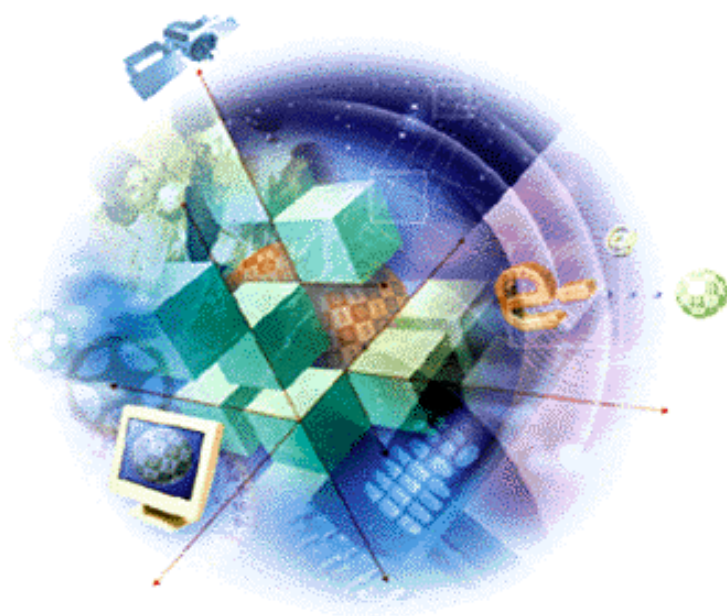


TRANSPORTA UN SAKARU INSTITŪTS



**Разработка приложений
В
PROGRESS 4GL
Version 9**



**Авторы:
Графеева Н. Г.
Томыткина Т. Б.**

**Обработка:
Шамшин Ю. В.**

Рига 2002

Данное пособие предназначено в помощь изучающим курс основ системы PROGRESS.

Предполагается проведение работ в компьютерном классе под руководством преподавателя.

Программа курса начинается со знакомства с базовыми понятиями языка 4GL и основами событийного программирования. Слушатели создают собственные базы данных и используют их в дальнейшем обучении. В состав курса входит знакомство с технологиями программирования, визуальными методами разработки приложений, механизмами поддержки целостности данных, элементами администрирования.

Приложения включают в себя как материалы справочного характера, так и дополнительные главы, встраиваемые в курс по желанию слушателей.

Материалы курса не подлежат коммерческому распространению без разрешения авторов.

Об авторах:

Графеева Наталья Генриховна - доцент кафедры системного программирования математико-механического факультета Санкт-Петербургского Государственного Университета (nat@ngl184.spb.edu)

Помыткина Татьяна Борисовна - ведущий программист отдела программного обеспечения общеуниверситетских служб Санкт-Петербургского Государственного Университета (tb@pom.usg.ru)

Санкт-Петербург 2000

Обработка:

Шамшин Юрий Васильевич – доцент кафедры компьютерных наук и информационных технологий Института Транспорта и Связи (yury@tsi.lv)

Рига 2002

Пособие печатается с разрешения компании «Компьютерные системы для бизнеса» (CSBI EE) (<http://progress.csbi.ru>) для учебных целей и некоммерческого использования.

СОДЕРЖАНИЕ

PROGRESS: КРАТКАЯ СПРАВКА	5
I. PROGRESS 4GL	12
1. СТРУКТУРА ЯЗЫКА.....	12
1.1. Первое знакомство со средами для создания программ.....	12
1.2. Базовые типы данных	13
1.3. Идентификаторы и комментарии	13
1.4. Описания переменных	13
1.5. Операции.....	15
1.6. Стандартные функции	16
1.7. Управляющие операторы	18
1.8. Массивы	19
1.9. Процедуры без параметров и разделяемые переменные	20
1.10. Процедуры и функции с параметрами	21
1.11. PERSISTENT процедуры.....	23
1.12. Системная переменная ERROR-STATUS	25
2. СОБЫТИЙНОЕ ПРОГРАММИРОВАНИЕ И ОБЪЕКТЫ ИНТЕРФЕЙСА	26
2.1. Категории объектов интерфейса.....	26
2.2. Атрибуты	27
2.3. Методы	27
2.4. Объекты интерфейса типа BUTTON.....	27
2.5. Объекты интерфейса типа IMAGE и RECTANGLE	29
2.6. Объекты интерфейса типа FILL- IN	31
2.7. Групповые триггеры	33
2.8. Объекты интерфейса типа TOGGLE-BOX	33
2.9. Объекты интерфейса типа RADIO-SET	35
2.10. Объекты интерфейса типа SLIDER	36
2.11. Объекты интерфейса типа EDITOR	37
2.12. Объекты интерфейса типа SELECTION-LIST	38
2.13. Объекты интерфейса типа COMBO-BOX.....	39
2.14. ActiveX объекты	40
II. БАЗЫ ДАННЫХ.....	42
3. ОРГАНИЗАЦИЯ БАЗ ДАННЫХ.....	42
3.1. Создание базы данных	42
3.2. Создание таблиц	43
3.3. Создание полей.....	44
3.4. Описание индексов	45
3.5. Секвенции	45
3.6. Триггеры базы данных.....	46
4. РАБОТА С ДАННЫМИ.....	47
4.1. Добавление записей к таблице.....	47
4.2. Выборка данных из таблиц	47
4.3. Корректировка записей.....	54
4.4. Удаление записей	57
4.5. Связные таблицы.....	58
4.6. Буферы записей	63
4.7. Триггеры базы данных.....	67
4.8. Рабочие и временные таблицы.....	69
5. ОБЕСПЕЧЕНИЕ ЦЕЛОСТНОСТИ БАЗ ДАННЫХ	71
5.1. Механизмы поддержки целостности баз данных.....	71
5.2. Транзакции.....	71

5.3. Локализация данных в многопользовательских задачах	77
III. ПРОГРАММИРОВАНИЕ ИНТЕРФЕЙСОВ И ОТЧЕТОВ	82
6. РАЗРАБОТКА ИНТЕРФЕЙСОВ	82
6.1. Фреймы	82
6.2. Динамические объекты интерфейса	85
6.3. Окна	88
6.4. Организация меню	90
6.5 HELP - поддержка	93
7. СОЗДАНИЕ ОТЧЕТОВ	95
7.1. Переопределение входного и выходного потоков	95
7.3. RESULTS - интерактивный построитель отчетов	99
8. КОНСТРУИРОВАНИЕ ПРИКЛАДНОЙ ПРОГРАММЫ.....	101
9. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ INTERFACE-TEMPLATE.....	105
9.1. Препроцессор	105
9.2. Создание и использование interface-template.....	108
IV. КОМПОНЕНТНАЯ МОДЕЛЬ РАЗРАБОТКИ ПРИЛОЖЕНИЙ	110
10. СТАНДАРТНЫЕ СРЕДСТВА ADM.....	110
10.1. Идеология создания приложений в ADM	110
10.2. Технология сборки приложений	112
10.3. Использование виртуальных страниц	120
11. АРХИТЕКТУРА ADM.....	123
11.1. Структура SmartObjects	123
11.2. Взаимодействие SmartObjects	124
11.3. Управление записями в ADM	128
V. ЭЛЕМЕНТЫ АДМИНИСТРИРОВАНИЯ.....	131
12. МНОГОТОМНЫЕ БАЗЫ ДАННЫХ.....	131
13. ЗАЩИТА ОТ НЕСАНКЦИОНИРОВАННОГО ДОСТУПА	133
13.1. Connect-security	133
13.2. Compile-security	133
13.3. Runtime-security	134
13.4. Schema-security	135
14. РЕГЛАМЕНТНЫЕ РАБОТЫ.....	136
14.1 Копирование и восстановление баз данных (Backup/Restore)	136
14.2. Поддержка after-imaging	137
ПРИЛОЖЕНИЯ.....	139
ПРИЛОЖЕНИЕ 1. Запуск и снятие сеансов Progress, стартовые параметры	139
ПРИЛОЖЕНИЕ 2. Способы открытия (connect) и закрытия (disconnect) баз данных	139
ПРИЛОЖЕНИЕ 3. Форматы данных	140
ПРИЛОЖЕНИЕ 4. Цветовая настройка графической среды	140
ПРИЛОЖЕНИЕ 5. Файлы базы данных PROGRESS	141
ПРИЛОЖЕНИЕ 6. Разделители слов в WORD-индексах	141
ПРИЛОЖЕНИЕ 7. Мета-схема базы данных	141
ПРИЛОЖЕНИЕ 8. Структура генерируемых в АВ программ	142
ПРИЛОЖЕНИЕ 9. Работа со шрифтами	142
ПРИЛОЖЕНИЕ 10. Компиляция приложений.....	143
ПРИЛОЖЕНИЕ 11. Операции и функции допустимые в препроцессорных выражениях	143
ПРИЛОЖЕНИЕ 12. Организация библиотек	144
ПРИЛОЖЕНИЕ А. Progress/SQL	145
ПРИЛОЖЕНИЕ В. Report Builder.....	153
ПРИЛОЖЕНИЕ С. Вопрос - ответ	162

PROGRESS: КРАТКАЯ СПРАВКА

В компьютерном мире под словом PROGRESS обычно подразумевают семейство продуктов **Progress Software Corporation**. Основанная в 1981 году, со штаб-квартирой в городе Bedford (USA), эта корпорация имеет свыше 1100 сотрудников, работающих в 49 представительствах по всему миру. Еще в 60-ти странах Progress представлен компаниями-дистрибьюторами. В странах СНГ интересы корпорации представляет компания **CSBI EE** (Санкт-Петербург, Москва).

Progress Software Corp. является одним из крупнейших поставщиков средств для построения информационных систем масштаба промышленного предприятия, ориентированных на интенсивную обработку транзакций в распределенных вычислительных средах в реальном масштабе времени.

Приложения, разработанные на Progress, могут работать без переписывания на более чем 160-ти программно-аппаратных платформах, включая Windows-NT, Novell, Unix и AS/400.

Progress Software Corp. предлагает средства разработки приложений в архитектуре клиент/сервер (более 10,000 одновременных пользователей, размер базы данных до 80,000 ТБайт), а также на основе Web-технологий, предназначенных для транзакционной обработки в среде Internet/Intranet/Extranet.

Современные технологии, предлагаемые Progress Software Corp. позволяют создавать приложения для N-уровневой архитектуры, поддерживать Java-хранимые процедуры, реализовывать "открытый" сервер приложений - универсальное средство для построения гетерогенных кроссплатформенных приложений, тесно интегрированных с Internet.

Последние инициативы Progress: ASPEN — Application Service Provider ENabler («продвижение технологий для сдаваемых в аренду приложений») и UAA — Universal Application Architecture («универсальная архитектура приложений»). Для реализации этих инициатив избран эволюционный путь: постепенное объединение всех серверных продуктов в один.

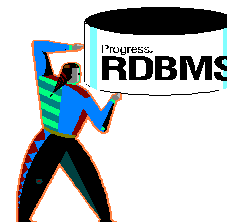
Компания Progress Software заявляет, что инструменты, необходимые для построения информационных систем начала третьего тысячелетия, или уже есть в ее распоряжении, или на подходе.

