Олькина Елена Викторовна. Проектирование человеко-машинного интерфейса. Экзамен. Лек 16, пр 3, лаб 8. Гонеев – Проектирования интерфейса пользователя средствами 32api.

**Проектирование Windows-приложений. Структура Windows-приложений**

Под трансляцией приложения понимается процесс преобразования исходного кода программы на любом ЯП высокого уровня в исполняемый код, который процессор воспринимает как инструкцию к исполнению. В общем виде процесс трансляции в Windows выглядит следующим образом:

1. Исходный код программы преобразуется в объектный код, который необходим для организации ссылочной целостности приложения.
2. Линковщик на основе объектного файла и заголовочных инструкций организует схему сборки приложения, подаваемое компилятором.
3. Компилятор в завершающем этапе – сборке приложения, собирает объектный файл, заголовочные модули и при необходимости функции и библиотеки в единый исполняемый файл.

Windows различает следующие виды приложений, в зависимости от используемого типа окон и процедур обработки сообщений:

1. Стандартное приложение Windows. Данное приложение использует обычное окно-приложение, одно из которых является главным (если их несколько), для него организуется стандартный цикл обработки сообщений и стандартный механизм взаимодействия с ОС.
2. MDI-приложения. Это приложения, основой функционирования которых является одновременная обработка нескольких окон, для этого используется механизм синхронизации обработки нескольких очередей сообщений (для главного окна – отдельная очередь сообщений, для зависимых окон может быть по одному, а может быть и нет).
3. Приложения помощи (хелперы). Особенность данного приложения в том, что всю обработку сообщений от этого окна и организации логики диалога берёт на себя ОС. Как правило, данный инструментарий в Windows изменить нельзя.
4. Диалоговые приложения. Данный вид приложений отличается от первого тем, что помимо стандартных средств функционирования приложения, в него встраивается инструмент ведения диалога. Данный инструментарий напрямую связан с модулем логики диалога ОС и разрабатывается в соответствии со стандартами проектирования графического пользовательского интерфейса.

Если рассматривать приложение с точки зрения разработчика, то приложение состоит из окна и кода, который обрабатывает сообщения от него, при этом элементы управления, располагающиеся в окне, могут генерировать сообщения о произошедшем событии, и реакцию приложения пользователь видит в том же окне. С точки зрения ЯП и ОС Windows, окна – это объекты, над которыми выполняются действия. Объект принадлежит определённому классу окна, который описывает множество данных – параметров состояния окна и функцию изменения этих параметров. Из этого можно сделать вывод, что ОС воспринимает не только окна, как окна, но и все элементы, располагающиеся в окне также, как окна.

Стандартное Windows-приложение имеет 2 обязательных элемента в своей структуре:

1. Оконная функция приложения. В стандартном графическом приложении Windows – WinMain. Назначение – как минимум создание главного окна приложения, описание и регистрация класса окна, если это необходимо и организация цикла обработки сообщений.
2. Функция окна. Назначение – организация реакции на действия пользователя при возникающих событиях. Главная особенность – корректное завершение работы Windows-приложения. WndProc. Оконная функция имеет зарезервированное имя, которое воспринимается компилятором как стартовая функция приложения, а имя функции окна может быть отличным от стандартного имени.

Особенности функции окна:

1. Данная функция имеет список формальных параметров, определённых в ОС и стандартный тип возврата.
2. Данная функция называется функции обратного вызова в связи с тем, что только ОС может её вызывать.
3. Сообщения, обработчик которых не реализован в функции окна приложения, передаётся ОС автоматически для стандартной обработки.

Алгоритм функционирования Windows-приложения:

1. Компилятор ищет в коде стартовую функцию приложения (winMain). Данная функция выполняется полностью и безоговорочно, единственный момент, когда функция может прервать своё выполнение, когда после создания цикла обработки сообщений пользователь пытается сделать ещё что-то (функция завершается аварийно). Данная функция отрабатывается ОС единожды и в процессе функционирования приложения больше никогда к ней не обращается. Главным выходом этой функции должен быть идентификатор главного окна приложения, которым будет оперировать функция окна, и запущенный цикл обработки сообщений. Оконная функция приложения обеспечивает взаимодействие приложения Windows с ОС.
2. После того, как главное окно приложения создано, оно способно реагировать на действия пользователя, что обеспечивает взаимодействие пользователя и приложения (wndProc). Пользователь, выполнив какое-либо действие, генерирует входящий сигнал в модуль логики диалога ОС. ОС регистрирует произошедшее событие, присвоив код данному событию. В соответствии с присвоенным кодом сообщение помещается в ту очередь сообщений, откуда оно может быть выбрано для данного приложения, после этого ОС вызывает функцию обратного вызова.
3. Функция окна выбирает сообщение из очереди и в соответствии со своим кодом пытается найти его обработчик. Если успешно – обработчик вызывается, сообщение обрабатывается, и функция окна передаёт управление ОС. Если обработчика нет – вызывается функция DefWindowsProc, которая сначала передаёт необработанное сообщение ОС, потом передаёт управление ОС, ОС обрабатывает поступившее сообщение стандартным образом. Данный цикл вызова функции окна повторяется до тех пор, пока пользователь работает с приложением и генерирует сообщения.
4. Если пользователь завершает работу приложений, генерируется сообщение WM\_DESTROY. Данное сообщение вызывает функцию PostQuitMessage(0). Данная функция помещает в очередь обработки сообщений 0, который, при выборе его в цикле обработки сообщений, завершает данный цикл. Как только ОС регистрирует завершившийся цикл обработки сообщений и понимает, что рабочих потоков больше нет, она предпринимает действия по завершению приложения и разрушению виртуального адресного пространства.

