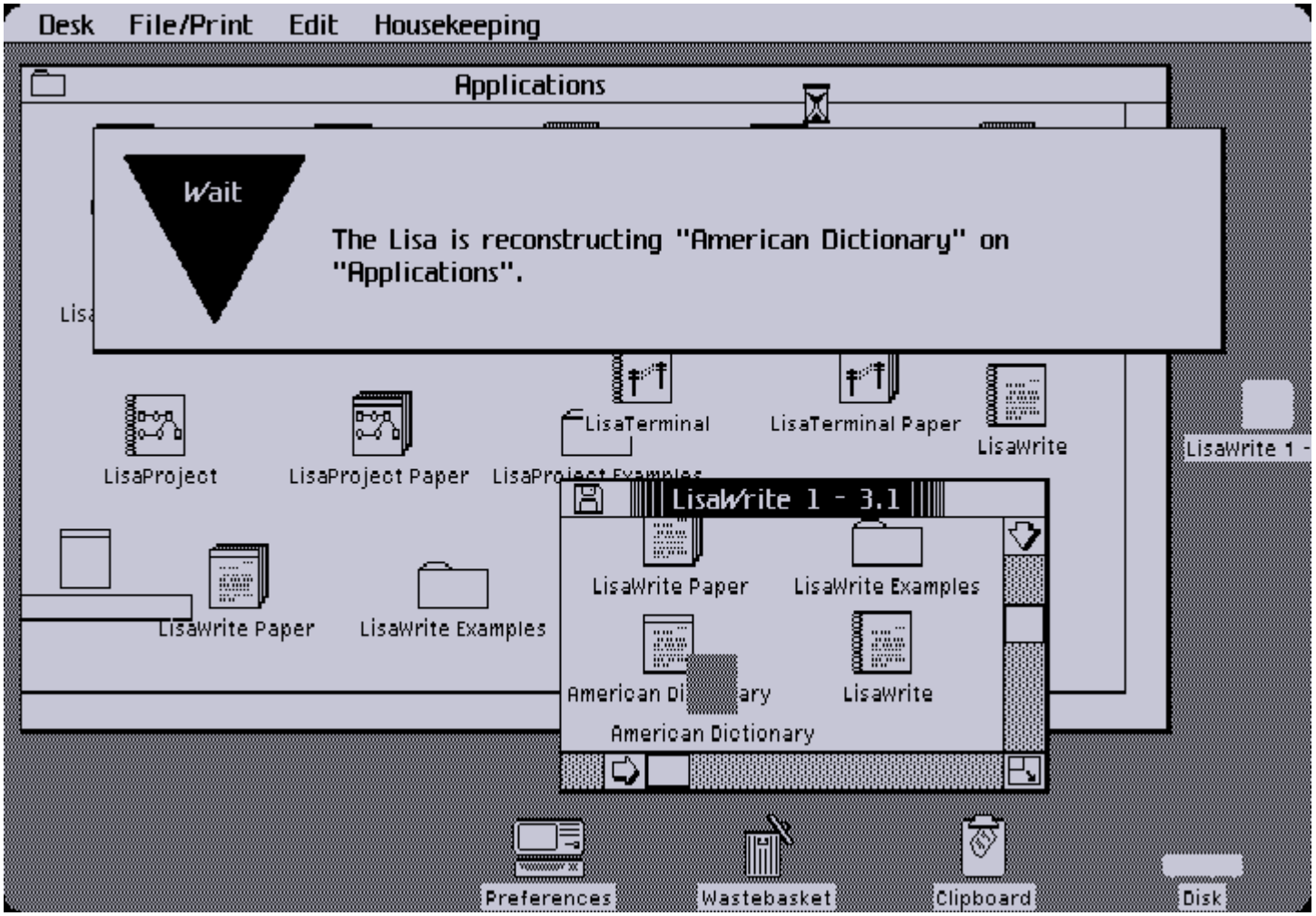

Рисунок 5. GUI Macintosh

Вообще Стив не был прав, так как и он сам, и обвиняемый им Гейтс взяли концепцию GUI у Xerox Lab, просто так получилось, что Джобс оказался первым.

В 1986 году программист Джон Соча создал оригинальный файловый менеджер для MS-DOS, которая до этого не имела практически никакого графического оформления - **Norton Commander**. Роль окон в нем играют панели, которые делят экран по вертикали и содержат списки папок и файлов. В верхней и нижней части менеджера располагаются текстовые меню, позволяющие выполнять различные операции.

Рисунок 6. Norton Commander

В 1988 году вышел его аналог - **DOS Shell**, он также относится к псевдографическим интерфейсам, имитирующим графику, оставаясь на самом деле текстовым интерфейсом.