ВВЕДЕНИЕ

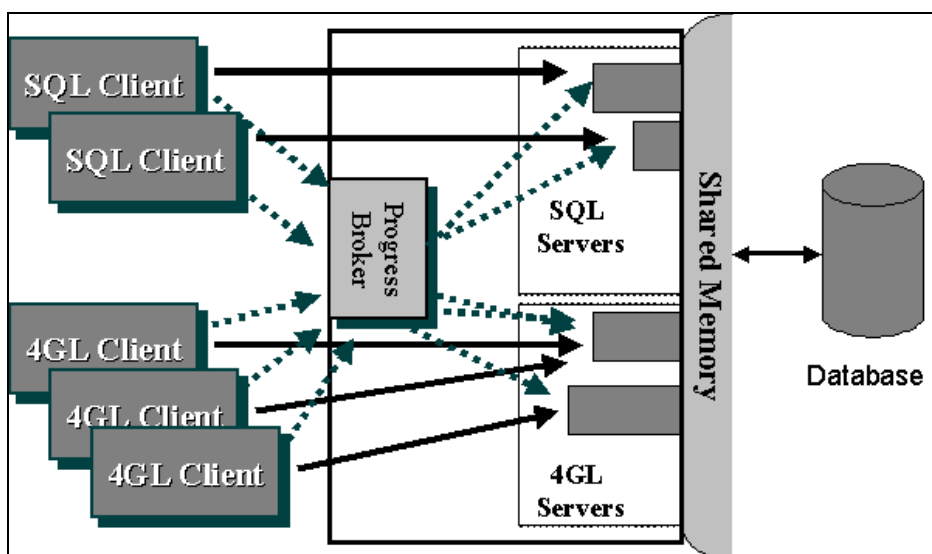
1.

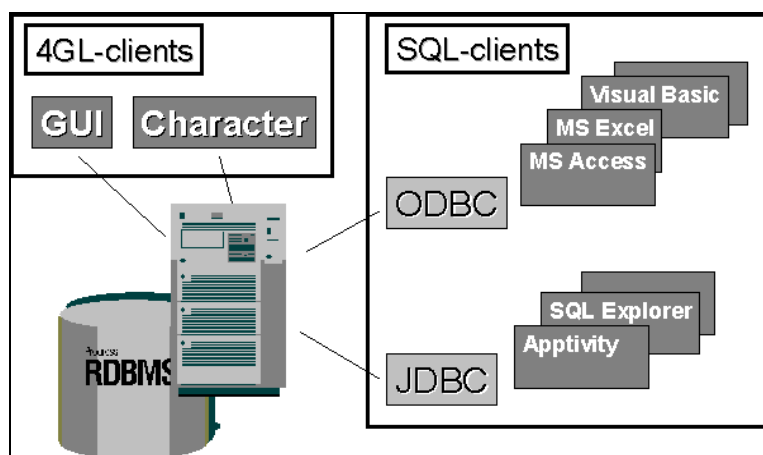
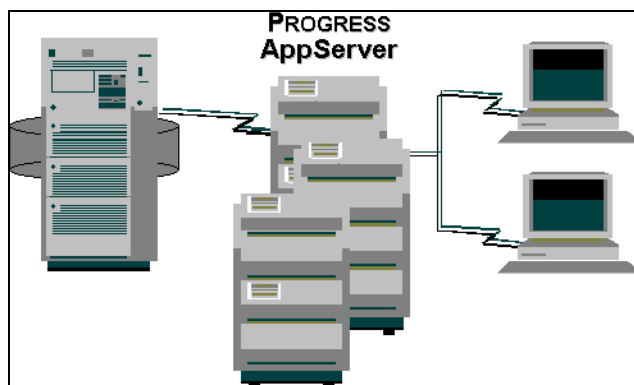
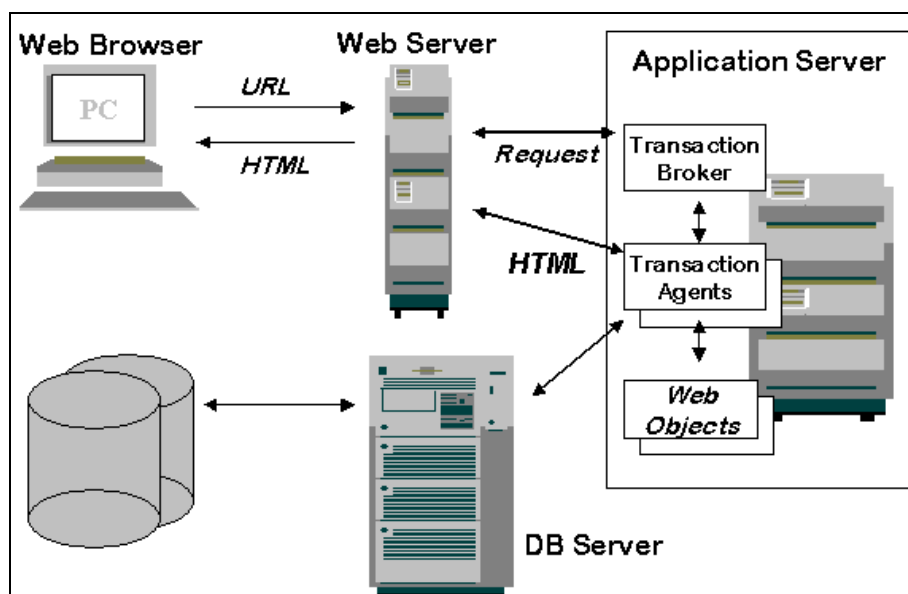
Основные характеристики СУБД Progress:

- БД большого объема (до 80 000 Тб)
- Многопользовательский режим работы с данными
- Возможность on-line администрирования для критических БД
- Поддержка промышленных стандартов:
 - операционные системы
 - сетевые протоколы
 - пользовательские интерфейсы
 - SQL ANSI92, ODBC, JDBC
- Переносимость приложений между платформами
- Поддержка физической и логической целостности на уровне БД
- Поддержка распределенных БД
- Гибкие возможности по организации распределенной обработки данных:
 - клиент/сервер
 - N-уровневая архитектура
 - Web-архитектура



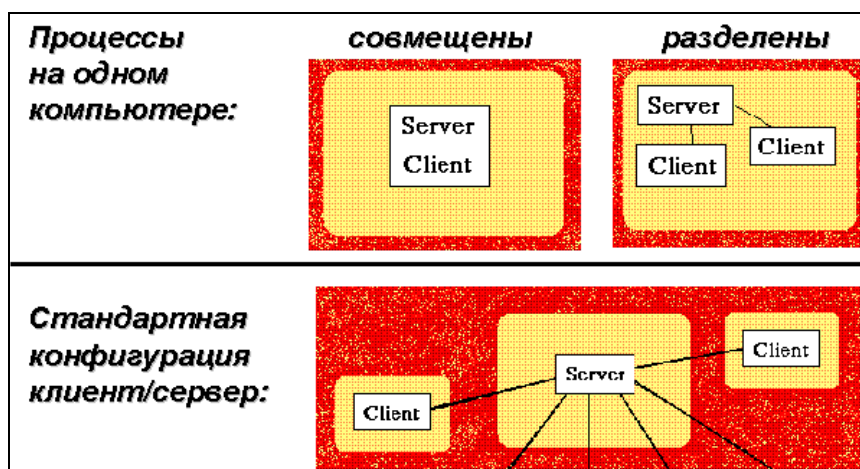
2.

Архитектура клиент/сервер:

3.**Клиенты Progress:****4.****N-уровневая архитектура:****5.****Web – архитектура:**

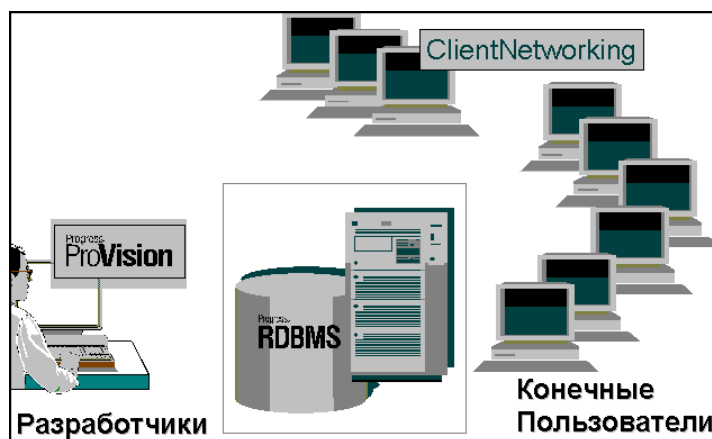
6.

Простейшие варианты конфигурации клиент/сервер:



7.

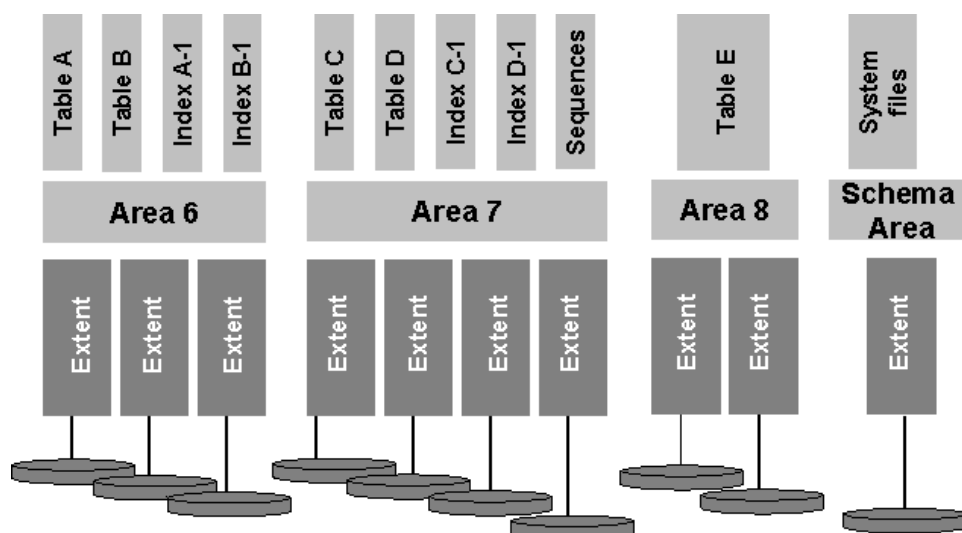
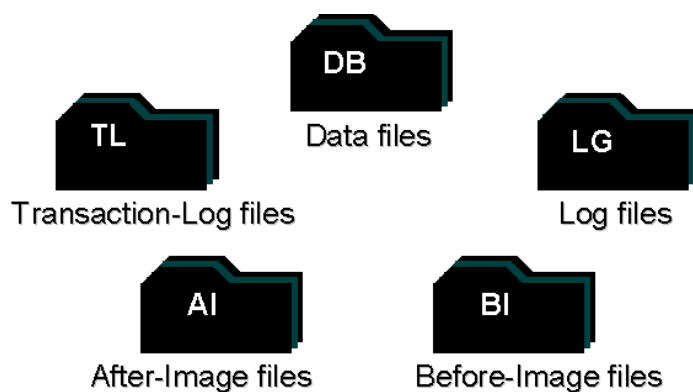
Категории клиентов:

