**Лекция 2**

**Обработка ошибок в Windows**

При реализации ПО надо учитывать, что любая ф-я WinAPI может быть обработана неверно, что для приложения может иметь последствия. Если вызывается ф-я WinAPI, она проверяет переданные параметры, а потом –выполняет свою работу. Параметры передаются стандартно. Передача параметров в ф-ю:

В ф-ю передается указатель на адрес памяти, где хранится параметр (даже если он простого типа). Потом ОС (в частности, транслятор языка) определяет размер параметра, его кодировку и тип. Если данные параметры не соответствует тому, что требует ф-я, происходит их изменение следующим образом: в зависимости от разрядности процессора происходит приведение типа (как расширяющее, так и сужающее), а также изменяется разрядность сетки.

Повторить двоичную СС, способы перевода

Если в ф-ю передается недопустимый параметр и ее нельзя выполнить еще по какой-то причине, ф-я возвращает значение, говорящее об ошибке в ее работе. Типы функций и значения ошибок взаимоувязаны.

|  |  |
| --- | --- |
| **Тип ф-и** | **Значение ошибки** |
| VOID | Такие ф-и почти всегда выполняются успешно. Таких ф-й достаточно мало. Здесь понятие ошибки специфично |
| BOOL | Если вызов заканчивается неудачно, возвращается ноль(не логическое). В остальных случаях – значение, отличное от нуля |
| HANDLE | Если ф-я сработала ошибочно, то обычно возвращается NULL. В остальных случаях – идентифицируется объект, которым можно манипулировать. Некоторые HANDLE-ф-и могут возвращать значение INVALID\_HANDLE\_VALUE, равно -1. Чтобы выяснить, сработала ф-я ошибочно или нет, надо смотреть документацию |
| PVOID | Такие ф-и работают с памятью напрямую. Об ошибке говорит значение NULL. В остальных случаях – возвращается адрес блока данных в памяти |
| LONG/DWORD | Данные ф-и являются счетчиками или указателями. Если ф-я не считала и сосчитать значение счетчика, она возвращается 0 или -1 |

При возникновении ошибки в Win-приложениях надо разобраться, почему вызов ф-и оказался неудачным. Для анализа каждая ошибка кодируется 32-битным числом, которое записывается в локальную память потока, в котором произошла ошибка. Этот механизм позволяет отслеживать ошибки для каждого потока отдельно => ошибка одного потока не может изменить код ошибки другого потока. Когда ф-я, выполнившаяся с ошибкой, вернет управление ОС, ее значение говорит об ошибке и в каких условиях эта ошибка произошла. Получить код ошибки можно с помощью ф-и GetLastError(), которая выдает код последней ошибки в потоке. Данная ф-я возвращает код последней ошибки в потоке, т.к. локальная память каждого потока может хранить только один код ошибки. Чтобы отловить каждую ошибку в потоке, данную ф-ю надо вызывать сразу после той, которая совершила ошибку. Все определенные ошибки хранятся в одном файле – winError.h по структуре:

Идентификатор ошибки

Код ошибки

Описание ошибки

Описание ошибки на национальном языке

Чтобы по коду ошибки получить ее описание, юзается ф-я FormatMessage().

32-битный код ошибки разбит на регионы, каждый из которых содержит ин-ю о параметрах ошибки.

31 – 30 биты: код тяжести ошибки. Если 0 – успех, 1 – ин-я, 2 - предупреждение, 3 – ошибка.

29 бит – значение показывает, кем определен код ошибки – Microsoft или юзер. Если 0 – ошибка Microsoft, 1 – ошибка юзера

28 бит – зарезервирован всегда, 0.

27 – 16 биты – значение кода подсистемы, определяемой в Windows. Он связан не только с типом ошибки, но и с регионом адресного пространства, в котором она произошла

15 – 0 биты – код исключения ошибки, определяется юзером или Microsoft

SetLastError() – установка кода ошибки.

**Процессы в Windows**

**Адресное пространство процесса**

Процесс – экземпляр выполняемой программы, который состоит из 2-х компонентов. 1 – объект ядра процесса, через который ОС управляет данным процессом. В ядре хранится статическая ин-я о процессе. 2 – адресное пространство. В нем – код и данные всех exe- и dll-модулей исполняемой программы. Именно здесь находятся области памяти, динамически распределяемые для стеков потоков, для самих потоков и других нужд. Все процессы инертны. Чтобы процесс что-то выполнил, в нем надо создать минимум один поток. Именно потоки отвечают за исполнение кода в адресном пространстве. Каждый процесс может владеть несколькими потоками, которые одновременно выполняются в адресном пространстве процесса.

Чтобы все потоки в процессе работали, ОС отводит каждому из них определенное процессорное время. Данные отрезки времени – кванты и распределяются между потоками по принципу карусели. При создании процесса первый поток в системе создается автоматически ОС. Если поток не создается, то процесс сразу умирает. Следующие потки могут создаваться как ОС, так и юзером, но они всегда являются дочерними по отношению к главному.

**Типы приложений Windows**

**Процедура сборки и компиляции**

В зависимости от процедуры сборки win-приложения есть 2 типа приложений, в которых процедура сборки и компоновки кардинально отличаются.

1. Приложения с графическом интерфейсом – GUI-приложения
2. Консольные приложения – CUI.

В последних версиях Windows визуальные данные приложения сложно отличить, т.к. истинная консоль, как правило, закрыта от юзера. Максимум – эмуляция консольной строки. Собственно консольные приложения выполняются в окне. У GUI-приложений весь интерфейс чисто графический. Они создают окна, имеют меню, способны организовывать диалог через диалоговые окна и взаимодействовать с юзером через windows-сообщения.

CUI-приложения всегда работают в текстовом режиме. Даже если текстовый режим представлен в главном окне. Приложения не формируют окон, не способны реагировать на действия юзера, не обрабатывают сообщений и не требуют графический библиотек Windows.

При создании проекта приложения Windows компоновщику нужно указать ключ подсистемы, на основе которой будет собираться Win-приложение. Если компоновщик получает ключ /SUBSYSTEM:CONSOLE, он понимает, что нужно произвести компоновку консольного приложения с подключением всех необходимых библиотек. Для граф. приложений и их компоновки ключ - /SUBSYSTEM:WINDOW.

Когда юзер запускает приложение, загрузчик ОС проверяет номер/тип подсистемы приложения, хранящиеся в заголовке exe-файла, и определяет, что за приложение нужно запустить: графические или консольное. Если тип указывает на консольное приложение, то загрузчик создает текстовое консольное окно и «привязывает» к нему исполняемое консольное приложение; иначе – программа просто загружается в память.

В любом Win-приложении должна быть входная ф-я, за реализацию которой отвечает разработчик. В Windows для загрузчика ОС есть 4 типа этой ф-и:

1. WinMain
2. wWinMain
3. main
4. wmain

Данная ф-я фактически не является входной для ОС. Вместо этого происходит обращение к стартовой ф-и из библиотек Си. Эта ф-я инициализирует все нужные библиотеки (именно поэтому в win- приложениях не надо подключать библиотеки), инициализирует ф-и по работе с памятью, обеспечивает корректное создание любых объявленных глобальных и статических Си-объектов. Только после этого вызывается входная ф-и Win-приложения и передается ей управление.

|  |  |  |
| --- | --- | --- |
| **Тип приложения** | **Входная ф-и** | **Стартовая ф-и Windows** |
| GUI, работающая с ANSI | WinMain | WinMainCRTStartUp() |
| GUI, работающая с UNICODE | wWinMain | wWinMainCRTStartUp() |
| CUI, работающая с ANSI | main | mainCRTStartUp() |
| CUI, работающая с UNICODE | wmain | wmainCRTStartUp() |

Нужную стартовую ф-ю в библиотеке Си выбирает компоновщик при сборке exe-файла. Если он находит ключ «…WINDOW», то в коде он ищет 2 ф-и WinMain и wWinMain. Если нет ни одной, то ошибка «Неразрешенный внешний символ» и аварийно завершает сборку приложения. Иначе – вызывается соответствующая стартовая ф-я и ее код встраивается в exe-файл.

Большинство IDE для Win-приложений имеют одно грубое допущение: они не предлагают для компоновщика выбора в типе приложения (по дефолту всегда ключ «…WINDOW»), а для данного ключа подключает все библиотеки Windows, которые могут понадобиться оконному приложению, создает по дефолту главное окно приложения и скрывает его от юзера. Практически все IDE не разрешают изменить данный порядок компоновки юзеру, а если и разрешают (не запрещают изменять эти параметры), то не предоставляют для этого никаких средств.