Оба эти приложения заметно облегчили работу с данными, избавив пользователей от необходимости вводить DOS-команды. Долгое время эти программы были очень популярны, используются они иногда и сейчас.

Выйдя из команды разработчиков Apple Lisa, в 1982 году Стив Джобс возглавил собственный проект **Macintosh**. Разработанная для маков система получила название **Mac OS**. Только в седьмой версии появилось нововведение – поддержка цветов, до этого момента интерфейс системы был практически монохромный.

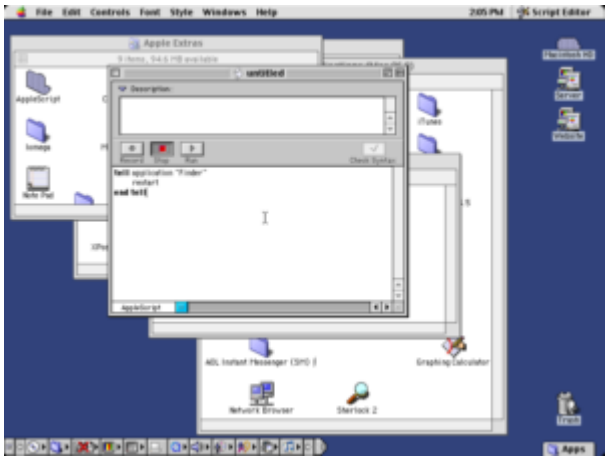

Рисунок 7. GUI MAC OS 9.2.2

На сегодняшний день актуальна версия MAC OS 2000 года.

Microsoft также не сидели сложа руки, и в *1985* году представили свою графическую оболочку для MS-DOS с говорящим названием **Windows**. Оболочка частично поддерживала цветную графику, в ней имелись *32x32*-пиксельные иконки, простые меню и диалоги. Фиксированной области, в которой бы отображались значки запущенных приложений, пока не было, располагаться они могли в любом месте экрана, перекрываясь при этом открытыми окнами.