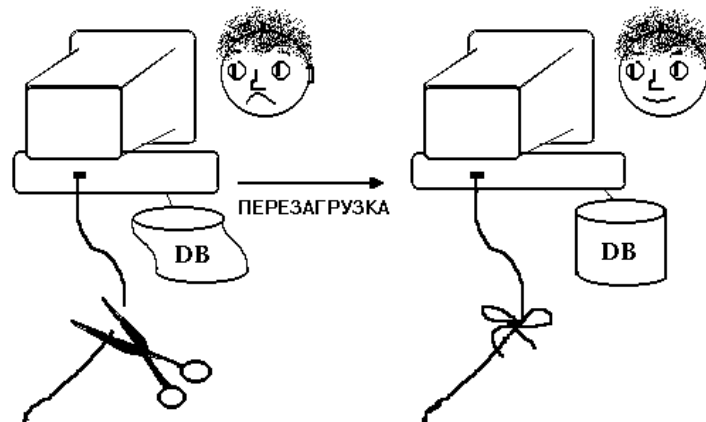
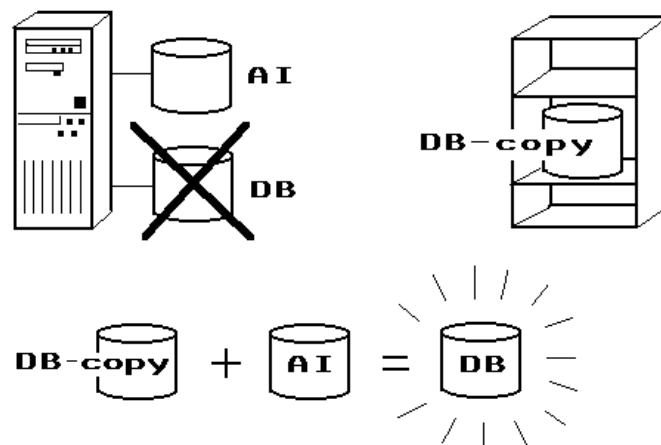
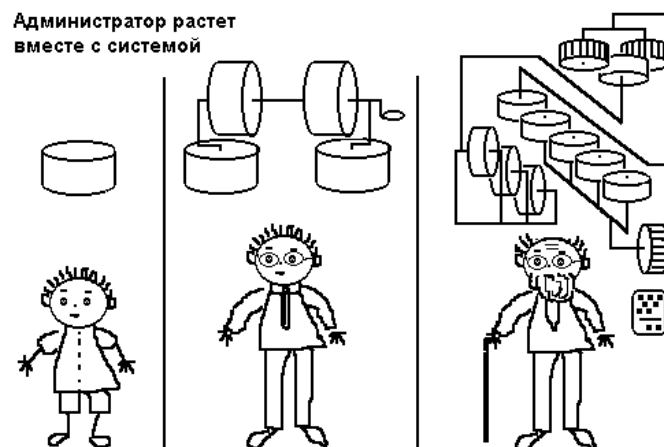


8.

Многопользовательская работа с данными:

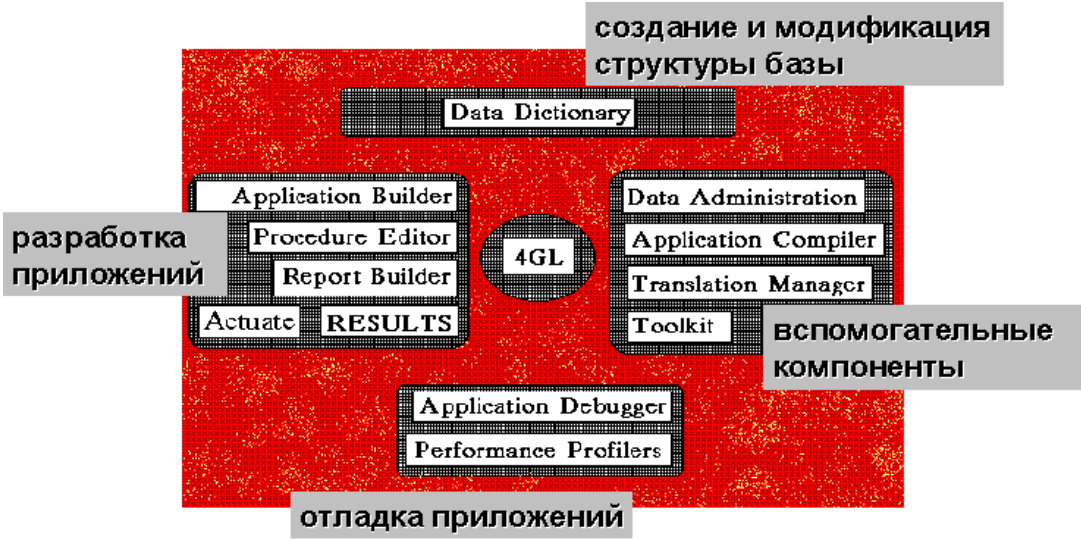


9.**Логические объекты базы данных:****10.****Физическая организация базы данных:****11.****Файлы базы данных:**

12.*Механизм откатов (before imaging):***13.***Механизм накатов (after imaging):***14.***Администрирование:*

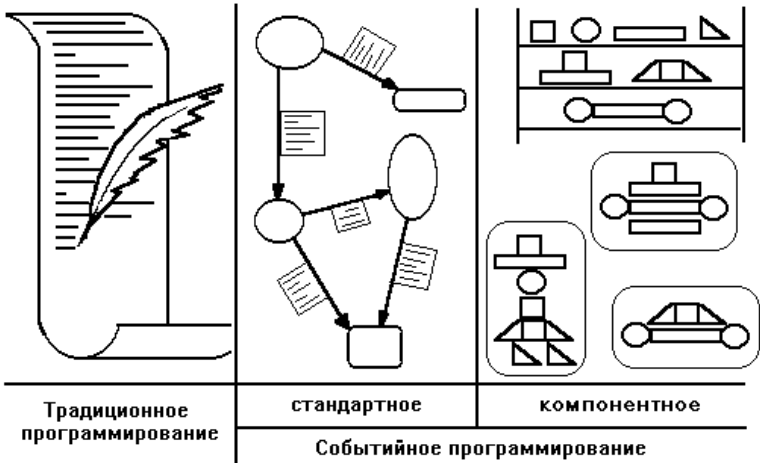
15. —————

Среда Разработчика (ADE):



16. —————

Модели программирования в Progress:



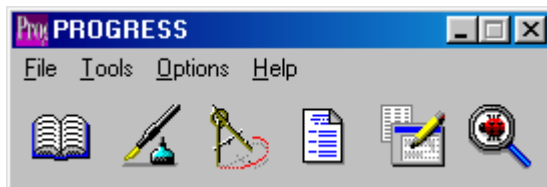
end. —————

I. PROGRESS 4GL

1. СТРУКТУРА ЯЗЫКА

1.1. Первое знакомство со средами для создания программ

Стандартное начало сеанса разработчика в Progress - запуск Application Development Environment (ADE) Desktop (командой proade, выбором соответствующего пункта меню или иконки в среде Windows). ADE Desktop представляет собой управляющую панель для запуска основных компонент Progress.



Data Dictionary	- словарь данных;
Procedure Editor	- текстовый редактор процедур;
ApplicationBuilder	- основная среда для разработки приложений;
Results	- среда для создания символьных отчетов;
Report Builder	- среда для создания графических отчетов;
Application Debugger	- отладчик.

Выберем на панели Desktop кнопку Procedure Editor. Это многобуферный текстовый редактор со всеми основными возможностями, необходимыми для написания и отладки программ. В рабочем поле редактора напомним первую программу:

DISPLAY " HELLO, WORLD! ".

Программа на языке Progress 4GL - это последовательность операторов, каждый из которых заканчивается точкой.

Запустим на выполнение эту программу: Compile -> Run (или клавишей F2).

Закроем Procedure Editor (File -> Exit) и познакомимся еще с одной компонентой Progress. Выберем на панели Desktop кнопку AppBuilder.

Заметим, что вызов компонент в среде Progress осуществляется, как правило, иерархически. Если вызвать AppBuilder через пункт меню Tools компоненты Procedure Editor, дальнейшая работа в Procedure Editor будет возможна только после закрытия AppBuilder. Исключением является запуск редактора через Tools -> PRO*Tools или через пункт Tools -> New Procedure Window. В этих случаях стартует однобуферный Procedure Editor, который активен независимо от других компонент Progress.


AppBuilder (или Application Builder - AB) - это интерактивная среда со встроенным редактором для создания и отладки приложений. Построение интерфейсов здесь в значительной степени автоматизировано чтобы избавить программиста от рутинной работы.

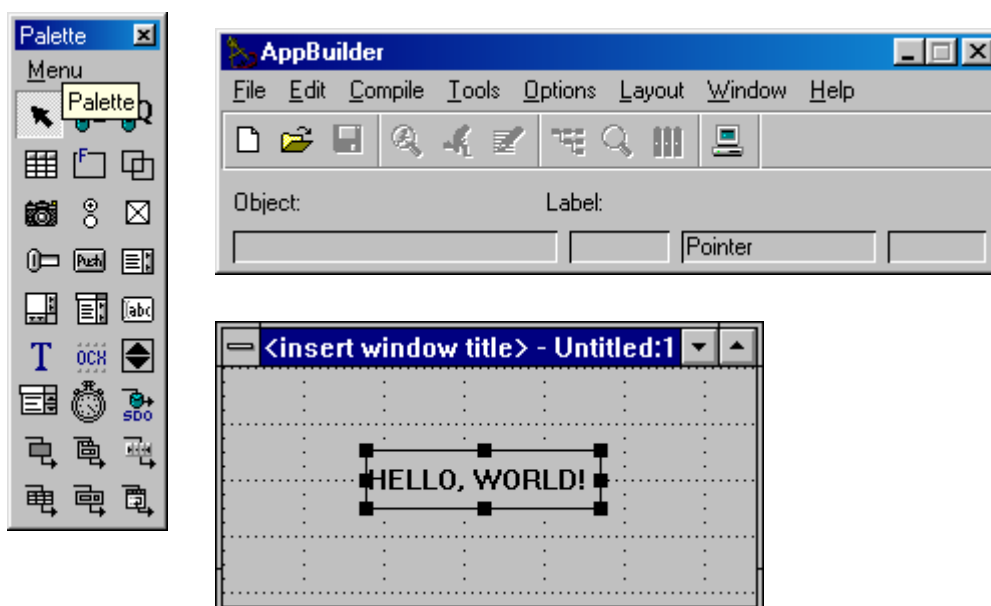
AB имеет два управляющих окна:


- Первое содержит основное меню и поименованные кнопки, дублирующие функции некоторых пунктов меню.
- Второе окно представляет из себя палитру объектов.

Создадим и запустим на выполнение в этой среде программу, аналогичную предыдущей:



- нажмем кнопку  в главном управляющем окне;
- выберем Window в открывшемся списке объектов; OK;
- после того как развернется окно, нажмем кнопку T на палитре и щелкнем мышью по рабочему полю окна, размещая в нем текстовый объект;
- в поле Text в нижней части главного управляющего окна наберем текст "HELLO, WORLD!" вместо "text 1".



- запустим эту программу на исполнение либо нажатием кнопки Run () в главном управляющем окне, либо через Compile -> Run, либо клавишей F2.

Завершить выполнение программы можно нажатием кнопки Stop ().

Более подробное знакомство со средой АВ состоится в следующей главе. А сейчас следует закрыть АВ. Изучение базовых возможностей языка Progress 4GL будет проходить в Procedure Editor, который следует снова вызвать через Desktop.