**Ошибки в Windows-приложениях и обработка ошибок**

Прежде чем изучать функции Windows необходимо понять специфику их работы в целом, а также какие они могут допускать ошибки и как они обрабатываются. В общем когда приложение вызывает любую функцию Windows, она проверяет передаваемые ей параметры, а затем пытается выполнить свою работу. Если функции был передан недопустимый параметр или если данную функцию нельзя выполнить по какой-либо другой причине, она возвращает значение, свидетельствующее об ошибке. В документации Microsoft определены типы данных для возвращаемых значений большинства функций Windows и значения, которые свидетельствуют об ошибке.

|  |  |
| --- | --- |
| Тип функции | Значение, говорящее об ошибке |
| VOID | Такие функции всегда (или почти всегда) выполняются успешно |
| BOOL | Если вызов функции заканчивается неудачно, возвращается 0 (цифра 0), в остальных случаях возвращаемое значение отлично от нуля |
| HANDLE | Если функция вызывается неудачно, то обычно возвращается NULL, в остальных случаях HANDLE идентифицирует объект, которым может манипулировать приложение. Следует с осторожностью относится к ситуации, когда некоторые функции HANDLE возвращают значение -1 (это значение используется в 64-разрядных Windows и конкретное сообщение об ошибке определяется в документации Microsoft |
| PVOID | При неудаче – NULL, в остальных случаях – адрес блока данных памяти |
| LONG/DWORD | Если по какой-то причине функция не выполнилась удачно (или выполнила не то, что требовалось) обычно она возвращает или 0, или -1. Если функция по данным Windows выполняется удачно, но результат выполнения отличный от ожидаемого, то принято называть такие функции условно ошибочными |

При возникновении любой ошибки необходимо разобраться, почему вызов функции оказался неудачен. Для этого в Windows за каждой ошибкой закреплён свой код – 32х-битное число. Функция Windows, обнаружив ошибку через механизм локальной памяти поток, сопоставляет соответствующий код ошибки с вызывающим её потоком. Данный механизм позволяет работать потокам независимо друг от друга, не вмешиваясь в чужие ошибки и не перезаписывая их значение. Для того, чтобы узнать, какая произошла ошибка, используется функция GetLastError(). Все ошибки Windows прописаны в файле winerror.h в следующем формате:

1. Идентификатор ошибки.
2. Текст сообщения ошибки на английском языке.
3. Номер ошибки в соответствии с документацией Microsoft.
4. Текст ошибки на национальном языке Windows.

При этом следует помнить, что функцию GetLastError нужно вызывать сразу же после неудачного вызова функции Windows, иначе код ошибки может быть потерян. Другими словами, механизм работы функции GetLastError организован таким образом, что он обращается к ядру процесса, в котором определён только один идентификатор для ошибки.

Некоторые функции Windows всегда завершаются успешно, но по разным причинам. Например, если была попытка создания окна с определённым идентификатором, то попытка может быть успешна либо потому что оно действительно было создано, либо потому что объект с таким идентификатором уже существует, в этом случает возвращаемое значение функции будет свидетельствовать об ошибке, хотя окно рабочее, и генерируется 32х-битный код ошибки.

Помимо функции GetLastError() возможно использовать функцию FormatMessage(), которая по коду ошибки из файла winerror.h вытаскивает её текстовое описание.

Структура 32х-битного кода:

|  |  |  |
| --- | --- | --- |
| Биты | Содержимое | Значения |
| 31-30 | Код тяжести ошибки | Функция вернула 0 – успех, 1 – информация, 2 – предупреждение, 3 - ошибка |
| 29 | Указывается, ем определён бит: Microsoft или пользователем | 0 – Microsoft, 1 - пользователем |
| 28 | Зарезервирован | Всегда 0 |
| 27-16 | Код подсистемы | Значение определяется Microsoft |
| 15-0 | Код исключения | Определяется Microsoft или пользователем |
|  |  |  |

**Создание библиотек в Windows**

Одной из основ функционирования Windows являются динамически подключаемые библиотеки, в которых реализована как функциональная часть ОС, так и специализированные функции, к примеру элементы оконного интерфейса, драйвера, утилиты и т.д. Если рассматривать конкретный графический пользовательский интерфейс, в частности модуль логики диалоги ОС, то можно выделить основополагающие системы библиотеки Windows, без которых работа этого модуля невозможна:

1. Kernel32.dll – содержит функции управления памятью, процессами и потоками в ОС.
2. User32.dll – поддержка пользовательского интерфейса и функции, связанные с созданием окон и обработкой от них сообщений.
3. GDI32.dll – работа с текстом в графических приложениях Windows, а также поддержка графики и графических библиотек Windows.

Все остальные системны библиотеки Windows предназначены для поддержки специализированных свойств и событий и обязательно требуют для своей работы наличие трёх главных библиотек. В Window 3.11 были попытки создания первых библиотек. В Windows 95 была попытка создать первый прообраз логики диалога. Предпосылки для масштабного использования библиотек в ОС Windows:

1. Расширение функциональности приложений. То есть по требованию приложения dll может загружаться в адресное пространство процесса, дополняя собственный код приложения отдельными функциями, расширяя тем самым его функциональность. При этом учитывалась ещё одна особенность: одна и та же библиотека могла быть использована многими приложениями, причём не только библиотеки Windows, а также сторонние библиотеки.
2. Использование библиотек позволило использовать разные ЯП для их реализации.
3. Более простое управление проектом в модульном программировании.
4. Экономия памяти.
5. Разделение ресурсов.
6. Упрощение локализации.
7. Решение проблем, связанных с особенностью различных платформ.
8. Реализация специфических возможностей.

Зачастую создать dll проще, чем приложение, поскольку она является лишь набором автономных функций, пригодных для использования любыми Windows-приложениями. Связано это с тем, что библиотека не могут содержать цикл обработки сообщений или генерировать и обрабатывать сообщения приложений Windows. В связи с тем, что данные элементы отсутствуют в библиотеках, они могут быть использованы в любых приложениях, где требуется их код. Файлы с исходным кодом библиотек компилируются и компонуются точно также, как и файлы приложений, за исключением того, что компоновщику необходимо указать, что необходимо собрать не приложение, а библиотеку. Чтобы приложение или другая библиотека могла вызвать функции, содержащиеся в dll, образ её файла необходимо сначала спроецировать на адресное пространство вызывающего процесса (АПП), тогда её функции будут доступны всем потокам этого процесса. После загрузки библиотеки, они теряют свою индивидуальность. Для исполняющихся потоков в процессе код и данные библиотеки – это просто дополнительный код и данные, оказавшиеся в адресном пространстве процесса.

Способы загрузки библиотек в адресное пространство:

1. Неявное связывание.