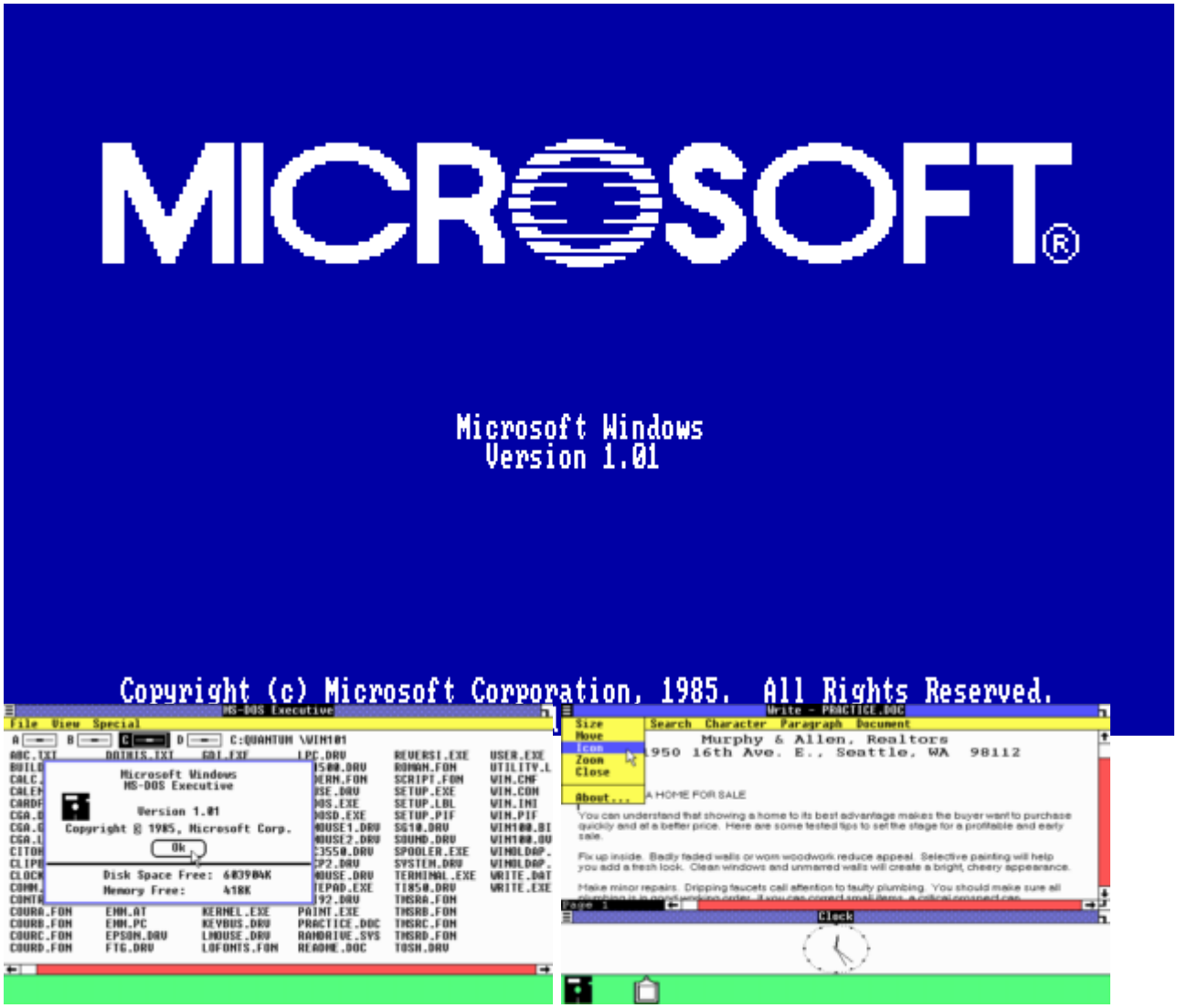

Рисунок 8. Первая Windows

Сами окна в первой версии были достаточно примитивными. Их можно было перетаскивать мышкой, изменять их размер, но при этом сами они не могли

перекрывать друг друга. Сворачивать их также было нельзя. В смысле удобства первая версия Windows значительно уступала системам от Apple.

Версии Windows 1.0, 2.0 *и* 3.0 не были операционными системами в том смысле слова, в котором его принято понимать сегодня. Это были скорее графические оболочки MS-DOS.

В Windows 3.11 уже имелась полная поддержка цветов, окна могли перекрывать друг друга, их можно было сворачивать и разворачивать. Незначительно улучшилась графика отдельных элементов (объемные кнопки и полосы прокрутки), использовались пропорциональные шрифты, внешний вид программ File Manager и Program Manager реализуется в стиле самой оболочки. Цвета элементов интерфейса пользователь мог изменять по своему усмотрению.

Настоящая революция в оформлении Windows свершилась в 1995 году, именно тогда в системе появляются хорошо ныне всем знакомые кнопки Пуск**,** Проводник**,** Панель задач и рабочий стол со значками, который в тоже время являлся отдельной папкой. В этой же версии был реализован показ дисков в папке «Мой компьютер»**,** и способ управления файлами из меню, вызываемого правой кнопкой мыши. Важным нововведением стал переход на 32-битную архитектуру.

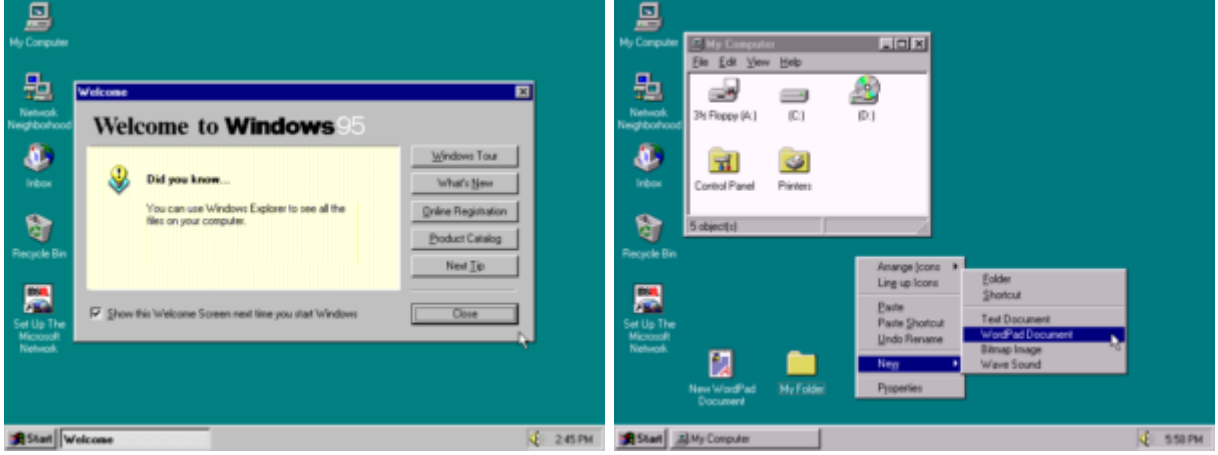

Рисунок 9. Пользовательский интерфейс Windows 95

Рассматривая интерфейсы современных операционных систем, нельзя не заметить явные сходства с графическими оболочками первых GUI Xerox Alto и Apple Lisa. Является ли это следствием недостатка воображения дизайнеров? Скорее всего, причина сходства – потребности физиологии пользователей, которые особо не изменились с тех времён.