1.2. Базовые типы данных

В Progress представлены все основные типы данных. Ниже перечислены базовые типы и приведены примеры соответствующих констант:

Тип	Изображение константы
CHARACTER	"Hello"
DATE	10/25/94
DECIMAL	-3,746.93
INTEGER	997,115
LOGICAL	True / false или yes / no

- Изображение констант типа DATE зависит от стартовых параметров Progress(см. приложение1).
- Диапазон значений для дат 1/1/32768 до н.э.- 12/31/32767 н.э.
- В значениях DECIMAL могут учитываться до 50 цифр, включая 10 знаков после точки.
- Диапазон возможных значений для типа INTEGER от -2,147,483,648 до 2,147,483,647.

1.3. Идентификаторы и комментарии

Идентификатор в Progress - это последовательность букв, цифр и специальных знаков (\$, &, #, %, -, _), которая начинается с буквы. Например: student-num max1 prise\$

Комментарии выделяются следующим образом: /* это комментарий */

1.4. Описания переменных

Простейшее описание переменной выглядит так:

DEFINE VARIABLE variable-name AS data-type

[INITIAL constant]

[LABEL string]

[FORMAT string]

В результате такого рода описаний переменные приобретают не только тип, но и формат для вывода, метку и начальное значение в соответствии со следующими соглашениями:

Тип	Формат по умолчанию	Начальное значение
CHARACTER	x(8)	"" - пустая строка
DATE	99/99/99	?
DECIMAL	->>, >>9.99	0
INTEGER	->, >>>, >>9	0
LOGICAL	yes/no	No
RECID	>>>>>9	?
ROWID		Последовательность байт длины 0
HANDLE		?
WIDGET-HANDLE		?
COM-HANDLE		?

В качестве метки по умолчанию используется идентификатор.

Опция INITIAL используется для явного задания начального значения переменной.

Опция FORMAT позволяет задать формат ввода/вывода переменной (см. примеры использования в Приложении 3).

Опция LABEL используется для явного задания метки переменной.

Переопределение меток и форматов возможно также в операторе DISPLAY.

Переменные могут описываться по подобию уже существующих переменных:

```
DEFINE VARIABLE variable-name LIKE variable-name [INITIAL constant][LABEL string][FORMAT string]
```

Пример 1.4.1

```
DEFINE VARIABLE integer_var AS INTEGER.
DEFINE VARIABLE decimal_var AS DECIMAL.
DEFINE VARIABLE string_var AS CHARACTER.
DISPLAY integer_var decimal_var string_var.
```

Пример 1.4.2

```
DEFINE VARIABLE integer_var AS INTEGER
  LABEL "INTEGER" INITIAL "1111" FORMAT ">>>>99".
DEFINE VARIABLE decimal_var AS DECIMAL
  LABEL "DECIMAL" INITIAL "12.5" FORMAT ">>>>9.9".
DEFINE VARIABLE string_var AS CHARACTER
  LABEL "CHARACTER" INITIAL "HELLO!".
DISPLAY integer_var decimal_var string_var.
```

Пример 1.4.3

```
/**** VARIABLE DEFINITIONS *****/
DEFINE VARIABLE integer_var AS INTEGER
  LABEL "INTEGER" INITIAL "1111" FORMAT ">>>>99".
DEFINE VARIABLE decimal_var AS DECIMAL
  LABEL "DECIMAL" INITIAL "12.5" FORMAT ">>>>9.9".
DEFINE VARIABLE string_var AS CHARACTER
  LABEL "CHARACTER" INITIAL "HELLO!".
/**** FRAME DEFINITIONS *****/
DEFINE FRAME frame1
  WITH CENTERED TITLE "VARIABLES".
/**** MAIN BLOCK *****/
DISPLAY integer_var decimal_var string_var WITH FRAME frame1.
```

В приведенном выше примере используется конструктор FRAME, предназначенный для управления экранными формами. Подробное описание работы с фреймами будет приведено в разделе 6.1.

Пример 1.4.4

```
/**** VARIABLE DEFINITIONS *****/
DEFINE VARIABLE integer_var AS INTEGER
  LABEL "INTEGER" INITIAL "1111" FORMAT ">>>>99".
DEFINE VARIABLE decimal_var AS DECIMAL
  LABEL "DECIMAL" INITIAL "12.5" FORMAT ">>>>9.9".
DEFINE VARIABLE string_var AS CHARACTER
  LABEL "CHARACTER" INITIAL "HELLO!".
```

```

/***** FRAME DEFINITIONS *****/
DEFINE FRAME frame1
  WITH CENTERED TITLE "VARIABLES".
DEFINE FRAME frame2 integer_var COLON 20
  decimal_var COLON 20
  string_var COLON 20
  WITH CENTERED SIDE-LABELS TITLE "THE SAME VARIABLES".
/***** MAIN BLOCK *****/
DISPLAY integer_var decimal_var string_var WITH FRAME frame1.
DISPLAY integer_var decimal_var string_var WITH FRAME frame2.

```

1.5. Операции

Progress имеет достаточно стандартный набор операций:

Численные операции:

Операция	знак операции	описание операции
унарный минус	-	смена знака выражения
унарный плюс	+	сохранение знака выражения
Деление	/	деление одного выражения на другое; результат – типа DECIMAL
остаток от деления	MODULO	операнды и результат целочисленные
Умножение	*	умножение двух выражений
Бинарный минус	-	вычитание двух выражений
Бинарный плюс	+	сложение двух выражений

Операции сравнения:

Операция	знак операции
Меньше	< LT
Больше	> GT
меньше либо равно	<= LE
больше либо равно	>= GE
Равно	= EQ
не равно	<> NE

Операции с датами:

Операция	знак операции	описание операции
Вычитание	-	первый операнд - типа DATE, второй - DATE (INTEGER), результат - INTEGER (DATE);
Сложение	+	один из операндов типа DATE, другой - INTEGER, результат - DATE;

Строковые операции:

Операция	знак операции	описание операции
Конкатенация	+	соединение строк
сравнение с шаблоном	MATCHES	операнды - строковые, результат - логический ("progress" MATCHES ".rog*" выдает значение true);
Проверка префикса	BEGINS	операнды - строковые, результат - логический ("progress" BEGINS "pr" выдает значение true).

Логические операции:

Операция	знак операции
Отрицание	NOT
И	AND
Или	OR

Приоритеты операций:

Операция	Приоритет
Унарные -, +	7
MODULO, /, *	6
Бинарные -, +	5
BEGINS, MATCHES, <, >, <=, >=, =, <>	4
NOT	3
AND	2
OR	1

Обратим внимание на то, что знаки операций и операнды должны отделяться друг от друга пробелами.

Пример 1.5.1

Программа находит среднее арифметическое для трех вещественных чисел.

```
/***** VARIABLE DEFINITIONS *****/
```

```
DEFINE VARIABLE x AS DECIMAL.
```

```
DEFINE VARIABLE y AS DECIMAL.
```

```
DEFINE VARIABLE z AS DECIMAL.
```

```
/***** FRAME DEFINITIONS *****/
```

```
DEFINE FRAME frame1
```

```
    WITH CENTERED TITLE "AVERAGE".
```

```
/***** MAIN BLOCK *****/
```

```
SET x y z WITH FRAME frame1.
```

```
DISPLAY (x + y + z) / 3 LABEL "AVERAGE(X,Y,Z)" WITH FRAME frame1.
```

Оператор SET используется для ввода данных с клавиатуры.

1.6. Стандартные функции

Стандартные функции языка Progress в зависимости от типов параметров и результата могут быть разбиты на группы. Рассмотрим некоторые из них:

Функции с числовыми аргументами:

Функция	Описание
MAXIMUM(x, y...)	Максимум из последовательности
MINIMUM(x, y...)	Минимум из последовательности
LOG (x, y)	Логарифм x по основанию y
SQRT (x)	Корень квадратный из x
ROUND (x, n)	Округление x с точностью до n знаков после точки
TRUNCATE (x, n)	Отсечение x с точностью до n знаков после точки
EXP (x, y)	Возведение в степень, где x – основание, y - показатель

Функции преобразования типов:

Функция	Описание
DECIMAL(s)	Преобразование строки s к типу DECIMAL
INTEGER(s)	Преобразование строки s к типу INTEGER
STRING(n[,format])	Преобразование значения любого типа к строковому виду (возможно, с указанием формата)
DATE(m, d, y)	Преобразование трех целых чисел к виду DATE

Функции работы со строками:

Функция	Описание
NUM-ENTRIES(list)	число слов, входящих в строку list, где list - список слов, отделенных друг от друга запятыми (NUM-ENTRIES("Hello,World") выдает значение 2)
LOOKUP(s, list)	номер строки s, если строка s входит в список list, и 0 - в противном случае (LOOKUP("среда","воскресенье,понедельник,вторник,среда,четверг,пятница,суббота") выдает значение 4)
ENTRY(n, list)	слово с номером n из списка list
CAPS(s)	строка s большими буквами
INDEX(s, s1)	индекс первого символа подстроки s1, если s1 является подстрокой s, и 0 - в противном случае (просмотр строки s происходит слева направо)

R-INDEX (s, s1)	индекс первого символа подстроки s1, если s1 является подстрокой s, и 0 - в противном случае (просмотр строки s происходит справа налево)
TRIM (s)	строка s без хвостовых и ведущих пробелов
LENGTH (s)	длина строки s
SUBSTRING (s, n1[, n2])	вырезка подстроки

Функции с аргументами типа DATE:

функция	Описание
DAY (d)	номер дня в месяце
MONTH (d)	номер месяца
YEAR (d)	номер года
WEEKDAY (d)	номер дня в неделе
TODAY	текущая дата

Функции времени:

функция	Описание
TIME	текущее время (в секундах, начиная с полуночи)

Условная функция:

функция	Описание
IF B THEN E1 ELSE E2	в зависимости от значения логического выражения B выдает в качестве результата одно из выражений E1, E2 (следует иметь в виду, что типы выражений E1 и E2 должны соответствовать друг другу, то есть возможно сочетание типов DECIMAL и INTEGER, но невозможно DECIMAL и CHARACTER).

Пример 1.6.1

Программа по дате определяет номер дня в неделе.

DISPLAY WEEKDAY(TODAY).

Пример 1.6.2

Программа по трем сторонам треугольника определяет возможность построения треугольника и его площадь.

```

/***** VARIABLE DEFINITIONS *****/
DEFINE VARIABLE a AS DECIMAL FORMAT "99.99".
DEFINE VARIABLE b AS DECIMAL FORMAT "99.99".
DEFINE VARIABLE c AS DECIMAL FORMAT "99.99".
DEFINE VARIABLE p LIKE a.
/***** FRAME DEFINITIONS *****/
DEFINE FRAME frame1
    WITH CENTERED TITLE "triangle".
/***** MAIN BLOCK *****/
SET a b c WITH FRAME frame1.
p = ( a + b + c ) / 2.
DISPLAY IF a + b > c AND b + c > a AND a + c > b
    THEN STRING(SQRT(p * ( p - a ) * ( p - b ) * ( p - c )))
    ELSE "it is not possible"
    LABEL " s of triangle" FORMAT "x(20)" WITH FRAME frame1.

```

Пример 1.6.3

Следующий пример позволит уточнить некоторые приемы работы с датами. Запустите эту программу на исполнение дважды с входными датами 01/01/45 и 01/01/70.

```

DEFINE VARIABLE d AS DATE.
SET d.
DISPLAY DAY(d) LABEL "DAY"
    MONTH(d) LABEL "MONTH"
    YEAR(d) LABEL "YEAR".