При неявном связывании библиотеки и адресного пространств загрузка библиотеки осуществляется в момент вызова любой из функции библиотеки, при этом в адресное пространство загружается только вызываемый код (не вся библиотека!). Для того, чтобы данный процесс был возможен, при загрузке приложения оно должно правильно прописать ссылки на подключаемые библиотеки:

uses библиотека/путь к ней;

…

вызов функции;

end.

Данный способ называется связывание при загрузке.

1. Явное связывание (во время выполнения приложения) Осуществляется в коде самого приложения: LoadLibrary (”имя библиотеки (путь)”). Функции этой библиотеки доступны приложению только после вызова функции LoadLibrary. Ещё одна особенность этого метода – в АПП загружается весь код библиотеки, каким бы большим он не был.

Выгрузка библиотек Windows из АПП осуществляется при вывозе функции FreeLibrary и фактически представляет собой уничтожение ссылок на подключённые библиотеки, после чего доступ к функциям библиотеки не может быть осуществлён.

Этапы создания dll:

1. Необходимо подготовить заголовочный файл с прототипами функций, структурами и идентификаторами, экспортируемыми из библиотеки. Этот файл включается в исходный код всех модулей создаваемой библиотеки. Также такой файл используется и при сборке исполняемого модуля.
2. Реализуется исходный код библиотеки (библиотек) на выбранном ЯП, учитывая при этом концепцию входа и выхода библиотеки. Так как данный код кроме как для компиляции не нужен, т.е. он не участвует в сборке исполняемого модуля библиотеки, этот код может остаться коммерческой тайной разработчика. Код должен содержать реализацию объявленных функций в заголовочном файле, определения структур и идентификаторов, объявленных там же, а также при необходимости указание экспортируемых и импортируемых элементов из других библиотек или исполняемых модулей.
3. Компилятор преобразует исходный код модулей библиотеки в объектные файлы. На каждый модуль библиотеки свой объектный файл.
4. После компиляции работу начинает компоновщик, который собирает все объектные файлы с привлечением заголовочных файлов в единый загрузочный dll-модуль, в который в конечном итоге помещается двоичный код и переменные (как глобальные, так и статические), относящиеся к коду данной библиотеки. Создаваемый на данном этапе файл потребуется при компиляции исполняемого модуля библиотеки.
5. Если компоновщик обнаружит, что библиотека экспортирует хотя бы одну переменную или функцию из другой библиотеки, то он создаст ещё и lib-файл. Данный файл совсем крошечный, поскольку в нём нет ничего, кроме списка символьных имён функций и переменных, экспортируемых из сторонней библиотеки. Данный файл также используется при компиляции конечного файла библиотеки. Только после данного этапа можно приступить к сборке исполняемого модуля библиотеки.
6. Во все модули исходного кода, где есть ссылки на внешние функции и переменные, структуры данных и идентификаторы, необходимо включить заголовочный файл, созданный на 1 этапе. Включение данного файла позволит правильно собрать все созданные промежуточные файлы: объектный код, созданный на основе исходного кода, сам заголовочный файл и lib-файл, если таковой имеется. Далее компилятор генерирует исполняемый файл библиотеки, который может использоваться уже другими библиотеками и исполняемыми файлами.

**Самостоятельно:** *Модификаторы, которые могут быть использованы в исходном коде библиотеки: dllexport, dllimport. Для каждого модификатора посмотреть их заменяемый аналог: dllexport 🡪 extern и для dllimport. Примеры, где dll для обработки сообщений – не разбирать. + Работа с графическим контекстом приложений (DC) + Создание многооконных документов с помощью MDI (Румянцев).*

**Проектирование графического пользовательского интерфейса (ГПИ)**

При создании и реализации различных приложений разработчики создают некие экранные формы для организации диалога между пользователем и приложением. Так как сложность приложений на сегодняшний момент сильно возросла (т.е. приложения могут содержать не одно окно, множество управляющих элементов, расположенных в этих окнах, а также модули, которые обрабатывают данные окна) встаёт вопрос об организации взаимодействия и корректной передачи данных между различными окнами и их пользователями. В теории проектирования ГПИ вводят понятия взаимосвязи различных окон, при этом она рассматривается как сложный граф со всеми усложняющими элементами: петлями, циклами и т.д., управление которым и возлагается на разработчика в процессе реализации кода приложения. Помимо данного аспекта теория ГПИ включает в себя управление доступностью различных элементов на разных этапах работы приложения, управление безопасностью выполнения каких-либо действий и вычислений при функционировании самого приложения, а также защиту приложения от непродуктивных действий пользователя.

Выявление и правильная обработка всех взаимосвязей между элементами в приложении производится в рамках проектирования пользовательского интерфейса и называется разработкой логики диалога с пользователем. При реализации многооконных приложений перед разработчиком всегда встаёт вопрос: каким образом предусмотреть все возможные варианты действия пользователя в том или ином окне и как изменения, которые делает пользователь, должны отражаться на других окнах и управляющих элементах? То есть взаимосвязи рассматриваются не только внутри одного окна между его управляющими элементами, но и между управляющими элементами нескольких окон, если такие взаимосвязи предполагаются.

Отдельной задачей разработки логики диалога с пользователем стоит задача взаимного вызова различных окон. Как правило, разработчиком эта задача решается в ходе написания основного кода приложения, при этом команды управления диалогом перемешиваются с командами обработки данных. Такая политика создаёт значительные трудности модификации такого приложения, что делает управление интерфейсом достаточно трудоёмким процессом. Именно в решении данных задач возникает теория ГПИ, которая предполагает проектирование интерфейса отдельно от проектирования кода приложений и реализация интерфейса отдельно от реализации кода приложений.

**Структура пользовательского интерфейса в ОС Windows**

1

5

4

2

7

6

2

3

3

1

8

1. Прикладной интерфейс
2. Модуль логики диалога ОС
3. Окна приложения с управляющими элементами

Стрелки:

1. События
2. Контекст
3. Сигналы
4. Обратная семантическая связь
5. Команды
6. Прикладные данные
7. Запрос
8. Экран

При проектировании диалога с пользователем определяется состояние приложения или системы как устойчивое положение приложения, в котором не происходит никаких вычислений и выполнения алгоритмов, в котором пользователь может выполнять различные действия (как действия, которые могут повлечь изменения состояния системы и изменение интерфейса в частности и действия, которые не взывают изменения приложения).