**Автоматически подключаемые библиотеки**

1. KERNEL32.DLL

Ядро ОС, память, загрузчик ОС

1. USER32.DLL

Диалоги, регистрация событий, работа юзера

1. GDI32.DLL

Графика

1. ADWAPI32.DLL

Работа с железом – драйверы

*Если разрядность выше, чем 32, то…*

**Архитектура памяти Windows**

Для выполнения любой программы в Windows нужна память, в которой она будет хранится как данные, которыми она манипулирует, так и код, который выполняется. Эта память по отношению к работающему процессу является виртуальной и по определению самой Windows – эта область памяти закрыта => если в процессе выполняется какой-либо поток, то он может получить доступ только к той памяти, которой владеет ее процесс. Если в процессе есть несколько поток, они все имеют доступ к одной и той же области памяти, которая закреплена за их процессом. Важно, что адресное пространство является виртуальным => для того, чтобы процесс мог обратиться по какому-либо адресу в этом виртуальном пространстве, его надо спроецировать на физическую память.

Каждое виртуальное адресное пространство разбивается на разделы и их назначение и размер зависят от ядра и версии Windows.

Разделы виртуального адресного пространства:

1. *Раздел для выявления нулевых указателей.*

Резервируется для того, чтобы разработчики программ могли выявлять ошибки в указателях программ и определять ссылку на несуществующие адреса памяти. Любая попытка чтения или записи по адресам данного региона вызывает нарушение доступа. Если ф-я выдает подобную ошибку, то ее возвращаемое значение всегда NULL. При данном значении ф-я, вызывающая ошибку, пытается обратиться повторно по адресу 0000 (8 или 16 нулей).

1. *Для совместимости с различными версиями ОС и DOS-программами.*

Этот раздел появился и развивался с каждой новой версией Windows и нужен, чтобы обеспечить запуск Win-приложений, созданные под платформы другой разрядности. В данном разделе находится код программ, требующих совместимости, но этот раздел закрыт для запускаемого приложения и доступ к нему имеет только ОС. При запуске программ разной разрядности, у каждой есть свой регион в этом разделе, он обособлен, ОС следит, чтобы из данных регионов не было ссылок на другие регионы адресного пространства.

1. *Для кода и данным пользовательского режима.*

Это раздел представляет собой закрытую, неразделяемую часть адресного пространства процесса. Ни один процесс не может получить доступ к данных другого процесса, размещенного в данном разделе (здесь приложение имеет полный доступ только к своему разделу). Основной объем данных, принадлежащих процессу (данные процесса, кода процесса, ссылки на библиотеки и сторонние ресурсы, статические и глобальные переменные и объекты. Здесь нет никаких ограничений у приложения. Сюда, после запуска процесса, подгружается доп. информация, не имеющая отношения к процессу, но нужная для выполнения: часть кода ядра, драйверы устройств, кэш-буферы IO, области памяти, не сбрасываемые в файл подкачки, таблицы для контроля страниц памяти в процессе, информация из файловой системы, если она нужна процессу).

1. *Закрытый раздел.*

Этот раздел заблокирован Microsoft и любая попытка обращения к нему приводит к нарушению доступа. Этот раздел резервируется, чтобы упростить внутреннюю реализацию ОС и нужен для организации взаимодействия приложения и ОС. Информация о том, где хранить очередь сообщений и какие сообщения относятся к данному приложению. Информация из очереди сообщений, относящихся к данному приложению. Этот раздел заблокирован и приложение прямого доступа к нему не имеет, но обратиться к нему может ОС (или ее отдельный процесс, запущенный в режиме ядра).

1. *Для общих файлов, проецируемых в память.*

В этом разделе хранятся код и данные, разделяемые между всеми запущенными процессами (независимо от разрядности и раздела, в котором хранятся данные процесса). Сюда грузятся самые первые 4 библиотеки. Данные в этом разделе приложениями только читаются, практически изменить они не могут, за редким исключением, если оно имеет приоритет 0. Изменять такие приложения могут только данные в своих подгружаемых библиотек. Сюда грузятся все библиотеки, на которые ссылается приложение, здесь же хранятся объекты, связанные с этими библиотеки и могут появляться сообщения, генерируемые кодом библиотек.

1. *Для кода и данных режима ядра.*

Этот раздел очень специфичен и юзается для сетевой поддержки Windows. Сюда помещаются драйверы устройств и код низко уровневого управления сетевыми потоками, код управления памятью и файловой системой, код сетевых протоколов, сетевых драйверов и подключений. Эти данные защищены, доступ из них можно получить только по ряду ограниченных адресов (адреса, доступ к которым открыт данной ОС). Поток, который пытается обратиться по одному из адресов памяти в данном разделе, вызовет нарушение доступа в случае, если поток не имеет прав на чтение этих данных.

Какая бы программа ни запускалась и какой бы разрядности ни была платформа данные разделы всегда есть в адресном пространстве и их использование определяет ОС.

**Библиотеки Windows**

Динамически Подключаемые библиотеки – снова функционирования ОС, т.к. весь код ядра, системных компонент и утилит реализованы в виде библиотек. С развитием ОС в библиотеки стали помещать не только исполняемый код, но и отдельные компоненты, которые требуются приложениям только по мере их требования. Это позволило сократить код и объем исполняемых компонент => добиться экономии виртуальной памяти. При юзании библиотек удалось добиться следующих улучшений:

1. Расширение функциональности приложений. DLL можно грузить в адресное пространство процесса динамически => приложение, определив, какие действия от него требуется, подгружает только нужный код.
2. Возможность юзания различных языков программирования. Т.к. DLL – откомпилированный компонент, приложение и ОС все равно, на каком языке реализован первоначальный код. Это же предоставлено и разработчикам, сняв с них ограничение в выборе языка реализации.
3. Более простое управление проектом. Если в процессе разработки ПО отдельные части создаются разными группами, то при юзании библиотек упрощается взаимодействие этих групп. Достаточно каждой библиотеке сделать описание реализованных ф-й и предоставить их другим участникам проекта.
4. Экономия памяти. Загрузчик ОС при необходимости может подгружать в адресное пространство не только код библиотеки, но и код отдельных ф-й, если загрузка динамическая.
5. Экономия и разделение ресурсов. В библиотеке можно реализовать такие ресурсы, как: шаблоны диалоговых окон, битовые карты, растровые изображения, значки, иконы и отдельны классы окон, элементы локализованного интерфейса.
6. Упрощение локализации ОС и приложений.
7. Решение проблем, связанных с особенностями платформ (с появлением 8-ки и 10-ки – почти самое важное улучшение). К библиотеке одной разрядности можно подключить библиотеку другой разрядности.

**Лекция 5**

**Этапы создания библиотек.**

Для создания библиотеки в зависимости от ее наполнения и ее экспорта/импорта, можно выделить следующие этапы создание динамически подключаемой библиотеки:

1)Прежде всего необходимо подготовить заголовочный файл с прототипами функций, структурами и идентификаторами, экспортируемыми из библиотеки. Данный файл включается в исходный код всех модулей будущей библиотеки

2)Ос-ся реализация исходного кода в самой библиотеке с полным определением всех функций и переменных, который должна использовать библиотека

3)В идеале, если исходный код библиотеки один, то на него создается один объектный файл, который и используется при сборке исполняемого модуля. Если файлов библиотеки несколько, то создается несколько объектных файлов. По одному на каждый модуль.

4)Компоновщик собирает все объектные модули, если их несколько, в один загрузочный dll-модуль, в который в конечном итоге помещаются двоичный код и переменные(глобальные и статические, которые использует данная библиотека или их экспортирует).Данный файл необходим при компиляции исполняемого модуля библиотеки

5) Если компоновщик обнаружит, что библиотека экспортирует хотя бы одну переменную или функцию, то она создаст еще и LIB-файл для данной библиотеки. Данный файл по размеру совсем крошечный, тк содержит только список символьных имен функций и переменных, экспортируемых из библиотеки. Итоговая компиляция библиотеки также невозможна без файла LIB

6)Загрузочный модуль библиотеки( или несколько модулей) и LIB-файл(при его наличии) компилируются в результате чего получается исполняемый модуль библиотеки, который можно использовать и подключать к любому приложению.

**Дома самостоятельно посмотреть код, который будем на лабе делать. Просто найти, как пишется исходный код, как создается объектный файл, как пишется заголовочный, как создается** LIB **файл и как происходит итоговая компиляция.**

Компилируя любую библиотеку следует помнить, что она представляет собой всего лишь набор автономных функций, который могут использоваться любой программой Windоws, несмотря на то, на каком ЯП они реализованы. Следует помнить, что в библиотеке нет кода, предназначенного для обработки циклов сообщений, для создания окон и управления ими, при компоновке библиотеки следует учитывать, что процедура ничем не отличается от компоновщик исполняемого модуля(exe). За исключением того, что компоновщику необходимо указать ключ /DLL. В этом случае, компоновщик записывает в исполняемый файл библиотеки(в его заголовочную часть) информацию, по которой загрузчик ОС понимает, что это библиотека, а не exe- модуль. При этом, действия загрузчика также отличаются от его действий при загрузке исполняемого модуля. Загрузчик определяет, какое приложение вызывает библиотеку, находит адресное пространство данного процесса, а в нем необходимый раздел для загрузки библиотеки и проецирует тело библиотеки в найденное адресное пространство. При этом, любая библиотека теряет свою индивидуальность. Для потоков и процесса код и данные библиотеки лишь дополнительный код, оказавшийся в адресном пространстве. Кроме того, любые созданные библиотекой объекты( точнее, функции этой библиотеки) принадлежат вызывающему процессу. DLL в принципе, ничем владеть не может.