Хотя первые графические интерфейсы и были примитивны, но была в них и какая-то подкупающая простота, которой так иногда не хватает перегруженным визуальными эффектами оболочкам современных программ и операционных систем.

Таким образом, можно сделать вывод, что программные интерфейсы в ходе своей эволюции стали больше удовлетворять потребностям пользователей.

3. Web API

Эти стандарты, которые называются протоколы веб-сервисов, представляют собой наборы методов, определяющих способ передачи данных и доступа к API. Самые популярные протоколы, REST и SOAP, сейчас лидируют в этой гонке и используются подавляющим большинством открытых API-интерфейсов.

SOAP (Simple Object Access Protocol) до недавнего времени считался безусловным фаворитом у разработчиков API. Но сейчас 70% открытых API соответствуют протоколу REST. SOAP по-прежнему используется во многих крупных технических компаниях и обеспечивает поддержку устаревших систем, которые могут быть совместимы только с ним.

REST (Representational State Transfer) — это новый протокол веб-сервисов, позволяющий работать с большим количеством форматов данных. Кроме того, REST предпочтительнее для разработчиков, так как предлагает меньшее время загрузки и более высокую эффективность.

Если провести сравнительный анализ этих двух подходов, можно выделить в них отличия и сходства:

1. SOAP – это целое семейство протоколов и стандартов, следовательно это более тяжеловесный и сложный вариант с точки зрения машинной обработки. Поэтому REST работает быстрее.
2. SOAP используют HTTP как транспортный протокол, в то время как REST базируется на нем. Это означает, что все существующие наработки на базе протокола HTTP, такие как кеширование на уровне сервера, масштабирование, продолжают так же работать в REST архитектуре, а для SOAP необходимо искать другие средства. Взамен этого SOAP сервисы получают такое мифическое свойство, как возможность работать с любым протоколом транспортного уровня вместо HTTP, однако практической пользы от этого нет.
3. REST может быть представлен в различных форматах (и XML и JSON), а SOAP привязан к XML. Это важно, особенно если представить себе вызов сервиса из javascript, ответ на который мы, естественно, хотим получать в JSON.
4. REST это простота, а SOAP делает упор на стандарты.
5. SOAP работает с операциями, а REST – с ресурсами. Этот факт в совокупности с отсутствием клиентского состояния у RESTful сервисов приводит к тому, что например транзакции или другая сложная логика должна реализовываться на SOAP.

Заключение

В данном исследовании изучены прикладные программные интерфейсы, их основные особенности, правила построения, а также рассмотрена эволюция прикладных программных интерфейсов. Проведена классификация основных существующих на данный момент стандартизированных программных интерфейсов.

Сделан вывод, что современному разработчику программного обеспечения необходимо включать в создаваемый продукт прикладные программные интерфейсы, для создания качественного программного обеспечения, которое

будет успешно решать поставленные задачи, иметь возможность взаимодействовать с другими программами, при этом имея возможность модификации и усовершенствования.

Цель работы достигнута.

Список используемой литературы

1. Гордеев А.В., Молчанов, А.Ю. Системное программное обеспечение: Учебник / А.В. Гордеев, А.Ю. Молчанов. - СПб.: Питер, 2008. – 458 с.

2. Иванова Г.С. Объектно-ориентированное программирование: Учебник. / Г.С. Иванова. – М.: Изд-во МГТУ им. Баумана, 2003. – 416 с.

3. Трубачёва С.И. Программирование в операционных системах: Учебно-мет.пособие. / С.И. Трубачёва -Тольятти: Изд-во ВУиТ, 2006. – 396 с.

4. Жданова Т.А., Бузыкова Ю.С. Основы алгоритмизации и программирования: учеб. пособие / Т.А. Жданова, Ю.С. Бузыкова. – Хабаровск : Изд-во Тихоокеан. гос.ун-та, 2011. – 56 с.

5. Аузяк А.Г, Богомолов Ю.А. Программирование и основы алгоритмизации: Для инженерных специальностей технических университетов и вузов. / А.Г. Аузяк, Ю.А. Богомолов, Казань: Изд-во Казанского национального исследовательского технического ун-та - КАИ, 2013. - 153 с.

6. Основы алгоритмизации и программирования: учебное пособие / Г. Р. Кадырова. – Ульяновск : УлГТУ, 2014. – 95 с.

7. Белов П.М. Основы алгоритмизации в информационных системах: Учебн. Пособие. - Спб.: СЗТУ, 2003. – 85с.

8. Макаров В.Л. Программирование и основы алгоритмизации.: учебн. пособие. / В.Л. Макаров - Спб., СЗТУ, 2003. - 110с.

9. Неббет Г. Справочник по базовым функциям API Windows NT/2000. - / Г. Неббет. - М.: Вильямс, 2002. – 528 с.