Управление приложением для пользователя заключается во взаимодействии с окнами приложения. Пользователь взаимодействует с окнами, порождая события. Окна или экранные формы преобразуют событие во входные сигналы для модуля логики диалога ОС. Не любое событие будет являться входным сигналом, а то, которое повлечёт изменение состояния приложения. При обработки входных сигналов в модули логике диалога изменяется контекст приложения (стрелка 2), т.е. набор окон и управляющих элементов, доступных пользователю в конкретном состоянии приложения. Изменение контекста всегда отображается на экране.

**Прикладной интерфейс** – это специализированные программы для обращения к БД или алгоритмов обработки, не принадлежащих приложению. Прикладной интерфейс начинает работать в том случае, если получает команду от модуля логики диалога, в ответ на команду интерфейс может генерировать обратный сигнал (обратную семантическую связь). Он также может изменять контекст интерфейса приложения. Окна и управляющие элементы могут взаимодействовать с интерфейсом минуя модуль логики диалога только в том случае, если такое взаимодействие не изменяет состояние приложения.

Приложения Windows характеризуются ещё и тем, что одновременно на экране может быть несколько окон, которые не только взаимодействуют между собой, но и каждое из которых имеет свой внутренний диалог, т.е. изменение одних управляющих элементов влияет на состояние других в одном и том же окне. В другом случае иногда одно и то же действие пользователя может приводить к различным входным сигналам и соответственно к разному изменению контекста приложения. В теории проектирования интерфейса такой диалог называется **конкурентным**. В этом случае модуль логики диалога содержит диалоги не только всех окон, но и отвечает за взаимодействие различных элементов этих окон.

**Проектирование логики диалога с использованием простой транзитивной сети**

Использование транхитивных сетей для проектирования логики диалога является самым простым из способов организации диалога пользователя с приложениями, т.к. данный способ является:

1. Графическим, что добавляет удобство чтения диалога.
2. Используется аппарат конечного автомата, который жёстко детерминирован.

В этом случае логика диалога отображается в виде направленного графа, у которого вершины представляют состояния системы, дуги – переходы между этими состояниями. Также вводится правило именования дуг для транзитивных сетей:

1. Сверху дуга именуется согласно входным сигналам или обратной семантической связи прикладного интерфейса (либо то, что делает пользователь приложений, либо то, что делает ОС или другие приложения, которые могут влиять на работу логики диалога (**может не отображаться**, если действие системы или приложения не вызвано действием пользователя или ОС – дуга подписывается только снизу).
2. Снизу дуга именуется согласно действиям, которое должно выполнить приложение при переходе из одного состояния в другое в ответ на действие пользователя или ОС.
3. Имена состояний должны отображать статику приложения в конкретном состоянии.

**Пример отображения и построения простой транзитивной сети.**

Реализуется приложение, которое работает с БД в двух режимах:

1. В режиме таблицы.
2. В режиме запроса.

Окно приложения будет выглядеть:

5

4

3

1

4

2

8

6

7

1. Выпадающий список, где выбираем, в каком режиме будет работать приложение.
2. Текстовое поле для ввода имени таблицы БД.
3. Listbox (развёрнутый многострочный список), в котором отображается список полей выбранной таблицы.
4. Кнопки “OK” и “Далее”.
5. Многострочное текстовое поле, в котором отображается текст запроса на SQL к БД.
6. Список, предполагающий выбор операции для запроса SQL.
7. Текстовое поле для ввода значения, которое будет использоваться в запросе SQL.
8. Окно вывода.

При работе приложения в 1 режиме необходимо выбрать режим в выпадающем списке, ввести имя таблицы БД для определения источника данных для приложения. При нажатии кнопки “ОК” происходит попытка связи с физическими данными, если успешно – то данные отображаются в 8.

Во 2 режиме последовательность действий следующая. При помощи 2 и 3 с именами полей таблицы формируется запрос на SQL, который отображается в 5. При вводе имени таблицы и нажатии кнопки “ОК” заполняется Listbox 3 с полями таблицы, при этом изменить имя таблицы уже нельзя. Далее из списка полей двойным щелчком выбираем те поля, которые будут отображены в разделе SELECT SQL-запроса. Если в данный момент нажимает кнопку “ОК”, происходит выполнение запроса и отображения данных в 8. Если нажимаем кнопку “Далее”, переходим к добавлению раздела WHERE в SQL-запрос. При простом нажатии на поле 3 выбираем действие оператора сравнения 6 и значение 7, мы добавляем условие в раздел WHERE. Добавление происходит также двумя кнопками: если нажимаем “ОК”, то формирование условия заканчивается и SQL-запрос выполняется, если нажимаем кнопку “Далее”, то формируем сложное условие (AND, OR), в котором также задействуются поля 3, 6, 7.

Состояния:

1. Выбор режима.
2. Выбор таблицы.
3. Связь с физическими данными.
4. Сообщение об ошибки.
5. Просмотр данных.
6. Выбор таблицы для запроса.
7. Отображение полей выбранной таблицы.
8. Просмотр данных.
9. Выбор поля для формирования условия.
10. Выбор знака для условия.
11. Ввод значения для условия.
12. Формирование раздела WHERE в запросе.
13. Состояние финиш (F).

Стартовое и финишное состояния транзитивной сети имеют особое обозначение – для обозначения стартового состояния используется входящая в него стрелка, финишное состояние отображается в двойном кружке. **Соединение или ветвлении дуг на транзитивной сети недопустимо.**

Дуги:

**//Режим таблицы**

1-2: над – выбор режима работы, под – отображение режима в выпадающем списке.

2-3: над – нажатие “ОК”, под – подключение к БД.

3-4: под – вывод сообщения об ошибке.

4-2: над – нажатие “ОК”, под – сброс режима работы и имени таблицы (очистка полей).

3-5: под – соединение удачно, выбор данных.

5-1: над – нажатие “ОК”, под – очистка всех заполненных элементов.

**//Режим запросов**

1-6: над – выбор режима запросов, под – перевод приложения в режим запроса.

6-7: над – нажатие “Далее”, под – отображение полей таблицы, добавление раздела SELECT в запрос.

7: над – двойной щелчок по списку полей, под – добавление выбранного поля в раздел SELECT.

7-8: над – нажатие “ОК”, под – добавление раздела FROM в запрос, вывод данных в окно приложения.