При загрузке библиотеки в адресное пространство вызывается функция выделения памяти, которая резервирует регион в адресном пространстве, куда будет загружена библиотека. Если библиотека будет «выгружена» из адресного пространства процесса(процедура выгрузки для библиотеки невозможна) зарезервированный за библиотекой регион не освободится, тк система не фиксирует того, что данный регион был зареган для этой либы.Освобождение данного региона возможно только в том случае, когда вызывается функция освобождения памяти и процесс завершается, те адресное пространство разрушается. На основании этого, выделяют два вида связывания либы и связываемого модуля(вызывающего процесса):

1)Явное связывание библиотеки

Возможно только при использовании функции LoadLibrary() или ее разновидностей. В этом случае, ссылки на библиотеку устанавливаются в момент вызова функции и в адресное пространство процесса попадает весь код указанной библиотеки.

2)Неявное связывание библиотеки

Указывается ссылка на библиотеку в разделе описания внешних ссылок. После указания ссылки на библиотеку в коде программы при вызове любой из функций библиотеки, данная функция загружается в адресное пространство вызывающего процесса и исп-ся в исполняемом файле. Следует учесть, что при неявном связывании в адресное пространство попадает не весь код библиотеки, а только то, что было вызвано в ходе работы процесса. Несмотря на то, каким способом связывалась библиотека, процедура выгрузки для библиотеки идентична в обоих случаях. Производится она с помощью функции FreeLibrary(), которая уничтожает ссылки на библиотеку, оставляя код библиотеки в адресном пространстве.

**Самостоятельно можно посмотреть функции по работе с памятью в Windows,(деление региона, запись в память)**

**Проектирование пользовательского интерфейса. Структура интерфейса в Windows.**

Современные ср-ва разработки в большинстве своем предоставляют инструментарий для организации GUI любых пр-ний. Единственный недостаток такого способа разработки интерфейса это отсутствие концептуализации как минимум(при доработке, изменении или увеличении приложений увеличивается и объем ручной работы по орг-ции интерфейса в геометрической прогрессии) и как максимум невозможность предусмотреть все связи такого интерфейса во всех модулях разрабатываемой системы. Такими проблемами и их решением занимается отдельная область информационных систем, называющаяся **стандартизацией и проектированием GUI.** Стандарты разработки логики диалога с пользователем.

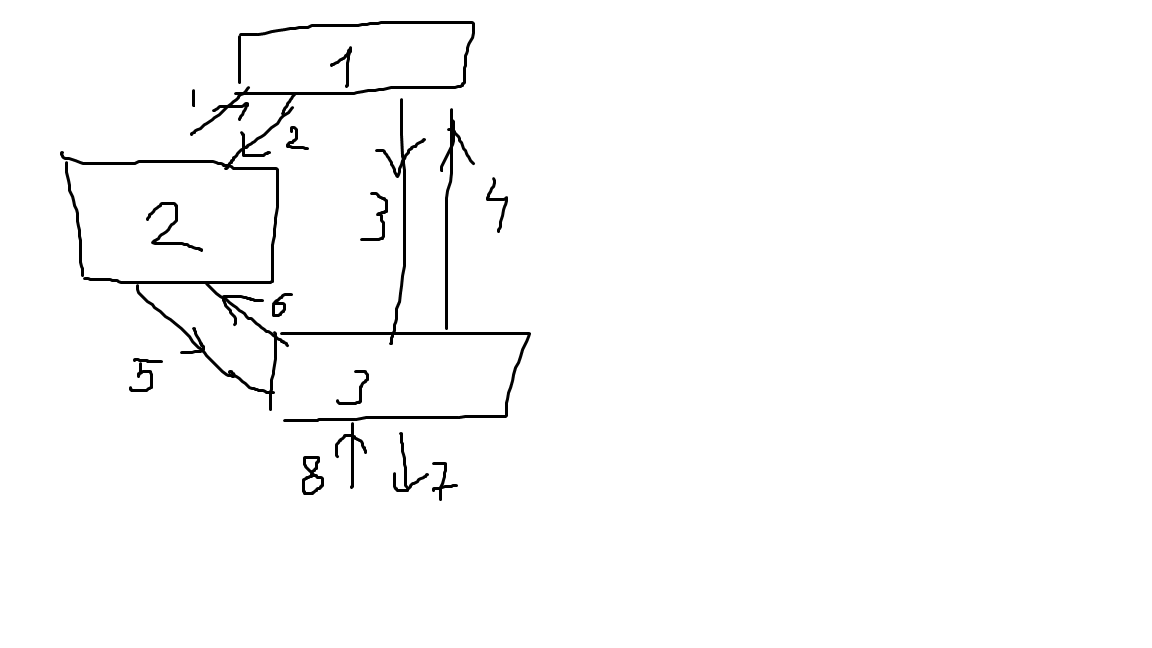
Рассмотрим процесс проектирования GUI на основе примера приложения по работе с БД, которая позволяет работать с приложением в двух режимах.

1. Режим таблицы
2. Режим запроса

При организации подобных приложений, которые работают со сторонними источниками (хранилище данных (БД), сетевые подключения, подключаемые динамические веб-ресурсы) предполагают отображение в окна приложений актуальной информации, независимо от того, в какой части пр-ния эти изменения были сделаны. При этом следует учитывать, что многооконное приложение также подчиняется концепции Windows по управлению окнами. Окна могут быть видимы, но не иметь фокуса; скрыты, но работать(свернутые); пассивными, но получать информацию; поэтому помимо алгоритмом обработки данных необходимо следить, чтобы интерфейс вовремя и правильно переключался, чтобы пользователь при выполнении своих действий не мог производить не продуктивный действия, которые могут повлиять на работу приложения. Помимо этого, внутри каждого окна мб свой внутренний диалог между окнами, который также необходимо учитывать в контексте общего диалога.

Отдельной задачей в ТППИ(теория проектирования пользовательского интерфейса) стоит вз-е окон и разных компонентов. При этом, следует учитывать, что орг-ция диалога, каким бы сложным и многоуровневым он не был, должен проектироваться с учетом требований предметной области приложения, а также требований эргономичности (пользователь при работе с приложением не должен задумываться о том, в какое окно и какую информацию ему нужно вводить и где прочитать результат своего запроса). Такие вопросы должны решаться только на уровне проектирования пользовательского интерфейса в соответствии с разработанными стандартами и моделями интерфейса. Несмотря на то, какой вид приложения реализуется, графическое или консольное(текстовое), организация диалога подразумевает задействование схемы пользовательского диалога Windows при разработке приложения.

Структура интерфейса и логики диалога windows



I прикладной интерфейс

II модуль логики диалога

III окна с визуальными элементами

1 команды

2 обратная семантическая связь

3 прикладные данные

4 запросы

5 контекст

6 входные сигналы

7 экран

8 событие

При проектировании диалога определяется состояние системы , устойчивое положение всех параметров системы, в т.ч. и параметры интерфейса, в котором пользователь может выполнять какие-либо действия. При этом следует учитывать, что действия могут как приводить к изменению интерфейса, так и не затрагивать его совсем. В каждом из этих случаев, обработка логики диалога ведется разными способами. Для пользователя любые приложения это окна с визуальными элементами. Весь диалог с приложением пользователь строит через эти элементы, порождая определенные события ( 8 стрелка).

Окна преобразуют события во входные сигналы для логики диалога (6 стрелка, входные сигналы) данный процесс невозможен без модуля логики диалога Windоws. При этом, не любое событие пользователя будем являться входным сигналом для диалога, а только то, которое влечет изменения в отображении интерфейсной части.

При обработке входных сигналов в модуле логики диалога, изменяется контекст изображения(5 стрелка), т.е. набор окон и их элементов, которые становятся доступны пользователю при обработке входных сигналов. Изменения контекста всегда отображается на экране (7 стрелка).

Прикладной интерфейс – специализированные программы(утилиты) для обращения к сторонним источникам (хранилище данных, ресурсы, выше было). Он функционирует независимо от того, каким образом работает приложение, получая от него только команды на выполнение( стрелка 1).

Получая команды, интерфейс выполняет свои действия (они никак не отображаются на интерфейсе приложения), но в ответ на команду интерфейс может генерировать обратную семантическую связь(2 стрелка).

Именно данная связь может генерировать изменения контекста интерфейса. Обратная семантическая связь также обрабатывается модулем логики диалога, что в свою очередь влияет на контекст (стрелка 5).

Приложение Windows, помимо управления внешним диалогом, может иметь свой внутренний диалог, функционирование которого не будет отображаться на пользовательском интерфейсе приложения. Такие ситуации возможны, когда пользователь через окно приложения делает запрос напрямую прикладному интерфейсу(стрелка 4). В этом случае интерфейс обратно присылает прикладные данные в ответ на запрос, причем в данном процессе модуль логики диалога не участвует.

**Простая транзитивная сеть как инструмент ППИ.**

Одним из наиболее распространенных способов проектирования логики диалога является простая транзитивная сеть. Она представляет собой графический способ представления диалога и отображается в виде направленного графа (**Дома: свойства направленных графов**).