```

Если Вас не устраивает значение стартового параметра -уу - измените его (по умолчанию значение этого параметра для Progress v.9 -1950)

Пример 1.6.4

Следующий пример демонстрирует приемы работы с функцией TIME.

```
DISPLAY TIME LABEL "TIME" FORMAT ">>>>>9"
      TRUNCATE(TIME / 3600, 0) LABEL "HOURS" FORMAT ">9"
      TRUNCATE(TIME / 60, 0) MODULO 60 LABEL "MINUTES" FORMAT ">9"
      TIME MODULO 60 LABEL "SECONDS" FORMAT ">9"
      STRING(TIME,"HH:MM:SS") LABEL "TIME".
```

1.7. Управляющие операторы

Оператор присваивания в Progress выглядит так:

I = P.

где I - идентификатор (получатель), P - выражение (источник).

Типы получателя и источника должны совпадать. Возможно лишь одно традиционное исключение из правила - получателям типа DECIMAL можно присваивать целочисленные значения.

Оператор присваивания можно использовать с опцией NO-ERROR. Эта опция указывает, что любая динамическая ошибка, которая произойдет в операторе присваивания должна быть проигнорирована. После того, как исполнение оператора завершится, можно проанализировать ошибку через системную переменную ERROR-STATUS(см. 1.12).

Оператор ASSIGN также, как и оператор присваивания, может использоваться для задания значения переменных. Его отличие от простого оператора присваивания в том, что в одном операторе можно сразу задать значения нескольких переменных (это эффективнее, чем несколько отдельных операторов присваивания). Например:

```
ASSIGN x = 12.1 y = 14.5 n = 5 m = 7.
```

Оператор ASSIGN также может использоваться с опцией NO-ERROR.

Сокращенная форма условного оператора:

```
IF B THEN O
```

где B - логическое выражение, O - один оператор.

При необходимости выполнить последовательность операторов, нужно эту последовательность заключить в операторные скобки:

```
DO: ... END.
```

Полная форма условного оператора:

```
IF B THEN O1 ELSE O2
```

где O1 и O2 - операторы.

Оператор выбора:

```
CASE E :
      WHEN K11 [OR WHEN K12] ... THEN O1
      WHEN K21 [OR WHEN K22] ... THEN O2
      ...
      [OTHERWISE On]
END [CASE].
```

где E - выражение, Kij - константы (возможные значения выражения E), O1, O2, ..., On - операторы.

Оператор цикла:

```
REPEAT: P END.
```

где P - последовательность операторов.

Оператор REPEAT является блоком. Выход из блока осуществляется либо по нажатию на клавишу END-ERROR (обычно это Esc), либо по оператору LEAVE:

```
LEAVE [ M].
```

где M - метка блока. Если метка не указана, управление передается следующему за блоком оператору.

Оператор цикла с параметром:

```
DO I = E1 TO E2 [ BY E3 ]: P END.
```

где I - идентификатор цикла; E1, E2, E3 - выражения; P - последовательность операторов.

Идентификатор I должен быть предварительно описан, а типы выражений E1, E2, E3 - соответствовать друг другу.

Оператор цикла с предусловием:

```
DO WHILE B: P END.
```

где I - идентификатор цикла; B - выражение, выдающее логическое значение; P - последовательность операторов.

Оператор паузы:

```
PAUSE [ n ].
```

где n - целое число.

Оператор паузы без параметра задерживает исполнение программы до нажатия на любую клавишу, а с параметром n - на n секунд.

Операторы выхода:

```
RETURN
```

(возврат в вызывающую процедуру или в редактор)

```
QUIT
```

(возврат в редактор или в операционную систему)

Пример 1.7.1

```
/* VARIABLE DEFINITIONS */
DEFINE VARIABLE d AS DATE LABEL "INPUT DATE".
DEFINE VARIABLE s AS CHAR LABEL "ROMA DATE" FORMAT "X(12)".
/* FRAME DEFINITIONS */
DEFINE FRAME frame1 WITH CENTERED.
/* MAIN BLOCK */
SET d WITH FRAME frame1.
s = STRING( DAY(d) ) + "/".
IF MONTH(d) >= 10 THEN s = s + "X" .
CASE MONTH(D) MODULO 10:
  WHEN 1 THEN s = s + "I".
  WHEN 2 THEN s = s + "II".
  WHEN 3 THEN s = s + "III".
  WHEN 4 THEN s = s + "IV".
  WHEN 5 THEN s = s + "V".
  WHEN 6 THEN s = s + "VI".
  WHEN 7 THEN s = s + "VII".
  WHEN 8 THEN s = s + "VIII".
  WHEN 9 THEN s = s + "IX".
END.
s = s + "/" + STRING(YEAR(d)).
DISPLAY s WITH FRAME frame1.
```

1.8. Массивы

В языке Progress предусмотрены только одномерные массивы.

В простейшем случае описание массива выглядит так:

```
DEFINE VARIABLE variable
```

```
AS datatype
```

```
EXTENT n
```

```
[INITIAL [ constant , ... constant]].
```

Доступ к элементам массива осуществляется по индексу. Индекс - выражение типа INTEGER.

Пример 1.8.1

```
/* VARIABLE DEFINITIONS */
DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE fact AS INTEGER EXTENT 5.
```

```

/* FRAME DEFINITIONS */
DEFINE FRAME frame1 WITH CENTERED.
/* MAIN BLOCK */
FACT[1] = 1.
DO I = 2 TO 5:
  FACT[I] = FACT[I - 1] * I.
END.
DISPLAY FACT[1] FORMAT ">9"
  FACT[2] FORMAT ">9"
  FACT[3] FORMAT ">9"
  FACT[4] FORMAT ">99"
  FACT[5] FORMAT ">99" WITH FRAME frame1.

```

1.9. Процедуры без параметров и разделяемые переменные

В языке Progress предусмотрены описание и вызов внешних и внутренних процедур. Каждая внешняя процедура хранится в отдельном файле, она не содержит заголовка и ее имя определяется именем файла. Внутренние процедуры описываются в тексте соответствующей внешней процедуры. Описание внутренней процедуры:

```
PROCEDURE proc-name:
```

```
...
```

```
END [PROCEDURE].
```

Вызов процедуры без параметров выглядит так:

```
RUN proc-name.
```

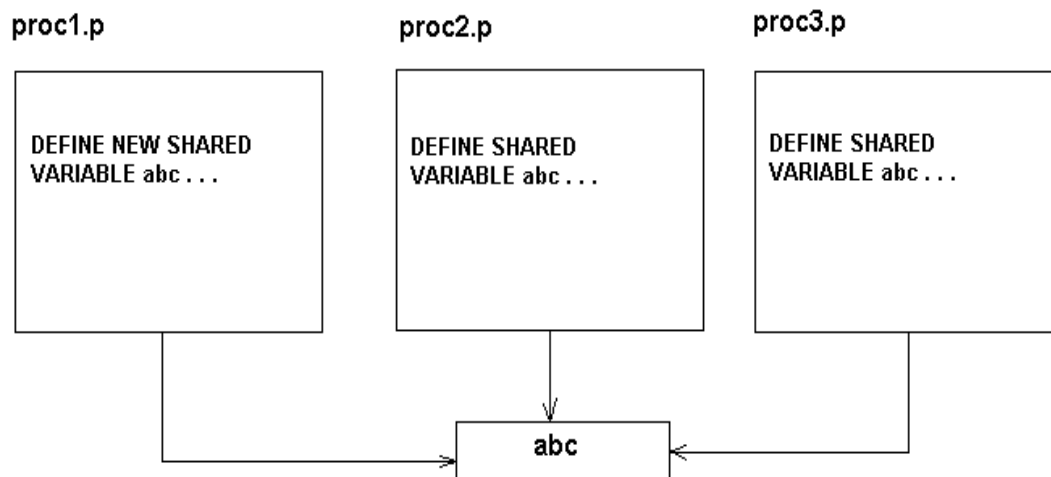
где proc-name - идентификатор процедуры (для внешней процедуры - имя файла, в котором хранится процедура);

```
VALUE(string-expression).
```

где string-expression - строковое выражение, выдающее идентификатор внешней или внутренней процедуры.

При вызове процедуры можно использовать опцию NO-ERROR.

Внутренние процедуры могут использовать переменные из соответствующей внешней процедуры.



Внешние процедуры могут использовать разделяемые переменные.

Синтаксис описания разделяемой переменной:

```
DEFINE [ [ NEW ] SHARED ] VARIABLE variable-name { AS data-type | LIKE field }
```

Описание разделяемой переменной в процедуре, которая создает эту переменную, выглядит так:

```
DEFINE NEW SHARED VARIABLE variable-name ....
```

а в процедуре, использующей переменную:

```
DEFINE SHARED VARIABLE variable-name....
```

Пример 1.9.1

Программа по радиусу определяет длину окружности и площадь соответствующего круга.

```
/* VARIABLE DEFINITIONS */
DEFINE NEW SHARED VARIABLE r AS DECIMAL FORMAT ">>9.99".
DEFINE NEW SHARED VARIABLE s AS DECIMAL FORMAT ">>9.99".
DEFINE NEW SHARED VARIABLE p AS DECIMAL FORMAT ">>9.99".
/* FRAME DEFINITIONS */
DEFINE FRAME frame1
  "ВВЕДИТЕ РАДИУС:" r SKIP(1)
  "ПЛОЩАДЬ КРУГА: " s SKIP(1)
  "ДЛИНА ОКРУЖНОСТИ:" p WITH CENTERED NO-LABELS.
/* MAIN BLOCK */
SET r WITH FRAME frame1.
RUN d:\examples\part1\proc1.p.
RUN d:\examples\part1\proc2.p.
DISPLAY s p WITH FRAME frame1.
```

Процедура proc1.p:

```
DEFINE SHARED VARIABLE r AS DECIMAL.
DEFINE SHARED VARIABLE s AS DECIMAL.
s = 3.1415 * r * r.
```

Процедура proc2.p:

```
DEFINE SHARED VARIABLE r AS DECIMAL.
DEFINE SHARED VARIABLE p AS DECIMAL.
p = 3.1415 * r * 2.
```

Помимо внутренних и внешних процедур в Progress есть понятие **удаленной процедуры (remote procedure)**. Реализация этого механизма происходит через **Компонентный сервер приложений (AppServer)**, который запускается в сети в непосредственной близости от данных. Это позволяет выделить бизнес-логику из приложения и приблизить ее к данным, уменьшая тем самым сетевой трафик. Здесь речь идет уже о N-уровневой архитектуре клиент/сервер, когда сервер выполняет свои прежние функции, за клиентской частью остается интерактивное взаимодействие с пользователем, а массивную обработку данных берут на себя удаленные процедуры при компонентных серверах.

1.10. Процедуры и функции с параметрами

В Progress предусмотрены три способа передачи параметра:

INPUT	- параметр используется как входной,
OUTPUT	- параметр используется как выходной,
INPUT-OUTPUT	- параметр используется при входе и выходе из процедуры.

Можно создавать пользовательские процедуры и функции с параметрами.

Способ передачи параметров указывается как при описании формальных параметров процедуры (функции), так и при вызове процедуры (функции) (по умолчанию параметры передаются как INPUT).