8-1: над – нажатие “ОК”, под – очистка всех заполненных полей приложения.

7-9: над – нажатие “Далее”, под – добавление раздела FROM и раздела WHERE в запрос.

9-10: над – выбор поля таблицы, нажатие “Далее”, под – добавление поля в раздел WHERE.

10-11: над – выбор знака (оператора сравнения), нажатия кнопки “Далее”, под – добавление оператора в раздел WHERE.

11-12: над – ввод значения для условия, нажатия кнопки “Далее”, под – добавление значения для сравнения в раздел WHERE.

12-8: над – нажатие “ОК”, под - вывод данных по сформированному запросу.

12-9: над – выбор оператора для сложного условия (AND, OR), нажатие “Далее”, под – формирования сложного условия в разделе WHERE.

Состояния “Старт” и “Финиш” на любой транзитивной сети необходимы для указания точек входа и выхода их приложения. Такие вершины содержатся в графе логики диалога всегда, не зависимо от количества и смысла других состояний сети. Для описания всегда используется одно состояние “Старт” и одно состояние “Финиш”. Единственное исключение из этого правила – наличие подсетей транзитивной сети, которые могут как иметь свои стартовые и финишные состояния, так и не иметь. Для пояснения транзитивной сети к ней всегда составляются три таблицы:

1. **Таблица контекста состояний.**

В строках (1) – состояния транзитивной сети, в графах (2) – объекты интерфейса и диалога.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1\2 | Объект меню | Таблица | Список с полями таблицы | Список со значениями | Список с операциями сравнения | Текстовое поле для имени таблицы | Текстовое поле для ввода значения | Кнопка “ОК” | Кнопка “Далее” | Многострочное текстовое поле |
| S | + | - | - |  |  |  |  |  |  |  |
| 1 | + |  |  |  |  |  |  |  |  |  |
| 2 | + |  |  |  |  | + |  | + |  | + |
| 3 |  |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  | + |  | + |  |  |
| 5 | + | +,- |  |  |  |  |  |  |  |  |
| 6 | + |  |  |  |  | + |  |  | + |  |
| 7 |  |  | + |  |  |  |  | + | + | + |
| 8 | + | +,- |  |  |  |  |  |  |  | + |
| 9 |  |  | + |  |  |  |  |  | + | + |
| 10 |  |  |  | + |  |  |  |  | + | + |
| 11 |  |  | + |  |  |  | + |  | + | + |
| 12 |  |  |  |  | + |  |  | + | + | + |
| F |  |  |  |  |  |  |  |  |  |  |

Обозначение: плюс указывает, что элемент интерфейса участвует в диалоге и доступен пользователю для управления, минус указывает на то, что элемент интерфейса необходимо переместить в недоступное состояние, если до этого он участвовал в диалоге (объект необходимо сделать либо недоступным, либо невидимым). Пустая ячейка говорит о том, что элемент интерфейса в данном состоянии не используется и никаких действий по его управлению выполнять не надо.

Простая транзитивная сеть логики диалога реализуется с использованием конечного автомата, который задаётся следующим математическим аппаратом: H = (Q, Sigma, P, delta, gamma, q0, F). Данная семёрка подчиняется следующим правилам:

1. Q – множество состояний диалога приложений.
2. Sigma – множество входных сигналов диалога.
3. P – множество команд или прикладных процедур.
4. delta – функция переходов диалогов, которая подчиняется закону: Sigma x Q -> Q.
5. gamma – функция прикладных действий, отражающая следующую зависимость: Sigma x Q -> P.
6. q0 – начальное состояние диалога, для которого всегда верно q0 принадлежит Q.
7. F – множество конечных состояний (если их несколько в случае сложных рекурсивных сетей), для которого верна следующая математическая зависимость: F входит в Q.
8. **Таблица состояний.**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sigma\Q | S | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Меню таблицы |  | Очистить поле для ввода имени таблицы |  |  |  |  |  |  |  |  |  |  |  |
| Кнопка “ОК” |  |  | Связь с физическими данными |  | Очистка поля |  |  | + FROM в запрос |  |  |  |  | Выполнение запроса |
| Сообщение об ошибке | … | … | … |  | … |  | … | … |  | … | … | … | … |
| Связь с таблицей | … | … | … |  | … |  | … | … |  | … | … | … | … |
| Режим “запрос” | … | … | …. |  | … |  | … | … |  | … | … | … | … |
| Кнопка “Далее” |  |  |  |  |  |  | + SLECT в запрос | + FROM и WHERE в запрос |  | + поле в раздел WHERE | + знак в раздел WHERE | + значение в раздел WHERE | + логическое условие в раздел WHERE |
| Щелчок по выбору операций | … | … | … |  | … |  | … | … |  | … | … | … | … |

Множество входных сигналов данной таблицы зависит от того, как в конкретном случае пользователь организует мелкие детали интерфейса.

Таблица состояний отображает возможность выполнения действия пользователя с данным объектом интерфейса в конкретном состоянии. Заполняется следующим образом: если действие возможно с конкретным объектом в контурном состоянии – в соответствующую ячейку вписывается, какое действие производится. Пустая ячейка означает, что, либо объект в данном состоянии недоступен, либо он доступен, но действий с ним выполнить нельзя.

1. Таблица переходов.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sigma\Q | S | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Меню таблицы |  | 2 |  |  |  |  |  |  |  |  |  |  |  |
| Кнопка “ОК” |  |  | 3 |  | 2 |  |  | 8 |  |  |  |  | 8 |
| Сообщение об ошибке | … | … | … |  | … |  | … | … |  | … | … | … | … |
| Связь с таблицей | … | … | … |  | … |  | … | … |  | … | … | … | … |
| Режим “запрос” | … | … | …. |  | … |  | … | … |  | … | … | … | … |
| Кнопка “Далее” |  |  |  |  |  |  | 7 | 9 |  | 10 | 11 | 12 | 9 |
| Щелчок на списки по выбору операций (или другое) | … | … | … |  | … |  | … | … |  | … | … | … | … |

Заполнение таблицы переходов должно быть идентичным с заполнением таблицы состояний (2), за исключением той разницы, что в таблицы состояний в ячейке указывается действие пользователя или системы, а в таблицы переходов в соответствующей ячейке указывается состояние, в которое перейдёт приложение по этому действию.