Состояния графа (вершины)- состояния системы в контексте UI, (**III элемент)**

Переходы между вершинами графа отображаются дугами. (обратная семантическая связь(**стрелки 2 и 6**)

Дуги на транзитивной сети именуются следующим образом:

Сверху дуги именуются согласно входным сигналам или обратной семантической связи, снизу дуга именуется согласно действию, которое должно выполнить приложение при переходе из 1 состояния интерфейса в другое.

Основные элементы приложения, работающего с БД:

1. Меню содержит
   1. Режим таблицы

Элементы, нужные для режима таблицы

* 1. Режим запроса

Меню с выпадающими 2 пунктами выше, 1 - текстовое однострочное окно

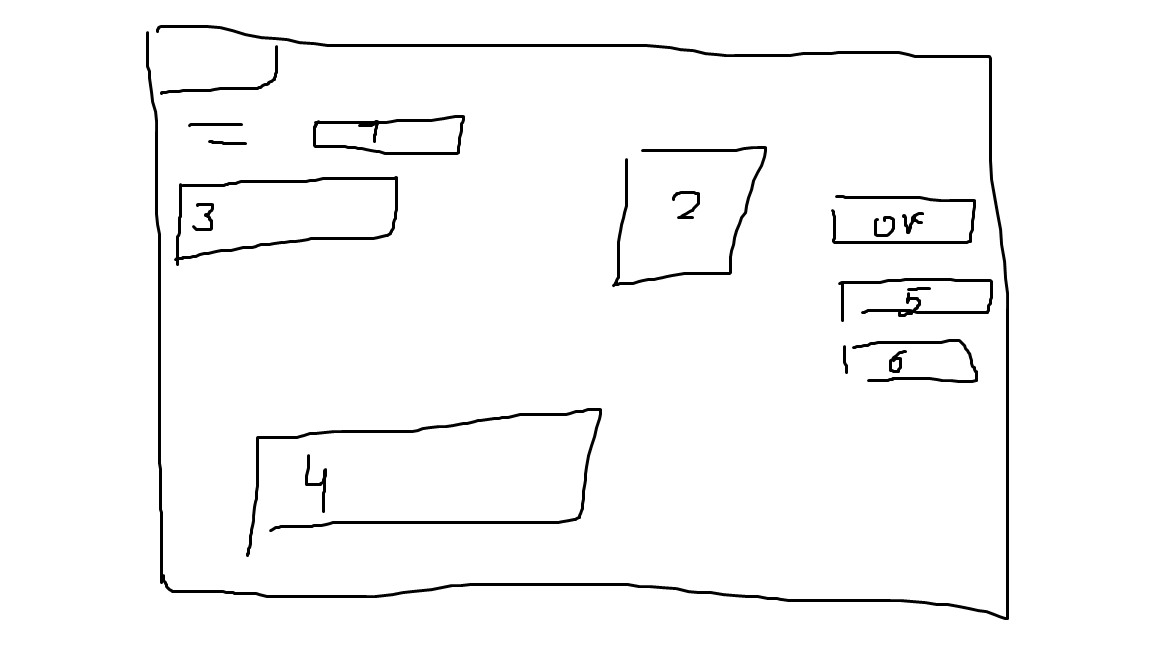
2 – многострочный список с полями выбранной таблицы

В однострочное текстовое поле вводится имя таблицы, к которой строится запрос. При любом событии (enter) в списке появляются все поля указанной таблицы. При нажатии или двойном нажатии на любое из полей в элемент многострочное текстовое поле(3) добавляется раздел запроса select с выбранными полями таблицы(выбрали 3 поле соответственно select 3 поле, выбрали 5 - + 5) при нажатии на кнопку ОК в раздел добавляется раздел from, выполняется сформированный запрос к бд, результат его отображается в таблице 4

В b Добавляется кнопка далее, добавляется выпадающий список (5) с операциями, однострочное текстовое поле, куда вводится значение для запроса (6)

Процесс функционирования в режиме запроса:

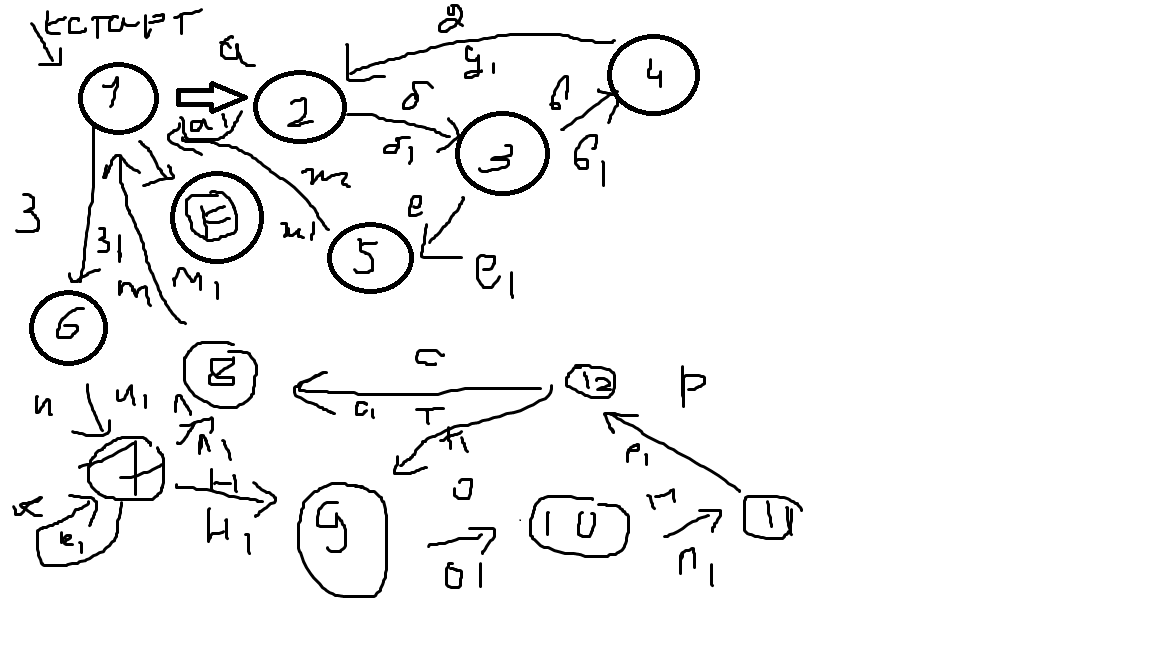
Вводим имя таблицы, к которой строим запрос, выводятся поля, при щелчке добавляются в Select, если нажимаем ОК, то выполняется запрос, если далее, то добавляется раздел FROM с именем таблицы и слово WHERE щелкая на поле, добавляем в WHERE как условие, выбирая операцию сравнения, добавляет WHERE , в 6 вводится значение для сравнение. Если ОК, то выполняем, если далее то сложное условие (or/and)



**19.03.2019 Лекция**

1. Выбор режима
2. Выбор таблицы для запроса
3. Связь с физическими данными
4. Сообщение об ошибке
5. Вывод результата запроса

F - финиш



А – щелчок на пункте меню «таблица»

А1 очистка и отображение текстового поля

Б – нажатие кнопки «ОК»

Б1 – отправка запроса в субд

В -

В1 вывод сообщения об ошибке на экран

Д - нажатие кнопки окей

Д1 – закрытие сообщения об ошибке. Очистка текстового поля

Е –

Е1 -Выполнение запроса к БД .Отображение результатов запроса

Ж – нажатие кнопки «ОК»

Ж1 – отображение первоначального окна приложения

З – нажатие пункта меню «Запрос»

З1 – Очистка текстового поля

И – нажатие кнопки далее

И1 – Добавление раздела select в запрос

К – двойной щелчок по полю в многострочном списке

К1 – добавление поля таблицы в раздел select

Л – нажатие кнопки «ОК»

Л1 – добавление раздела from в запрос. Выполнение запроса

М – нажатие кнопки «ОК»

М1 – возврат в начальное окно приложения

Н – Нажатие кнопки далее

Н1 – добавление раздела from в запрос и раздела where

О – Нажатие на поле таблицы в многострочном списке

О1 – добавление поля в раздел where

П – выбор знака сравнения в выпадающем списке

П1 – добавление знака в раздел where

Р – Ввод значения для условия. Нажатие кнопки далее

Р1 – Добавление значения в условие раздела where

С – нажатие кнопки «ОК»

С1 – попытка выполнить запрос в СУБД.

Т – нажатие кнопки далее. Выбор логического условия для раздела where

Т1 - добавление логической операции в раздел where

Состояния *старт* и *финиш* необходимы для указания точек входа и выхода из приложения

*Старт* – запуск приложения(первое окно с набором окон)

*Финиш*  - уничтожение адресного пространства, разрушение приложения

Такие вершины графа содержатся в графе логике диалога всегда независимо от количества и смысла других состояний.