Пример 1.10.1

Программа, вычисляющая корни квадратного уравнения.

```
/* VARIABLE DEFINITIONS */
DEFINE VARIABLE a AS DECIMAL FORMAT "99.99".
DEFINE VARIABLE b AS DECIMAL FORMAT "99.99".
DEFINE VARIABLE c AS DECIMAL FORMAT "99.99".
/* FRAME DEFINITIONS */
DEFINE FRAME frame1
  "ВВЕДИТЕ КОЭФФИЦИЕНТЫ УРАВНЕНИЯ : " a b c SKIP(1)
  "КОРНИ:" WITH CENTERED NO-LABELS TITLE "КОРНИ КВАДРАТНОГО УРАВНЕНИЯ".
/* MAIN BLOCK */
SET a b c WITH FRAME frame1.
/* a = 1, b = 4, c = 3 */
RUN pr (a,b,c).
/* PROCEDURE DEFINITIONS */
PROCEDURE pr:
  DEFINE INPUT PARAMETER a AS DECIMAL.
```

```

DEFINE INPUT PARAMETER b AS DECIMAL.
DEFINE INPUT PARAMETER c AS DECIMAL.
DEFINE VARIABLE d AS DECIMAL.
  IF a = 0
  THEN
    DISPLAY (- c / b) WITH FRAME frame1.
  ELSE
    DO:
      d = b * b - 4 * a * c .
      IF d > 0
      THEN
        DISPLAY ( - b + SQRT(d) ) / ( 2 * a )
          ( b + SQRT(d) ) / ( 2 * a ) WITH FRAME frame1.
      ELSE
        DISPLAY "complex roots" WITH FRAME frame1.
    END.
  END.
END.

```

Пример 1.10.2

Программа, вычисляющая факториал натурального числа.

```

/* FUNCTION DEFINITIONS */
FUNCTION fact RETURNS INTEGER (INPUT n AS INTEGER).
  IF n = 0
  THEN RETURN(1).
  ELSE RETURN(n * fact(n - 1)).
  END FUNCTION.
/* VARIABLE DEFINITIONS */
DEFINE VARIABLE n AS INTEGER.
/* FRAME DEFINITIONS */
DEFINE FRAME frame1
  "ВВЕДИТЕ ЗНАЧЕНИЕ : " n SKIP(1)
  "FACT(N) = " WITH CENTERED NO-LABELS.
/* MAIN BLOCK */
SET n FORMAT "9" WITH FRAME frame1.
DISPLAY fact(n) FORMAT ">>>>>9" WITH FRAME frame1.

```

Пример 1.10.3

Программа, определяющая по алгоритму Евклида наибольший общий делитель для пары натуральных чисел.

```

/* FUNCTION DEFINITIONS */
FUNCTION maxdel RETURNS INTEGER
  (INPUT a AS INTEGER, INPUT b AS INTEGER).
  IF a MODULO b = 0
  THEN RETURN(b).
  ELSE RETURN(maxdel( b, a MODULO b )).
  END FUNCTION.
/* VARIABLE DEFINITIONS */
DEFINE VARIABLE a AS INTEGER.
DEFINE VARIABLE b AS INTEGER.
/* FRAME DEFINITIONS */
DEFINE FRAME frame1
  "ВВЕДИТЕ ЗНАЧЕНИЯ : " a b SKIP(1)
  "MAXDEL(A,B) = " WITH CENTERED NO-LABELS TITLE "MAXDEL".
/* MAIN BLOCK */
SET a b WITH FRAME frame1.
DISPLAY maxdel(a,b) WITH FRAME frame1.

```

1.11. PERSISTENT процедуры

Процедурный указатель (procedure handle) - это указатель, идентифицирующий вызов процедуры в стеке вызовов процедур. Каждой активной внешней процедуре автоматически сопоставляется процедурный указатель. Существует специальная системная переменная THIS-PROCEDURE, указывающая на текущую внешнюю процедуру. Тип этой системной переменной - HANDLE.

Вызов внешней процедуры может быть PERSISTENT или NON-PERSISTENT. Все рассмотренные до сих пор вызовы были NON-PERSISTENT по умолчанию. NON-PERSISTENT вызов процедуры создает свое окружение (context), которое существует только до конца работы процедуры. PERSISTENT вызов процедуры создает окружение, которое сохраняется и по окончании вызова - до конца работы Progress-приложения. Как следствие, до конца работы приложения можно исполнять соответствующие внутренние процедуры и функции.

Вызов PERSISTENT процедуры выглядит так:

```
RUN proc-name PERSISTENT [SET handle-variable][run-options].
```

Вызов внутренней процедуры из внешней PERSISTENT:

```
RUN proc-name IN handle-variable[run-options].
```

В обоих случаях handle-variable - это переменная типа HANDLE, указывающая на соответствующую внешнюю процедуру. Удаление PERSISTENT процедуры можно осуществить оператором:

```
DELETE PROCEDURE handle-variable.
```

Пример 1.11.1

Пример вызова PERSISTENT-процедуры:

```
/* VARIABLE DEFINITIONS */
DEFINE VARIABLE prochand AS HANDLE.
FUNCTION maxdel RETURNS INTEGER (INPUT a AS INTEGER, INPUT b AS INTEGER) IN prochand.
FUNCTION fact RETURNS INTEGER (INPUT n AS INTEGER) IN prochand.
/* MAIN BLOCK */
RUN D:\examples\part1\mix.p PERSISTENT SET prochand.
RUN pr IN prochand ( 5, 4, 3).
DISPLAY "MAXDEL(60,45)=" maxdel(60, 45) FORMAT ">9" SKIP(1)
"FACT(4)=" fact(4) FORMAT ">9".
```

Процедура mix.p:

```
/* FUNCTION DEFINITIONS */
FUNCTION maxdel RETURNS INTEGER (INPUT a AS INTEGER, INPUT b AS INTEGER).
IF a MODULO b = 0
THEN RETURN(b).
ELSE RETURN(maxdel( b, a MODULO b )).
END FUNCTION.
FUNCTION fact RETURNS INTEGER (INPUT n AS INTEGER).
IF n = 0
THEN RETURN(1).
ELSE RETURN(n * fact(n - 1)).
END FUNCTION.
/* PROCEDURE DEFINITIONS */
PROCEDURE pr:
DEFINE INPUT PARAMETER a AS DECIMAL.
DEFINE INPUT PARAMETER b AS DECIMAL.
DEFINE INPUT PARAMETER c AS DECIMAL.
DEFINE VARIABLE d AS DECIMAL.
IF a = 0
THEN
DISPLAY "the single root - " (- c / b) FORMAT "->9.99".
ELSE
DO:
d = b * b - 4 * a * c .
IF d > 0
THEN
DISPLAY "1-st root - " (- b + SQRT(d) ) / (2 * a) FORMAT "->9.99"
"2-nd root - " (- b - SQRT(d) ) / (2 * a) FORMAT "->9.99".
ELSE
DISPLAY "complex roots". END. END.
```

Существуют специальные атрибуты, присущие такому классу объектов, как процедуры (синтаксис использования атрибутов – объект:атрибут). Среди них отметим:

Атрибут	тип	Описание
FILE-NAME	CHARACTER	Имя файла, в котором содержится соответствующая внешняя процедура
PRIVATE-DATA	CHARACTER	Строка, используемая произвольным образом
PERSISTENT	LOGICAL	TRUE, если PERSISTENT процедура и FALSE в противном случае
PREV-SIBLING	HANDLE	Ссылка на предыдущую процедуру в “куче” вызовов процедур текущей сессии
NEXT-SIBLING	HANDLE	Ссылка на следующую процедуру в “куче” вызовов процедур текущей сессии

Существует специальная системная переменная SESSION (типа HANDLE) , указывающая на цепочку вызовов PERSISTENT процедур. Обратим внимание на некоторые ее атрибуты для работы с PERSISTENT - процедурами.

Атрибут	тип	Описание
FIRST-PROCEDURE	HANDLE	Первая процедура в цепочке вызовов
LAST-PROCEDURE	HANDLE	Последняя процедура в цепочке вызовов

Для определения конца цепочки может быть использована логическая функция VALID-HANDLE.

Следующий пример демонстрирует некоторые приемы работы с цепочкой вызовов PERSISTENT процедур.

Пример 1.11.2

```

/* VARIABLE DEFINITIONS */
DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE phand AS HANDLE.
DEFINE VARIABLE s1 AS CHARACTER FORMAT "X(30)".
DEFINE VARIABLE s2 AS CHARACTER FORMAT "X(30)".
/* MAIN BLOCK */
DO I = 1 TO 9:
  RUN D:\NAT\BOOK9\EXAMPLES\PART1\PROC3.P PERSISTENT (I).
END.
/* FROM LEFT TO RIGHT */
phand = SESSION:FIRST-PROCEDURE.
DO WHILE VALID-HANDLE(phand):
  s1 = s1 + phand:PRIVATE-DATA.
  phand = phand:NEXT-SIBLING.
END.
/* FROM RIGHT TO LEFT */
phand = SESSION:LAST-PROCEDURE.
DO WHILE VALID-HANDLE(phand):
  s2 = s2 + phand:PRIVATE-DATA.
  phand = phand:PREV-SIBLING.
END.
DISPLAY s1 s2.

```

Текст процедуры proc3.p:

```

DEFINE INPUT PARAMETER n AS INTEGER.
THIS-PROCEDURE:PRIVATE-DATA = STRING(N).

```


1.12. Системная переменная ERROR-STATUS

Системная переменная ERROR-STATUS (типа Handle) обычно используется после операторов с опцией NO-ERROR. В приведенной ниже таблице перечислены атрибуты этой переменной и их типы:

Атрибут	тип	Описание
ERROR	LOGICAL	значение TRUE указывает на наличие ошибки
NUM-MESSAGES	INTEGER	общее количество ошибок в операторе

Пример 1.12.1

```

/* VARIABLE DEFINITIONS */
DEFINE VARIABLE I AS INTEGER.
DEFINE VARIABLE STR AS CHARACTER.
/* FRAME DEFINITIONS */
DEFINE FRAME frame1
  "ВВЕДИТЕ ЦЕЛОЕ ЧИСЛО:" STR WITH CENTERED NO-LABEL.
/* MAIN BLOCK */
SET STR WITH FRAME frame1.
ASSIGN I = INTEGER(STR) NO-ERROR.
IF ERROR-STATUS:ERROR
THEN MESSAGE "ЭТО НЕ ЦЕЛОЕ ЧИСЛО" VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.

```

В приведенной ниже таблице перечислены методы, применимые к системной переменной ERROR-STATUS:

Метод	тип	Описание
GET-MESSAGE (n)	CHARACTER	системное сообщение, соответствующее n ошибке в операторе
GET-NUMBER (n)	INTEGER	номер системного сообщения, соответствующего n ошибке в операторе

Пример 1.12.2

В примере продемонстрирован возможный вариант применения метода GET-NUMBER.

```

/* VARIABLE DEFINITIONS */
DEFINE VARIABLE I AS INTEGER.
DEFINE VARIABLE STR AS CHARACTER.
/* FRAME DEFINITIONS */
DEFINE FRAME frame1
  "ВВЕДИТЕ ЦЕЛОЕ ЧИСЛО:" STR WITH CENTERED NO-LABEL.
/* MAIN BLOCK */
SET STR WITH FRAME frame1.
ASSIGN I = INTEGER(STR) NO-ERROR .
IF ERROR-STATUS:ERROR
THEN
  MESSAGE IF ERROR-STATUS:GET-NUMBER(1) = 76
    THEN "DO NOT USE WRONG SYMBOLS!"
    ELSE "UNKNOWN ERROR"
    VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.