**Алгоритм обработки логики диалога в соответствии с теорией конечных автоматов:**

1. Необходимо выявить входной сигнал или обратную семантическую связь для логики диалога.
2. Найти по таблице состояний (в зависимости от входного сигнала и текущего состояния) необходимое действие и выполнить его.
3. Изменить состояние приложения или системы согласно таблице переходов, в зависимости от входного сигнала и текущего состояния логики диалога.
4. Установить контекст нового состоянии в соответствии с таблицей контекста состояний.
5. Повторять данную последовательность до тех пор, пока логика диалога не окажется в финишном состоянии.

Достоинства простой транзитивной сети:

1. Простая транзитивная сеть представляет точное формальное описание логики диалога любых прикладных приложений и систем.
2. Диалог представляется в двух формах как в графической, так и в табличной, что удобно не только читать диалог, но и его выполнять.
3. Такая реализация диалога легко редактируется и дополняется.

Недостатки простой транзитивной сети:

1. Опасность неэффективного использования памяти, т.к. таблицы могут содержать много однотипной информации, а также быть не до конца заполненными. Если граф представляется в линейном виде (минимум разветвлений), то реализация его в виде конечного автомата может оказать нецелесообразной. Если граф реализуется в произвольной форме, где количество переходов стремится к максимально возможному (связь всех со всеми), то реализации транзитивной сети крайне эффективна.
2. Система помнит только одно состояние, то есть нельзя прервать обычный порядок работы диалога, выполнить какие-либо действия и вернуться к состоянию, предшествующему нарушению диалога.

Чтобы избежать этих недостатков, рекомендуется реализация диалога с использованием рекурсивной транзитивной сети.

**Транзитивная рекурсивная сеть**

Основным признаком необходимости построения рекурсивной транзитивной сети является наличие подсетей. Для подсетей характерны все признаки основной сети, а отличие состоит в том, что в подсеть можно попасть из различных состояний основной сети, а затем вернуться из подсети в состояние, явившееся точкой входа в подсеть. При этом рассматривается два основных критерия, наличие которых говорит о том, что необходимо строить подсеть.

Обособленная часть программы или алгоритма, которая выполняется много раз, и вызывается из определенной точки сети. В этом случае подсеть строится по тем же правилам, что и основная сеть (наличие стартового и финишного состояния для подсети, правила именования состояний и переходов).

Рассмотрим пример построения рекурсивной транзитивной сети на предыдущем примере, с добавлением функции ведения архива выполненных запросов, с целью их сохранения и повторного выполнения. Подобный диалог реализуется в виде рекурсивной транзитивной сети, который основывается на том же конечном автомате с единственным дополнением: у такого диалога должен вестись стек состояний для корректной обработки входа и выхода из подсети. Данный диалог со стековым автоматом описывается следующей математической формулой: H = (Q, Sigma, P, Г, delta, q0,z0, F), где:

1. Q – множество состояний диалога приложений.
2. Sigma – множество входных сигналов диалога.
3. P - множество процедур.
4. Г - множество элементов стека.
5. delta - функция переходов сети.
6. q0 - начальное состояние сети.
7. z0 - начальное состояние стека.
8. F - множество конечных состояний сети.

В случае построения сети по первому критерию для основной сети вводится состояние, которое будет являться точкой входа в подсеть (прямоугольник с закруглёнными углами). В случае, если подсеть строится по второму критерию, т.е. при наличии рекурсии, то на основной сети никак не отображается состояние, являющееся точкой входа, поскольку в рекурсию можно войти из любого (из нескольких) состояния основной сети.

1

2

3

1 состояние – просмотр текстов запросов.

2 состояние – просмотр данных.

1-2 (2 дуга): над - нажатие на текст запроса, под - выполнение выбранного запроса.

2-1 (1 дуга): над - нажатие кнопки “вперёд”, под - выбор следующего запроса по списку.

2-1 (3 дуга) над - нажатие кнопки “назад”, под - выбор предыдущего запроса по списку.

Для реализации стекового автомата, таблицы (таблица контекста состояний, таблица состояний и таблица переходов) дополняются состояниями подсети, а также входными сигналами диалога внутри подсети и входными сигналами между сетью и подсетью (то есть сигнал входа в подсеть и сигнал выхода из подсети). Именно для данных сигналов функция переходов включает состояния и функцию, осуществляющую запись в стек, если входной сигнал пришёл в рамках основной сети. Если сигнал пришёл из подсети, то функция переходов должна содержать функцию извлечения из стека.

**Проектирование конкурентного диалога**

**Пример:** нажатие на режим “курсив” в одном случает приведёт к включению, в другом к выключению. Работу любой транзитивной сети следует рассматривать как передвижение единственной метки из одного состояния в другое, то есть система диалога в каждый момент времени может находиться в единственном состоянии. Поэтому недостатком использования транзитивных сетей, как простых, так и рекурсивных, является слабая возможность отражения конкурентного диалога. Конкурентный диалог характерен для систем, которые можно описать более чем одним состоянием одновременно. Если рассматривать работу всего двух переключателей (курсив, жирный), то нажатие каждого из них приводит систему в противоположное состояние, то есть для двух переключателей требуется минимум 4 состояния:

12

1 состояние – простой текст.

2 состояние – нажат курсив.

3 состояние – нажат жирный.

4 состояние – жирный + курсив.

1-2: над – нажат переключатель курсива, под – текст изменён на курсив.

2-1: над – нажат переключатель курсива, под – курсив выключен.

…

Если рассматривать подобную ситуацию более чем для двух переключателей или подобных состояний системы, то количество состояний вычисляется как 2^n, где n – количество переключателей. Подобное разрастание транзитивной сети приводит к тому, что появляется много однотипной информации, которая перегружает сеть. Первоначально данную ситуацию пытались решить вверением в транзитивную сеть так называемых **транзитных сетей** (1 транзитная сеть строится для одного переключателя). В этом случае транзитная сеть для каждого переключателя будет иметь вот такой вид:

Но с введение транзитных сетей появляются новых трудности. При реализации такого подхода через набор конечных автоматов необходимо иметь дополнительный механизм определения, какому конечному автомату относится входной сигнал. Кроме того, необходим механизм для формирования общего контекста для всех транзитивных и транзитных сетей. Для устранения данных недостатков было выработано альтернативное решение, такое как использование инструментария сетей Петри для проектирования конкурентного диалога.