Для проектирования графа и возможности далее реализации алгоритма работы логики диалога создаются 3 таблицы:

1. Таблица контекста состояний

Отображает для каждого состояния графа набор элементов(окон приложений, которые доступны пользователю для организации диалога). Обозначения следующие:

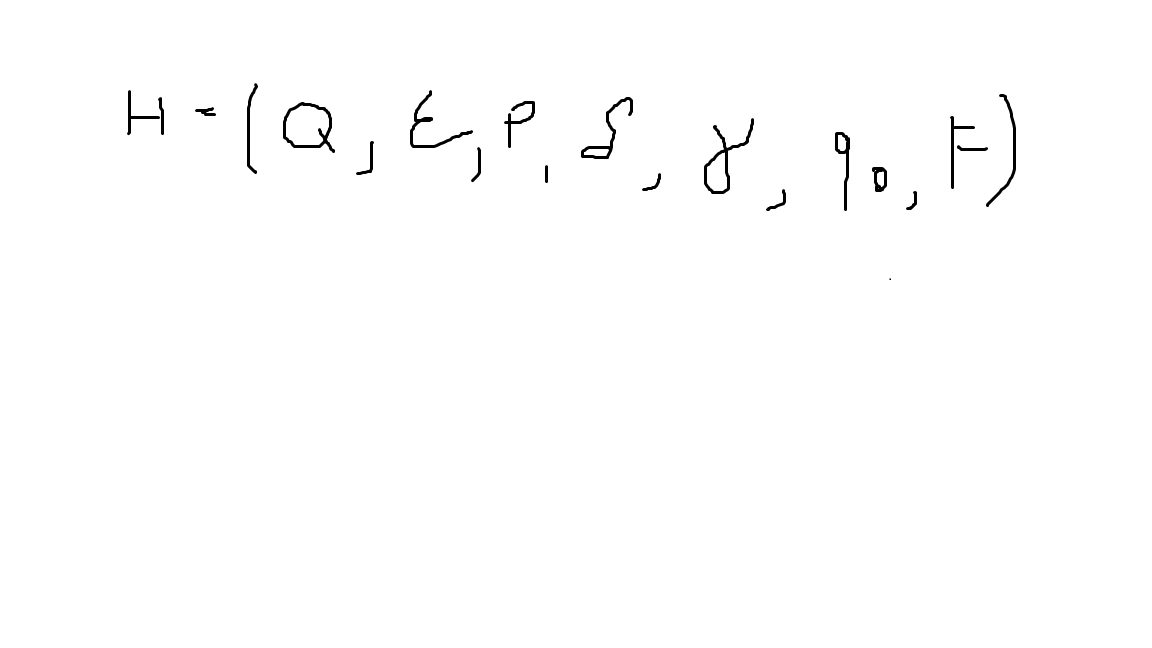
Если в ячейке таблицы стоит + - элемент виден и доступен в окне приложения.

Если стоит «-» - элемент не доступен для пользователей(или пользователь не может с ним взаимодействовать) виден.

Если ячейка остается пустой - элемент не виден в данном состоянии и в окне не отображается.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Справа объект(окно)  Внизу состояние | Меню | Таблица для отображения данных | Список с полями таблицы | Поле для ввода значения | Список с операциями сравнения | Поле для ввода имени таблицы | Многострочное текстовое поле для текста запроса | Кнопка окей | Кнопка далее | Сообщение об ошибке |
| Start |  |  |  |  |  |  |  |  |  |  |
| 1 | + |  |  |  |  |  |  |  |  |  |
| 2 | - |  | - |  |  |  |  | + |  |  |
| 3 |  |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  | + |  |  |
| 5 |  | - |  |  |  |  |  | + |  |  |
| 6 |  |  |  |  |  |  |  |  |  |  |
| 7 |  |  |  |  |  |  |  | + |  |  |
| 8 |  | - |  |  |  |  |  | + |  |  |
| 9 |  |  |  |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |  |  |
| 11 |  |  |  |  |  |  |  |  |  |  |
| 12 |  |  |  |  |  |  |  | + |  |  |
| Finish |  |  |  |  |  |  |  |  |  |  |

Простая транзитивная сеть реализуется с помощью конечного автомата, который имеет свой математический аппарат и в транзитивной сети он задается 7



1 Q - Множество состояний транзитивной сети

2 E Сигма– множество входных сигналов логики диалога

3 P – множество команд(действия пользователя) или прикладных процедур

4 Лямбда – функция переходов, которая отражает закон E\*Q -> Q, где множество входных сигналов\* множество приводит к

1. Гамма – функция прикладных действий, которая отражает закон Множество входных сигналов на множество состояний есть множество обратных E\*Q -> P
2. Начальное состояние графа, для которого выполняется условие q0 e Q
3. F – множество конечных состояний(если речь идет о подсетях и рекурсивных транзитивных сетях) причем F является подмножеством Q

Функция переходов и функция прикладных действий также задаются в виде таблиц для реализации алгоритма логики диалога. Для этого создаются таблицы состояний и таблица перехода

1. Таблица состояний

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Справа Q  Внизу E(то, что над стрелками) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| А |  |  |  |  |  |  |  |  |  |  |  |  |
| Б |  |  |  |  |  |  |  |  |  |  |  |  |
| В |  |  |  |  |  |  |  |  |  |  |  |  |
| Г |  |  |  |  |  |  |  |  |  |  |  |  |
| Д |  |  |  |  |  |  |  |  |  |  |  |  |
| Е |  |  |  |  |  |  |  |  |  |  |  |  |
| Ж |  |  |  |  |  |  |  |  |  |  |  |  |
| З |  |  |  |  |  |  |  |  |  |  |  |  |
| И |  |  |  |  |  |  |  |  |  |  |  |  |
| К |  |  |  |  |  |  |  |  |  |  |  |  |
| Л |  |  |  |  |  |  |  |  |  |  |  |  |
| М |  |  |  |  |  |  |  |  |  |  |  |  |
| Н |  |  |  |  |  |  |  |  |  |  |  |  |
| О |  |  |  |  |  |  |  |  |  |  |  |  |
| П |  |  |  |  |  |  |  |  |  |  |  |  |
| Р |  |  |  |  |  |  |  |  |  |  |  |  |
| С |  |  |  |  |  |  |  |  |  |  |  |  |
| Т |  |  |  |  |  |  |  |  |  |  |  |  |

Построчно действия над стрелками,

На пересечении то, что под стрелками

В состоянии Q при выполнении действия Е(сигма) система выполняет действие P

1. Таблица переходов(таблица действий)

Выглядит аналогичным образом, как и таблица состояний

В столбцах состояния Q(1, 2, .. 12)

По строкам те же действия Е(сигма) (а, б, …)

На пересечении Q

В ячейках, которые заполнены в таблице состояний действием Р, в таблице переходов будет стоять номер состояния, в который мы переходим при совершении этого действия. Т.е. если Q = 1, а Е = пункт меню таблица, то на пересечении их в таблице **2** пишем очистка поля для ввода имени таблицы, в таблице **3** пишем 2.

**Должны быть заполнены одинаковые ячейки в таблицах 2 и 3**

Работа алгоритма логики диалога сводится к следующему:

1. Выявляем входной сигнал или семантическую обратную связь
2. По таблице действий находим (в зависимости от входного сигнала и текущего состояния) необходимое действие и выполняем его
3. Изменяем состояние системы согласно таблице переходов в зависимости от входного сигнала и текущего состояния
4. Устанавливаем контекст нового состояния в соответствии с таблицей контекста состояния

Достоинства простой транзитивной сети:

1. Точное *формальное* описание логики диалога
2. На ЯП диалог реализуется в простой табличной форме
3. Такая реализация диалога легко редактируется и дополняется.

При этом выделяют один главный недостаток, который пытаются устранить до сих пор:

— Если диалог спроектированного приложения реализуется в последовательном виде (не разветвлен), то реализация такого диалога с помощью транзитивной сети становится крайне нецелесообразной.

Данный недостаток пытаются устранить с помощью усовершенствования простой транзитивной сети – рекурсивная транзитивная сеть. С помощью этого же механизма устраняется и другой недостаток простой транзитивной сети:

— Если учитывать, что граф логики диалога представляет собой конечный автомат, то система помнит только 1 текущее состояние, забывая об остальных предыдущих. Т.е. нельзя прервать порядок следования и выполнения диалога, выполнить какое-либо отстраненное действие и вернуться обратно к диалогу. Транзитивная сеть такого действия не позволяет.

Рекурсивная ТС, в отличие от простой сети, устраняет данный недостаток наличием стека, т.е. сохраняется текущее состояние, выполняется отстраненное действие и возвращаемся.