```

2. СОБЫТИЙНОЕ ПРОГРАММИРОВАНИЕ И ОБЪЕКТЫ ИНТЕРФЕЙСА

В этой главе работа будет происходить попеременно то в Procedure Editor, то в Application Builder (AB). Поэтому рекомендуется избрать основной компонентой AB и вызывать из нее по мере необходимости однобуферный Procedure Editor. Для этого:

- закрыть обычный Procedure Editor (запущенный через Desktop);
- стартовать AB;
- в управляющем окне AB через Tools -> New Procedure Window запустить редактор.

- Еще один вариант запуска однобуферного Procedure Editor: Tools -> PRO*Tools ->



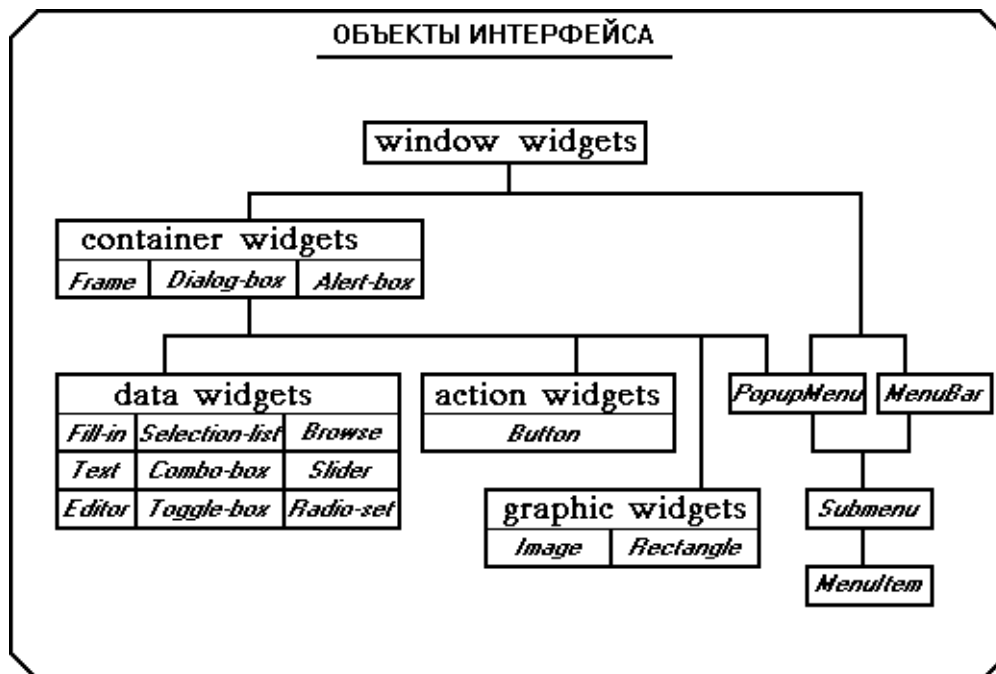
Однобуферный Procedure Editor и PRO*Tools являются вспомогательными инструментами и могут быть запущены параллельно основным компонентам среды разработчика (вызов основных компонент всегда иерархический).

2.1. Категории объектов интерфейса

Объекты интерфейса в Progress делятся на следующие категории:

Window Widget	окно		WINDOW	
Container Widgets	другие контейнеры	DIALOG-BOX	ALERT-BOX	FRAME
Action Widgets	используются для управления выполнением программы	BUTTON	MENU-BAR MENU-ITEM	SUBMENU
Data Widgets	используются для представления данных	FILL-IN SELECTION-LIST COMBO-BOX	TEXT BROWSE RADIO-SET	EDITOR SLIDER TOGGLE-BOX
Grafic Widgets	используются для оформления интерфейса	RECTANGLE		IMAGE

Иерархия объектов интерфейса такова:



Каждому типу объектов интерфейса соответствует свой набор **событий**, **атрибутов** и **методов**. Для каждой пары **объект-событие**, т.е. для конкретного события, случившееся по отношению к конкретному типу объекта интерфейса, существует умалчиваемая реакция системы (часто - игнорирование события), но может быть назначен и программный код - так называемый, **триггер** обработки события.

Progress поддерживает WIDGET-HANDLE переменные, которые можно использовать как указатели на объекты интерфейса.

2.2. Атрибуты

Синтаксис для всех типов объектов интерфейса, кроме фреймов:
 widget-name:attribute-name

для фреймов:

FRAME frame-name:attribute-name

Существуют атрибуты, присущие практически всем типам объектов интерфейса. Перечислим некоторые из них:

Атрибут	Тип	Описание
LABEL	CHARACTER	метка объекта интерфейса
FORMAT	CHARACTER	форматная строка
SCREEN-VALUE	CHARACTER	значение объекта интерфейса, которое содержится в экранном буфере.
MODIFIED	LOGICAL	имеет значение true, если содержимое экранного буфера отличается от содержимого объекта
SENSITIVE	LOGICAL	специфицирует фокусировку объекта интерфейса, по умолчанию – false
VISIBLE	LOGICAL	специфицирует видимость объекта интерфейса, для статических объектов по умолчанию - true, для динамических – false
HANDLE	WIDGET-HANDLE	ссылка на объект интерфейса
FRAME	HANDLE	ссылка на фрейм, к которому принадлежит объект
X Y	INTEGER	координаты левого верхнего угла объекта интерфейса
HEIGHT-CHAR HEIGHT-PIXEL WIDTH-CHAR WIDTH-PIXEL	INTEGER	размеры объекта
BGCOLOR FGCOLOR FONT	INTEGER	цвет фона, цвет пера и номер шрифта объекта интерфейса
SELECTABLE	LOGICAL	специфицирует возможность выбора объекта интерфейса в run-time
MOVABLE	LOGICAL	специфицирует возможность передвижения объекта интерфейса в run-time
RESIZABLE	LOGICAL	специфицирует возможность изменения размера объекта интерфейса в run-time
PRIVATE-DATA	CHARACTER	свободный атрибут для хранения любой информации – используется по усмотрению программиста
HELP	CHARACTER	строка помощи

2.3. Методы

В дополнение к атрибутам, некоторым объектам интерфейса сопоставлены методы. **Метод** - это некое специфическое действие над объектом интерфейса. Методы вырабатывают значения, подобно функциям. Синтаксис для всех типов объектов интерфейса кроме фреймов:

widget-name:method-name ([arg1 [, arg2] . . .])

а для фреймов:

FRAME frame-name:method-name ([arg1 [, arg2] . . .])

Методы будут рассмотрены далее для каждого конкретного типа объекта интерфейса.

2.4. Объекты интерфейса типа BUTTON

Типичное событие, связанное с объектами интерфейса типа BUTTON: CHOOSE -выбор кнопки.

Типичные атрибуты: HANDLE, LABEL, SENSITIVE, VISIBLE.

Методы, применимые к BUTTON:

Метод	Описание
LOAD-IMAGE-UP(file-name)	Динамически загрузить картинку на кнопку для нейтрального ее состояния.
LOAD-IMAGE-DOWN(file-name)	Аналогично для нажатой кнопки.
LOAD-IMAGE-INSENSITIVE(file-name)	Аналогично для неактивной кнопки.

Пример 2.4.1

В приведенной ниже программе демонстрируется создание и использование простейшего объекта интерфейса - BUTTON.

```
/* DEFINE WIDGETS */
DEFINE BUTTON hello LABEL "HELLO".
DEFINE BUTTON btn-exit LABEL "EXIT".
/* DEFINE FRAME */
DEFINE FRAME frame1
    hello btn-exit WITH CENTERED.
/* DEFINE TRIGGERS */
ON CHOOSE OF hello
DO:
    MESSAGE "HOW ARE YOU? I AM GLAD TO SEE YOU."
    VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
END.
```

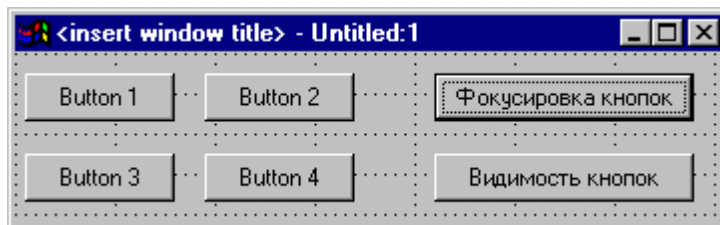
```
/* MAIN LOGIC */
ENABLE ALL WITH FRAME frame1.
WAIT-FOR CHOOSE OF btn-exit.
```


- Оператор MESSAGE посылает указанную строку в виде ALERT-BOX текущего окна.
- Оператор ENABLE ALL присваивает значение true атрибутам SENSITIVE всех объектов интерфейса указанного фрейма.
- Оператор WAIT-FOR ожидает указанное событие для завершения программы.


Для работы со следующим примером нужно перейти в Application Builder. В дальнейшем все примеры, в заголовке которых есть символы "AB", будут создаваться в этой среде. Примеры с обычными заголовками будут выполняться в Procedure Editor.

Пример 2.4.2 (AB)

Создадим приложение, в котором будем управлять фокусировкой и видимостью кнопок.



1. Откроем новое окно - Window - через кнопку  в управляющем окне AB.


2. Разместим в окне шесть кнопок через  на палитре объектов.

- В каждый момент времени один из объектов (в данном примере - кнопок) является активным. Выделение активного объекта производится щелчком мыши. Выделенный объект можно перемещать (указатель мыши фиксируется внутри объекта и передвигает его в нужное место). Можно изменять размеры активного объекта (указатель фиксируется на одной из ризок и передвигает ее). Удаляется выделенный объект с помощью клавиши Delete.
- Существует набор операций над группой активных объектов (перемещение, удаление, ...). Для выделения группы следует зафиксировать указатель мыши в верхнем левом углу предполагаемой прямоугольной области, охватывающей группу объектов, и передвинуть его в правый нижний угол области.
- Для удобства размещения на экране объектов предусмотрена разметка окна (Grid), изменять которую можно следующим образом:
- Options -> Preferences -> Grid Units выбор размера ячеек и интенсивности линий;


2. Событийное программирование и объекты интерфейса

- Options -> Display Grid нужно ли высвечивать разметку.
- Для выравнивания объектов можно использовать различные операции выравнивания, предлагаемые в пункте меню Layout -> Align ->

3. Все порождаемые нами объекты интерфейса по умолчанию приобретают некоторые характеристики (атрибуты) - имя, метку, расположение во фрейме, размер и т.д. Атрибуты эти могут быть изменены и дополнены через окно Property Sheet, которое открывается для активного объекта

через  или двойным щелчком по объекту. Для кнопок с метками "button-1" и "button-2" изменим значение атрибута enable: уберем флажок Enable в нижней части окна свойств – в разделе Other Settings.

4. Помимо изменения характеристик объектов, каждому из них можно сопоставить триггеры. Выбор

 приведет к вызову встроенного редактора (Section Editor), в котором можно написать триггеры для активного объекта. Напишем триггер на событие CHOOSE для кнопки с меткой "Фокусировка кнопок":

button-1:SENSITIVE = NOT button-1:SENSITIVE.

button-2:SENSITIVE = NOT button-2:SENSITIVE.