**Сети Петри в контексте проектирования конкурентного диалога**

Сеть Петри как механизм изначально не обозначен для проектирования сетей, которые имеют характеристики, отличные от характеристик конечного автомата. Поэтому данный механизм идеально подходит для проектирования конкурентного диалога. В теории сеть Петри представляют как граф, функционирование которого описывается перемещением многих меток по графу. Основные элементы, отображаемые на сетях Петри:

1. Положение. Тоже самое, что и состояния. Отображаются окружностями и представляют собой статическое представление диалога, в котором ни одно из положений или элементов не изменяется во времени.
2. Действие пользователя. То есть это события, происходящие в рамках проектирования диалога. Обозначаются эллипсом.
3. Переходы. Обозначаются прямоугольниками и представляют из себя триггерные элементы, которые срабатывают при условии активности всех предыдущих элементов.
4. Стрелки, показывают перемещение меток между всеми предыдущими элементами.

Характеристики функционирования сетей Петри:

1. Метки на сетях Петри могут содержать только окружности и эллипсы. Прямоугольники — это элементы, которые перемещают данные метки.
2. Переходы как триггерные элементы имеют входные дуги из окружностей и эллипсов и выходные дуги к окружностям.
3. Схема работы сети Петри в общем случае следующая:

если все вершины на входных дугах перехода содержат метки, то переход выполняется и все метки перемещаются в вершины выходных дуг переходов.

Если диалог содержит хотя бы один конкурентный элемент, то весь диалог приложения или системы проектироваться с помощью сетей Петри. В общем случае представление сети Петри будет иметь следующий вид (конкурентный диалог про включение курсива и жирного текста):

Картинка

1. Нажатие переключателя курсива.
2. Переход 1.
3. Нажатие переключателя курсива.
4. Переход 2.

Сети Петри могут содержать дополнительные входные дуги для переходов, оканчивающиеся не стрелкой, а окружностью. Такая дуга означает, что переход не выполняется, если вершина, соединённая этой дугой с переходом, содержит метку. Пример:

Приложение, которое позволяет выбрать страну, для каждой страны строится её список городов, которые также можно выбрать

Картинка 2 подписи посм у мишы

Переходы подписать переход 1,2,3,4(прямоугольники)

1. Список стран включён. (переход 8\_
2. Список стран выключен. ( состояние 7 – список стран скрыт)\_

Список город выключить – зависимое 6. Состояние список городов скрыт

3 переход список городов включён 5 состояние зависимое список городов отображается

3 картинка – сеть петри для примера про 2 переключателя курсив и жирный

**Текстовое представление диалога**

Наиболее удобные и простые виды описания диалога в текстовом представлении – это BNF форма и регулярные выражения. Основным отличием этих форм от сетей переход – внимание фокусируется не на состояниях системы, а на действиях пользователя. Форма BNF известна по описания обычных ЯП. В этой нотации присутствуют 2 основных элемента диалога: последовательность и вывод. В общем диалог в форме BNF можно представить:

Высказывание == пустая последовательноcть

| слово высказывание

| ‘(‘высказывание’)’ высказывание

Регулярные выражения более просты и коротки в записи. Основные элементы РВ:

1. Последовательность символов (или высказываний).
2. Применение специальных символов или их последовательности. (\* - Звезда Клини)

Применение \* означает, что последовательность, приведённая до него, повторяется 0 или множество раз. Наиболее широкое применение РВ получили в текстовых редакторах для формирования критерия поиска и установок характеристик, а также в лексическом анализаторе ЯП.

Пример РВ:

Рисование ломанной линии – выбор характеристики щелчок (движение-щелчок)\* двойной щелчок.

РВ представляют собой последовательность термов, каждый из который является примитивом для данного графического интерфейса или определяется другим РВ, при этом следует разграничивать примитивы и определения РВ. Примитивы для графического интерфейса не всегда ведут к изменению контекста диалога, при выделении очередного РВ подразумевается, что данное правило описывает задействование модуля логики диалога и изменение контекста диалога.

Ни BNF форма, ни РВ не имеют аппарата для проектирования конкурентного диалога. Данный недостатков устраняется в ещё одном текстовом представлении диалога – продукционные правила.

**Продукционные правила**

Основная конструкция, которая используется в описании продукционных правил имеет следующий вид:

If <условие> then <действие>

Смысловое представление каждого правила:

<условие>🡪<действие> или <условие>:<действие>

При построении системы продукционных правил следует учитывать, что все правила равноправны, то есть ни форма их записи, ни последовательность этой записи в системе ни на что не влияют.

Действия продукционных правил читаются следующим образом: модуль логики диалога подбирает условную часть правила (любое одно или несколько правил) в зависимости от события, которое было порождено пользователем, либо системой (приложением), если она выступает в качестве активной подсистемы. Когда условная часть продукционного правила становится истинной – правило срабатывает и выполняется действительная часть правила. По аналогии с другими моделями описания диалога действие может иметь вид ответа пользователю или изменения контекста самой системы (приложения).

На основе данной нотации (условная-действительная) выделяют 3 вида продукционных правил для описания диалога:

1. Продукционные правила, ориентированные на события.
2. Продукционные правила, ориентированные на состояния.
3. Смешанные продукционные правила.

**Продукционные правила, ориентированные на события**

Представим диалог, рисующий ломанную линию в нотации продукционных правил, ориентированных на события:

Выбрать\_линию 🡪 начать\_линию

<высветить “линия”>

Щелчок начать\_линию 🡪 неизменная\_линия

<резиновая нить “вкл”>

Щелчок неизменная\_линия 🡪 неизменная\_линия

<рисовать линию>

Двойной\_щелчок неизменная\_линия 🡪 <рисовать линию>

<резиновая нить “откл”>

При таком подходе условие и действие правила имеют вид множеств именованных событий. События могут быть 3х видов:

1. Пользовательские события, то есть то, что породил пользователь конкретным действием. Такие события в продукционном правиле отображаются именем с Большой буквы.
2. Внутренние события. Им имя начинается с маленькой буквы и используется для сохранения состояния диалога. Данные события порождаются только системой или приложением.
3. События отклика системы. В правилах они заключены в < … > и в нотации диалога – это всегда изменение контекста.