**Принципы функционирования направленных и ненаправленных графов(принципы конечного автомата)**

**Рекурсивная ТС**

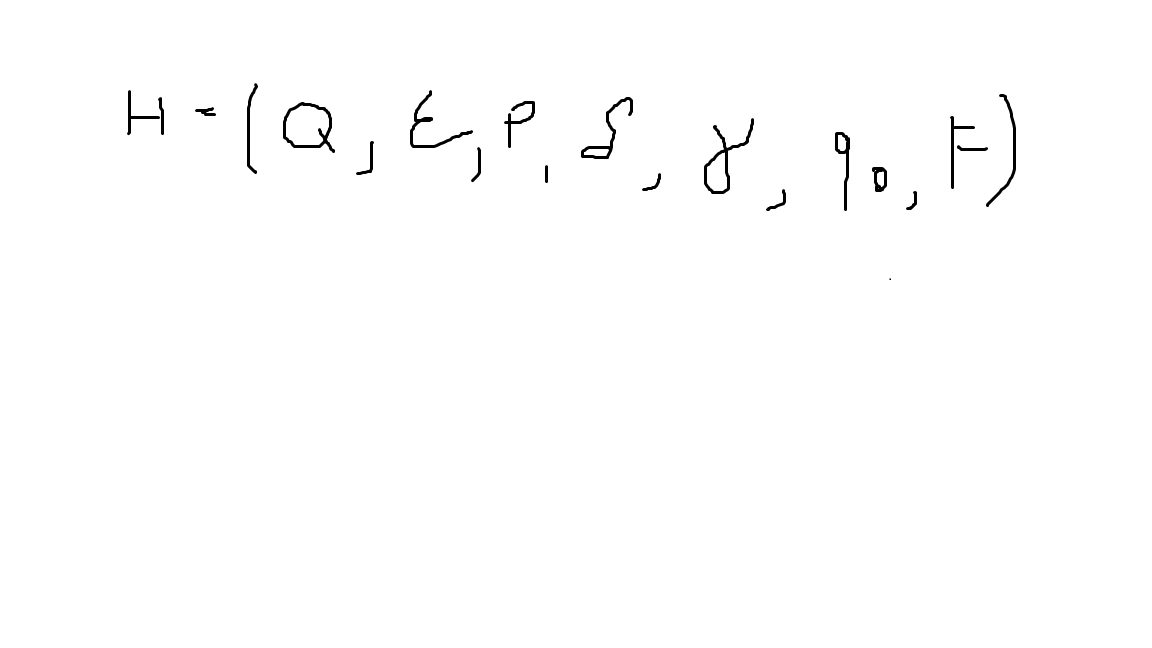
Основным признаком необходимости построения рекурсивной ТС является наличие подсетей. Для подсети х-ны все признаки (как по построению граф части, так и по функциональному отображению). Отличие состоит в том, что в подсеть можно попасть из состояний основной сети ( из различных состояний и сколько угодно раз), а затем вернуться в то состояние, которое явилось точкой входа в подсеть.

Рекурсивную ТС рассматриваем на том же примере по работе с БД, допуская, что в этом приложении должен иметься архив, в который попадают все правильно построенные и выполненные запросы SQL (условие выполняемости является признаком занесения в архив).

Помимо этого, архив должен предоставлять возможность просматривать архивированные запросы и повторно выполнять выбранный запрос.

Если рассматривать наш пример, то запросы попадают в архив из **состояний 5 и 8 основной сети,** а просмотреть архив и выполнить архивированный запрос можно из любого состояния основной сети.

Реализуется диалог, построенный в виде рекурсивной ТС с помощью стекового конечного автомата, характерным признаком которого является наличие стека состояний. В данный стек состояния заносятся при входе в подсеть, а при выходе из него извлекаются из стека. Математически стековый конечный автомат реализуется в виде следующей девятки.



Вместо двух последних z0 и Г

1. Q - Множество состояний транзитивной сети
2. E Сигма– множество входных сигналов логики диалога
3. P – множество команд(действия пользователя) или прикладных процедур
4. Лямбда – функция переходов, которая отражает закон E\*Q -> Q, где множество входных сигналов\* множество приводит к
5. Гамма – функция прикладных действий, которая отражает закон Множество входных сигналов на множество состояний есть множество обратных E\*Q -> P
6. Начальное состояние графа, для которого выполняется условие q0 e Q
7. Z0 – начальное состояние стека
8. Г – множество стековых состояний( /элементов стека)

Все законы, действенные для простой ТС, действуют и в рекурсивной за исключением того, что здесь появляется новый закон z0 принадлежит Г

На любой Т подсети есть

Подсеть также является однонаправленным графом и переходы подписываем также, как и на основной сети.

Но для правильного перехода из подсети в сеть добавляется обозначение подсети на осн сети.

1)Прямоугольник с кругленными углами – состояния, из которых переходим в подсеть(5 и 8 состояния)

2)когда в архиве можем просматривать и выполняться запросы( из любых состояний)

Допускается при условии, что в подсеть можно попасть из всех ( или из множества)состояний сети не переводить их обозначения в прямоугольник с скругленными углами. Особенно это касается больших сетей, где к-во состояний насчитывается не 1 десяток.

При этом, весь механизм входа и выхода для подсети никоем образом не изменяется.

**6 лаба**

Тема – рекурсивная ТС

В отчете листинг, графическое и табличная реализация РТС

Алгоритм реализации

Для реализации РТС с исп-ем стекового конечного автомата действует тот же алгоритм изменения контекста перехода из состояния в состояние, что и для простой ТС. Соответственно, реализуются теже таблицы для реализации интерфейса, что и для ПТС. 1 отличие в том, что для подсетей не строятся отдельные таблицы, а таблицы для основной сети просто дополняются состояниями, контекстом, и функцией действия для всех подсетей РТС

**Проектирование конкурентного диалога**

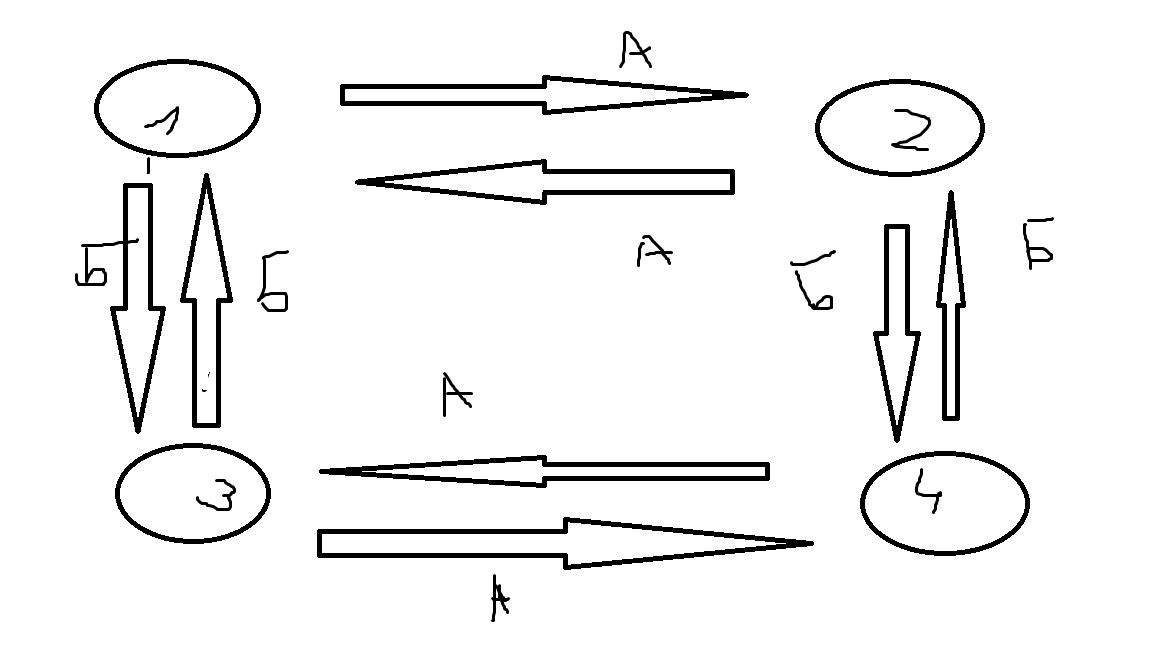
Работу любой ТС всегда можно рассматривать как передвижение единственной метки из одного состояния в другое(другие), обозначая тем самым, что в каждый момент времени система может находиться в 1 единственном состоянии. Поэтому, главным недостатком использования сетей переходов (как простых, так и рекурсивных) является слабая возможность отображения и построения конкурентного диалога. Конкурентный диалог характерен для систем, которые можно описать более чем одним состоянием одновременно. Наиболее часто конкурентный диалог возникает при проектировании различных настроек в приложении или переключателей в ИС.

Если рассматривать конкурентный диалог с математической точки зрения, то его модель описывает возможность перехода по 1 и тому же действию в различные состояния.

Для примера проектирования конкурентного диалога возьмём 2 переключателя (жирный и курсив как в ворде). Оба переключателя реализованы в одном приложении, при этом нажатие на каждый переключатель переводит его в противоположное состояние. В этом случае, если переключатель в приложении один, то для описания его диалога требуются 2 состояния. Если переключателей 2, то 4 состояния, если 3 переключателя то 8

Закон определения состояний для элементов конкурентного диалога это 2^n.

Если рассматривать данный диалог в виде сети Петри, то



Состояния

1 курсив выключен, жирный выключен

2 курсив выключен, жирный включен

3 курсив включен, жирный выключен

4 курсив включен, жирный включен

Наискосок нельзя, только через 2 действия

Переходы

А нажатие на переключатель жирный

Б – нажатие на кнопку курсив

Это отображение интерфейса транзитивной сетью.