Блок управления диалога, который поддерживает продукционные правила такого вида, должен иметь память, состоящую из множества событий. Правило срабатывает, если все события условной части находятся в памяти. Все события пользователя неизменно заносятся в память, а система откликается путём порождения собственных событий. Когда правило срабатывает – все события, упомянутые в его условии, удаляются из памяти, а все события действительной части правила заносятся в память. При реализации такого диалога ориентированные на события продукционные правила ведут к постоянного удалению и добавлению большого числа событий, что затрудняет хранение статической информации о диалоге. Иногда это может приводить к ситуации, когда событие присутствует в условной и действительной частях правила. Событие “неизменная\_линия” имитирует состояние “<рисовать линию>”, постоянно находится в памяти и постоянно приводит к двум действию: удалению себя из памяти и добавлению себя в память.

**Продукционные правила, ориентированные на состояния**

В системе продукционных правил, ориентированных на состояние, память представляет собой множество именованных значений. Отличие от предыдущего вида правил – элементы не удаляются из памяти при выполнении правила. В такой нотации состояния делятся на множество атрибутов, каждый из которых может принимать одно или несколько значений. При этом одни атрибуты представляют собой действия пользователя, остальные – работу системы (приложения).

Для диалога рисования ломанной линии выделяем 5 следующих атрибутов:

1. Мышь {не работает, выбор линии, щелчок, двойной щелчок}.
2. Состоянии линии {выбор изменён, начало, неизменная линия}.
3. Резиновая нить {вкл., откл.}.
4. Меню {ничего не выбрано, выбрана линяя}.
5. Рисование {ничего не рисуется, рисуется линия}.

При выделении данных атрибутов мы руководствуемся следующим: первых атрибут характеризует действие пользователя, второй – внутреннее состояние системы или приложения, 3-5 атрибуты – отклик системы или приложения. Продукционные правила, основанные на состояниях, имеют следующий вид:

Выбор изменён 🡪 мышь не работает, начало, высвеченная линяя

Щелчок, начало 🡪 мышь не работает, неизменная линяя, вкл.

Щелчок, неизменная линяя 🡪 мышь не работает, рисуется линия

Двойной щелчок, неизменная линяя 🡪 мышь не работает, рисуется линяя, откл.

Механизм работы продукционных правил, основанных на событиях:

Правило выполняется тогда, когда состояние системы соответствует его условной части. Под состоянием в данном случае понимаются атрибуты с их текущими значениями. Выполнение правила не приводит к обязательной смене значений атрибутов – это происходит, если действие правила содержит новое значение атрибута. Например, второе правило срабатывает, если атрибут мышь имеет значение щелчок, а атрибут состояние линии – значение начало, при любых других значениях остальных атрибутов. Выполнение данного правила приведёт к изменению атрибута мышь на мышь не работает, а состояние линии – на значение “неизменная линяя” и “резиновая нить” к значению “вкл.”. Атрибуты в системе правил, ориентированных на состояния, достаточно устойчивые, но бывают исключения (атрибут “мышь”). Он присутствует в обеих частях правилах также, как и в правилах, ориентированных на события, то есть при выполнении любого правила, атрибут “мышь” всегда меняет своё значение. С другой стороны, значение этого атрибута изменяется пользователем, что и приводит к выполнению правила.

Рассмотрим, к чему приведёт отсутствие атрибута “мышь” в действии правила. Пусть во 2ом правиле отсутствует указание на изменение значения этого атрибута. Тогда по окончанию его работы значение атрибута “мышь” остаётся равным “щелчок”. При значении атрибута “состояние линии”, равном “неизменная линяя”, немедленно срабатывает 3е правило. Если это правило не имеет указание изменить значение атрибута “мышь”, то 3е правило будет выполняться циклически (без каких-либо принуждений со стороны пользователя), пока действия пользователя не прервут этот цикл.

**Смешанные продукционные правила**

Классический подход к проектированию диалога требует применения одного из двух подходов построения продукционных правил. В свою очередь смешанный подход в отдельных случаях может быть наиболее эффективен. При этом подходе правила строятся по следующему сценарию:

Событие: условие 🡪 действие

Другими словами, учитывается особенность правил, основанных на событиях выделять произошедшее событие в контексте пользовательского диалога, а по правилам, основанных на состояниях – применять к этим событиям условия, в которых они происходят. Наступление события можно представить, как сигнал к выполнению правила. Но правило не срабатывает, если условия его выполнения не истинны. События порождаются пользователем и данный атрибут в правиле остаётся неизменным, пока он не встретится в действии этого правила. На данном подходе легко проектируется конкурентный пользовательский диалог.

**Пример построения смешанных продукционных правил.** Имеет 3 переключателя: жирный, курсив, подчёркивание, который работают в режиме конкурентного диалога. В данном диалоге выделяется 3 атрибута: подчёркивание {включён, выключен}, жирный {включён, выключен}, курсив {включён, выключен}.

В системе существует 3 события: переключение подчёркивания, переключение курсива, переключение жирного шрифта.

Строится 6 смешанных продукционных правил:

1. Переключение подчёркивания: подчёркивание = выкл. 🡪 подчёркивание = вкл.
2. Переключение подчёркивания: подчёркивание = вкл. 🡪 подчёркивание = выкл.
3. Переключение курсива: курсив = выкл. 🡪 курсив = вкл.
4. Переключение курсива: курсив = вкл. 🡪 курсив = выкл.
5. Переключение жирного: жирный = выкл. 🡪 жирный = вкл.
6. Переключение жирного: жирный = вкл. 🡪 жирный = выкл.

Некоторый правила в системе продукционных правил могут не иметь условий – это происходит в тех случаях, когда событие (действие пользователя), выполняемое в ОС, имеет наивысший приоритет перед всеми остальными событиями. При этом никак не учитываются условия, в которых происходит данное событие. В этом случае смешанное правило, например, для события нажатия на кнопку закрытие или “выход” будет иметь следующий вид:

Выход: завершить работу

В этом случае модуль логики диалога выполнение данного правила вообще никак не задействовано. Другими словами, событие регистрируется ОС и модулем выполняется, не оповещая при этом приложение и действие всегда передаётся напрямую приложению.