Проблема в большом количестве дублирующей информации

Для избежания этого недостатка ТС стали применять сети Петри, на которых факторы избыточности состояний и переходов сведен к минимуму.

При использовании ТС для проектирования конкурентного диалога через набор конечных автоматов, необходимо не только следить за правильностью всех переходов в сети и выделения достаточного количества состояний, чтобы учесть все изменения интерфейса, но и необходимо иметь дополнительный механизм определения, к какому конечному автомату будет относиться входной сигнал или обратная семантическая связь, а также, помимо этого, нужно иметь новый механизм для формирования общего для всех контекстов интерфейса.

**Сети Петри**

Сеть Петри, как механизм для проектирования конкурентного диалога, разрабатывалась с учетом того, чтобы устранить недостатки ТС. Неформально сеть Петри можно представить как перемещение многих меток по графу состояний, каждая метка при этом относится к одному конечному автомату.

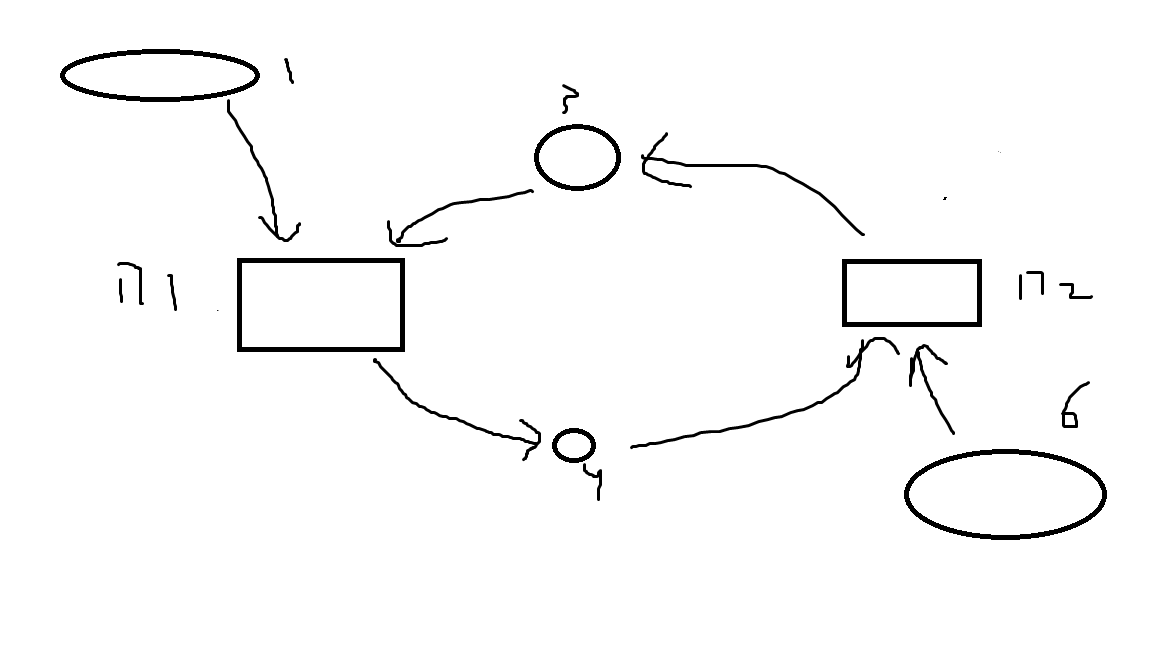
Основные элементы, которые используются на сетях Петри

1)Состояния обозначаются также кружками, надпись для состояния ставится рядом с кружком(не внутри)

2)Переходы как прямоугольники, обозначения рядом, не внутри

3) Действия пользователя, обозначаются овалом, подпись рядом

Окружности и эллипсы могут содержать метки, меток в переходах быть не может. Переходы на сетях Петри действуют как триггеры, которые определяют, возможен ли переход в состояние или нет.



Сеть Петри для переключателей

Подобная сеть отображает функционирование 1 переключателя

3 – курсив включен

4 – курсив выключен

П1 переход 1

П2 – переход 2

1, 6 – нажатие на переключатель курсива

В случае 2 переключателей одно состояние два описания

**Правила работы сети Петри**

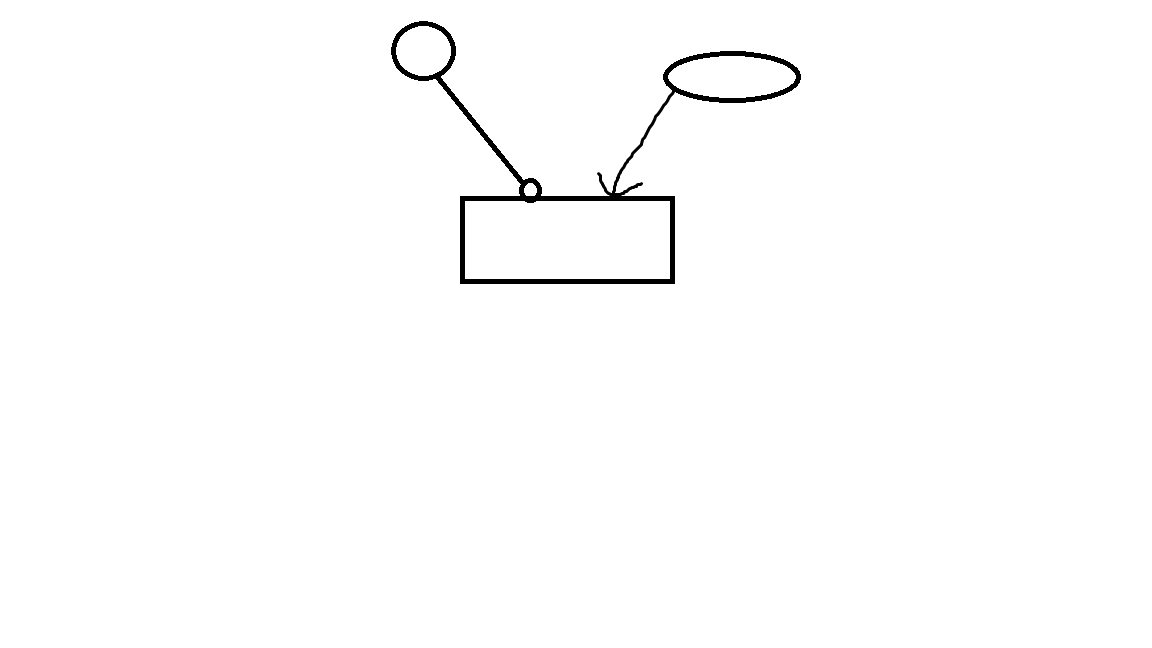
1)Если все вершины на входных дугах перехода содержат метки, то переход срабатывает, метки объединяются и одна метка перемещается в вершину выходной дуги перехода

**Ограничения в сетях Петри**

Как и все методики проектирования интерфейса, сети Петри имеют свои ограничения. Хотя, для проектирования конкурентного диалога данные ограничения обыгрываются на сетях Петри отдельными условными обозначениями.

Так, сети Петри могут содержать дополнительные входные дуги для ограничения действий переходов, которые оканчиваются не стрелкой, а окружностью.

Графически, выглядит след образом



Такая дуга означает, что переход не выполняется и не срабатывает как триггер, если вершина сети Петри соединенная этой дугой с переходом содержит метку. В конкурентном диалоге такая ситуация возможна, если действие с одним переключателем/элементом влияет на функциональность другого.

Пример, если в меню выбран режим таблицы, то это влияет на видимость и доступность элементов, связанных с формированием условия в запросе SQL.

Или, вывод списка городов конкретной выбранной страны. В этом случае, сеть Петри:

Состояния

1 – отображается окно со списком городов

2 – окно со списком стран видимо

3 – окно со списком городов скрыто

4 - окно со списком стран недоступно

А нажата кнопка скрыть

Б нажата кнопка показать

В нажата кнопка показать

Г нажата кнопка скрыть

1-> n3 –ограничивает возможность показа выбора городов, если не выбрана страна (не может выбрать город пока не выберем страну)

4 -> n2 – не дает возможность сменить страну, если отображены списки городов предыдущей страны (нужно скрыть список городов чтобы открыть список стран)

**Ставим метку в начальное состояние — старт**

**Финиш – действие, приводящее к состоянию выхода**

**7 ЛАБА**

Тема – многопоточное программирование. Проектирование интерфейса с использованием сети петри

Задание – реализовать ВП которое организовывает работу 2 потоков в процессе, первый из которых отвечает за бегущую строку в полосе статуса виндовс окна. Второй поток отвечает за последовательную загрузку изображений в главную(рабочую) область окна. Каждый поток должна останавливать одна кнопка и с помощью нее возобновлять поток. Для данного приложения реализовать интерфейс с помощью сети Петри как в графе, так и в коде. Предоставить граф и листинг.

Минимум 15 состояний

**Текстовое представление пользовательского интерфейса**

Текстовое представление пользовательского диалога в литературе представляется 2 мат. аппаратами:

1)BNF – выражения

2)Регулярные выражения

Основным отличием этих форм от сетей переходов является то, что внимание фокусируется не на состояниях системы или приложения, а на действиях пользователя.

BNF выражения известны по описанию обычных ЯП, тк основные элементы диалога это последовательность и выбор. Фактически, циклические действия представляются также в виде последовательности.

Пример описания диалога в виде BNF-выражения

Высказывание ::= пустая последовательность

| слово высказывания

| ‘(’высказывание’)’ высказывание (повторяющиеся высказывания сколько угодно раз)

Применительно к пользовательскому диалогу, любая BNF форма фактически представляет собой выбор действие(совершенное или нет).

В отличие от BNF выражений, регулярные выражения(последовательность символов) более просты и менее регламентированы. Элементы, которые не меняются в регулярных выражениях:

1)последовательность символов

2)спец символы(оператор выбора(|), звезда \* обозначение цикличности, ‘’ – неизменная последовательность символов, которая воспринимается так, как она записана в выражении) которые управляют

Применение звезды Клини (\*) означает, что последовательность, приведенная до \* повторяется 0 или сколько угодно много раз(пример(…)\*).

Данный инструментарий регулярных выражений позволяет их применять широко в текстовых редакторах для описания множества критериев отбора и поиска, а также в лексических анализаторах ЯП.

Эти две формы (BNF и регулярные) не способны проектировать конкурентный диалог.

**Продукционные правила**

Продукционные правила представляют собой особый вид регулярных выражений, разработанных с учетом требований разработки различных видов диалогов. Основная структура, которая описывает любой диалог представляется в форме условного выражения

If(you) then (действие)

Или если брать запись продукционного правила в нотации регулярных выражений, то эта запись будет иметь вид

Условие ->действие(вместо стрелки может встречаться ::)

Все правила, описывающие диалог, равноправны, и система подбирает условную часть правила, в зависимости от события, которое было порождено в сфере интерфейса(т.е. действие пользователя или системы, которое технически занимает память при выполнении, а следовательно меняет интерфейс)

Если меняются состояния без действия, то они не описываются в правилах.

**Действие = смена состояния. Если его нет, то нет правила описания.**

Когда условная часть правила становится истинной, правило срабатывает и выполняется действительная часть правила. Под действием здесь можно понимать как вид ответа пользователю, или изменение самого приложения(интерфейсной части).

На основе этого, выделяют 3 вида продукционных правил:

1)продукционные правила, основанные на событиях

2) продукционные правила, ориентированные на состояния

3) смешанные продукционные правила

**Продукционные правила, основанные на событиях**

Пример интерфейса приложения, которое с помощью мыши рисует ломаную кривую с эффектом резиновой нити. Основные правила, которые принимаются к определению продуционных правил, заключаются в след:

1)События могут быть 3 видов:

1.1 Пользовательские события.

В правилах записываются с заглавной буквы(фактически, это действия пользователя, не включая обратную семантическую связь)

1.2Внутренние события,

которые в правилах записываются с прописной буквы и используются для сохранения состояния пользовательского диалога(контекста диалога)

1.3 События отклика системы(обратная семантическая связь)в правилах продукционных указывается в угловых скобках и имеют вид в приложении, показывая изменения пользовательского интерфейса.

Продукционные правила, основанные на событиях, будут выглядеть след образом:

*Выбрать\_линию -> начать\_линию(отклик системы) <высвечивается\_”линия”>*

*Клик\_мыши начать\_линию -> неизменная\_линия <резиновая\_нить\_включена>*

*Щелчок неизменная\_линия -> неизменная\_линия <рисовать\_линию>*

*Двойной\_щелчок неизменная\_линия -><отрисоватьлинию><резиновая\_нить\_отключена>*

Данные правила отрисовывают только одну часть ломаной кривой с использованием эффекта резиновости.

Блок управления подобным диалогом, который поддерживает продукционные правила, ориентированные на события, должен иметь память, состоящую из множества событий.

Правила срабатывают в том случае, если все события условной части находятся в памяти, все события пользователя неизменно заносятся в память, а система откликается на действия пользователя путем порождения собственных событий.

Когда правило срабатывает, все события, указанные в условной части, удаляются из памяти, а все события действительной части заносятся туда.

Правила, ориентированные на события, всегда ведут к добавлению и удалению большого числа событий, что сильно затрудняет хранение статической информации о пользовательском интерфейсе. Это приводит к ситуации, когда событие присутствует и в условной и в действительной части правила, и сильно загружают оперативную память.

**Продукционные правила, ориентированные на состояния**

В системе правил, ориентированной на состояния, память представляет собой множество именованных значений. Отличие от правил, ориентированных на события, заключается в том, что при выполнении правила элементы не удаляются из памяти.

В подобных правилах, состояние представляет собой множество именованных атрибутов, которые в целом характеризуют UI в текущий момент времени.

Каждый атрибут может принимать в таких правилах только 1 значение, при этом одни из атрибутов представляют собой действия пользователя, другие – работу системы.

Относительно примера с рисованием резиновых нитей, можно выделить следующие 5 атрибутов и их значений:

Атрибут записывается перед фигурными скобками, его значения через запятую в фигурных скобках

Мышь{}

Резиновая нить{включена, выключена}

Меню {ничего не выбрано, выбрана линия}

Рисование{ничего не рисуется, рисуется линия}

1 атрибут Характеризует действия пользователя

2

3 4 5 – отклик системы

На основе данных атрибутов, правила имеют след вид:

1)*Выбор\_из\_меню ->мышь\_не\_работает, начало, высвечивается\_линия*

*2)Щелчок, начало ->мышь\_не\_работает, незменная\_линия, включена*

*3)Щелчок, неизменная\_линия -> мышь\_не\_работает, рисуется\_линия*

*4)Двойной\_щелчок, неизменная\_линия -> мышь\_не\_работает, рисуется\_линия, выключена(резиновость)*

Механизм работы продукционных правил следующий:

Правила выполняются тогда, когда состояния системы соответствует его условной части. Под состоянием в данном случае понимается множество атрибутов с их текущим значением. Выполнение правила не обязательно приводит к смене значений одного или нескольких атрибутов. Происходит это только в том случае, если действие правила содержит новое значение атрибутов.

Например, 2 правило срабатывает, если атрибут *мышь* имеет значение *щелчок*, а атрибут *состояние\_линии* значение *начала* при любых других значениях остальных атрибутов. Выполнение этого правила приведет к изменению атрибутов *мышь, состояние\_линии, резиновая\_нить*.

Атрибуты в системе правил, ориентированных на состояния, достаточно устойчивые, но бывают исключения. В приведенном примере, это атрибут *мышь*. Очень часто она присутствует в обеих частях правила, т.е. мышь меняет свое значение при выполнении каждого правила. С другой стороны, значение этого атрибута изменяется только пользователем, что и приводит к выполнению правила.

**Смешанные продукционные правила**

Классических подход к проектированию UI на основе продукционных правил, требует применения какой-то одной модели: либо модели, основанной на событиях, либо на состояниях. Но бывают ситуации, особенно в интерфейсах ссылочного типа смешанный подход может быть более эффективным, тк он учитывает не только действия или состояния, но и контекст, в которых они происходят.

Данный принцип демонстрирует смешанные продукционные правила, которые имеют вид :

*Событие: условие ->действие*

Т.е. в условной части правила учитывается контекст, в котором происходит условие для выполнения. Правило читается след образом:

Событие можно представить, как сигнал к выполнению правила , но правило не срабатывает, если его условие не истинно. События порождаются пользователем и атрибуты остаются неизменными, пока они не встретятся в действии правила.

Смешанные продукционные правила с подобным подходом могут применяться для проектирования конкурентного диалога.

Пример.

Переключатели шрифтов

3 переключателя

Подчеркивание{включено, выключено}

Курсив{включено, выключено}

Жирный{включено, выключено}

Тк действия 1 переключателя никак не зависят от функциональности или состояния других, выделяем 3 события в системе:

*Переключение\_подчеркивания*

*Переключение\_курсива*

*Переключение\_жирного*

Правила будут иметь след вид:

*Переключение\_курсива: курсив = выключен->курсив=включен*

*Переключение\_курсива: курсив = включен->курсив=выключен*

Для остальных переключателей правила составляются аналогичным образом.

Удобство:

Каждое правило затрагивает какой-либо один атрибут, не нужно учитывать состояние системы в целом (в отличие от сетей Петри).

В смешанных продукционных правилах очень часто условие может опускаться. В этом случае правило имеет вид:

*Событие ->действие*

Пример:

*Нажатие\_на\_кнопку\_закрытия->закрытие\_приложения*

**8 лаба**

Спроектировать и разработать интерфейс игры в крестики нолики

Реализовать 1 из 3 систем правил(на событиях, состояниях, смешанную)

В отчете 3 реализации, мат аппарат.

Вопросы

1)продукционные правила, основанные на событиях.синтаксис, назначение и функционирование

2, 3 для оставшихся двух