- Идентификаторы объектов Application Builder присваивает автоматически при создании объекта, поэтому в этом примере и далее следует использовать в триггерах текущие значения (здесь – идентификаторов кнопок), которые могут не совпадать с приведенными в тексте. Поэтому имена объектов рекомендуется вставлять в текст триггера из списка, представленного в Insert -> Object Name.

5. Напишем триггер на событие CHOOSE для кнопки с меткой "Видимость кнопок":


button-1:VISIBLE = NOT button-1:VISIBLE.

button-2:VISIBLE = NOT button-2:VISIBLE.

button-3:VISIBLE = NOT button-3:VISIBLE.

button-4:VISIBLE = NOT button-4:VISIBLE.

6. Сохраним созданную процедуру в файле с именем i02_0402.w ().

7. Запустим программу на исполнение, выбрав .

2.5. Объекты интерфейса типа IMAGE и RECTANGLE

IMAGE и RECTANGLE – это графические объекты, которые используются обычно для оформления интерфейса. Они размещаются во фреймах и всегда лежат под значимыми объектами (не способны перекрывать их).

IMAGE - это картинка, которая берется из файла и размещается во фрейме как самостоятельный объект. Progress поддерживает два типа графических форматов: .BMP и .ICO.

Основные атрибуты IMAGE: VISIBLE, X, Y, MOVABLE, HANDLE, FRAME.

Типичный метод: LOAD-IMAGE(file-name) - динамически загрузить картинку.

RECTANGLE – это прямоугольник, выполняющий чисто декоративные функции.

Основные атрибуты RECTANGLE: VISIBLE, X, Y, MOVABLE, HANDLE, FRAME, EDGE-PIXELS (ширина границы прямоугольника), FILLED (логический атрибут, показывающий сплошной ли это прямоугольник или только рамка).

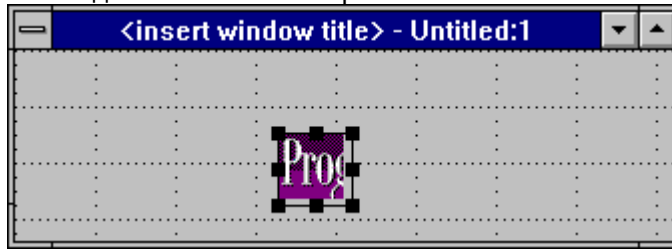
Пример 2.5.1





В программе демонстрируется простейший способ изображения объекта интерфейса IMAGE.

```
/* DEFINE WIDGETS */
DEFINE IMAGE newim1 FILE "GUI\ADEICON\progress.ico".
DEFINE IMAGE newim2 FILE "GUI\ADEICON\cnfginfo.ico".
DEFINE BUTTON exit IMAGE FILE "GUI\ADEICON\unprog.ico".
/* DEFINE FRAMES */
DEFINE FRAME img-frame
  newim1 newim2 skip(2) exit WITH TITLE "IMAGES".
/* MAIN LOGIC */
ENABLE ALL WITH FRAME img-frame.
WAIT-FOR CHOOSE OF exit.
```

Пример 2.5.2 (AB)

В этом примере мы начнем создавать маленькое приложение с использованием объекта типа IMAGE.







1. Откроем новое окно (через ).
2. Выберем в палитре объект image () и разместим его во фрейме.
3. Войдем в окно свойств объекта (через  или двойным щелчком по объекту) и выбором кнопки Image в верхней части этого окна привяжем к нему любую картинку Progress, указав какой-либо файл из директории ...\\progress\\gui\\adeicon\\.
4. В том же окне свойств выберем кнопку Advanced, чтобы перейти к дополнительным атрибутам объекта. В открывшемся окне поставим флажок Movable, чтобы сделать картинку перемещаемой во время исполнения процедуры.
5. Сохраним созданную процедуру в файле с именем i02_0502.w: .
6. Запустим программу на исполнение.

Пример 2.5.3 (AB)

Создадим программу, использующую объект интерфейса RECTANGLE.



1. Откроем новое окно (через ).
2. Выберем в палитре объект button () и разместим его во фрейме. Уберем метку кнопки.
3. Выберем в палитре объект rectangle:  и разместим его вокруг кнопки.
4. Войдем в окно свойств rectangle (двойным щелчком по объекту) и укажем, что он “заполненный” - в Other Settings установим флажок Filled.
5. Войдем во встроенный редактор () , но не затем чтобы писать какой-либо триггер, а для того чтобы попасть в раздел Definition. Сделать это можно через выбор Section. Оказавшись в нужной секции редактора, опишем вспомогательную переменную:
`DEFINE VARIABLE i AS INTEGER.`
6. Оставаясь во встроенном редакторе, перейдем в секцию триггеров и напишем для кнопки триггер на событие CHOOSE:
`IF i > 4 THEN i = 0.
ELSE i = i + 1.
RECT-1:BGCOLOR = i.`
7. Сохраним созданную процедуру в файле с именем i02_0503.w.
8. Запустим программу на исполнение.

2.6. Объекты интерфейса типа FILL-IN

Переменная любого типа, описанная в программе, по умолчанию представляется объектом интерфейса типа FILL-IN.

Типичные события, связанные с FILL-IN объектами интерфейса:

ENTRY - фокусировка объекта интерфейса
LEAVE - снятие фокусировки

Типичные атрибуты: FORMAT, HANDLE, LABEL, SCREEN-VALUE, VISIBLE, MODIFIED.

Пример 2.6.1

Следующая программа демонстрирует использование события ENTRY.

```
/* DEFINE VARIABLE */
DEFINE VARIABLE n AS INTEGER INITIAL 2.
DEFINE VARIABLE d AS CHARACTER INITIAL "B".
/* DEFINE FRAMES */
DEFINE FRAME frame1
    n COLON 20 d COLON 50
    WITH SIDE-LABELS CENTERED ROW 5 NO-BOX.
/* DEFINE TRIGGERS */
ON ENTRY OF n DO:
    n = n * 2.
    DISPLAY n WITH FRAME frame1.
END.
ON ENTRY OF d DO:
    d = d + "A".
    DISPLAY d WITH FRAME frame1.
END.
/* MAIN LOGIC */
ENABLE ALL WITH FRAME frame1.
WAIT-FOR RETURN OF d.
```

Пример 2.6.2.

Программа демонстрирует использование события LEAVE.

```
/* DEFINE WIDGETS */
DEFINE VARIABLE field1 AS CHAR FORMAT "X(25)"
    LABEL "ВАШЕ ИМЯ ?".
DEFINE VARIABLE field2 AS INTEGER FORMAT "99"
    LABEL "ВАШ ВОЗРАСТ ?".
DEFINE BUTTON btn-exit LABEL "EXIT".
/* DEFINE FRAMES */
DEFINE FRAME frame1
    SKIP(2) field1 SKIP(2) field2 SKIP(2)
    btn-exit
    WITH NO-BOX CENTERED SIDE-LABELS.
/* DEFINE TRIGGERS */
ON LEAVE OF field1 DO:
    field1 = field1:SCREEN-VALUE.
    MESSAGE "ВАШЕ ИМЯ " + field1 + "?"
    VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
END.
ON LEAVE OF field2 DO:
    MESSAGE "ВАШ ВОЗРАСТ " + field2:SCREEN-VALUE + "?"
    VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
END.
/* MAIN LOGIC */
ENABLE ALL WITH FRAME frame1.
WAIT-FOR CHOOSE OF btn-exit.
```

Вместо оператора

field = field:SCREEN-VALUE.

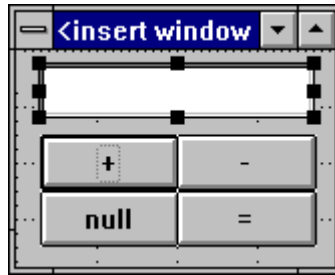
можно использовать оператор


ASSIGN field.

который сохраняет значение из экранного буфера непосредственно в самой переменной.

Пример 2.6.3 (AB)


Создадим программу-калькулятор (для сложения и вычитания).




1. Откроем новое окно через .

2. Отобразим в окне FILL-IN объект (выбрав его через палитру объектов - )

3. В окне свойств этого объекта установим флажок No-Label.

4. Войдем во встроенный редактор () , в раздел Definition, и опишем вспомогательные переменные:

```
DEFINE VARIABLE res AS INTEGER.  
DEFINE VARIABLE op AS CHARACTER.
```

5. Разместим в окне кнопку с меткой "+" и напомним для нее триггер на событие CHOOSE () :

```
op = "+".  
APPLY "ENTRY" TO fill-in-1.
```

- Оператор APPLY позволяет программно сгенерировать событие.

В окне свойств кнопки поставим флажок Flat.

Затем скопируем кнопку через Edit -> Copy, Edit -> Paste.

- Заметим, что при этом в Windows Clipboard копируется активный объект интерфейса со всеми его атрибутами (включая местоположение во фрейме) и триггерами. После вставки новый объект будет "лежать" на исходном и его следует "оттащить" в сторону.

Изменим метку полученной кнопки на "-" и подправим триггер для нее:

```
op = "-".  
APPLY "ENTRY" TO fill-in-1.
```

Подобным образом создадим кнопку "null" с триггером:

```
op = "".  
fill-in-1:SCREEN-VALUE = "".
```

и кнопку "=" с триггером:

```
op = "=".  
fill-in-1:SCREEN-VALUE = STRING(res).
```

6. Напишем триггер на событие ENTRY для fill-in-1:

```
DEFINE VAR m AS INTEGER.  
m = INTEGER(fill-in-1:SCREEN-VALUE).  
fill-in-1:SCREEN-VALUE = "".  
CASE op:  
  WHEN "+" THEN res = res + m.  
  WHEN "-" THEN res = res - m.  
  WHEN "" THEN res = m.  
END.
```

7. Сохраним созданную процедуру в файле с именем i02_0603.w .

8. Запустим программу на исполнение.

2.7. Групповые триггеры

Триггер обработки события может быть сопоставлен группе объектов интерфейса, как это показано в приведенном ниже примере.

Пример 2.7.1

```
/* DEFINE WIDGETS */
DEFINE VARIABLE field1
  AS CHAR FORMAT "x(11)" INITIAL "SWALLOW".
DEFINE VARIABLE field2
  AS CHAR FORMAT "x(11)" INITIAL "EAGLE".
DEFINE VARIABLE field3
  AS CHAR FORMAT "x(11)" INITIAL "ROBIN".
DEFINE VARIABLE field4
  AS CHAR FORMAT "x(11)" INITIAL "CAT".
DEFINE VARIABLE field5
  AS CHAR FORMAT "x(11)" INITIAL "TIGER".
DEFINE VARIABLE field6
  AS CHAR FORMAT "x(11)" INITIAL "COW".
DEFINE BUTTON btn-exit LABEL "exit".
/* DEFINE FRAMES */
DEFINE FRAME frame1
  btn-exit SKIP(2)
  field1 field2 field3 SKIP(2)
  field4 field5 field6
  WITH NO-BOX CENTERED NO-LABELS .
/* DEFINE TRIGGERS */

  ON ENTRY OF field1, field2, field3
DO:
MESSAGE "YOU CHOOSE BIRD"
  VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
END.
ON ENTRY OF field4,field5,field6
DO:
MESSAGE "YOU CHOOSE ANIMAL"
  VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
  IF SELF:SCREEN-VALUE = "TIGER"
  THEN MESSAGE "BE CAREFUL! IT IS A TIGER!"
    VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
END.
/* MAIN LOGIC */
DISPLAY field1 field2 field3 field4 field5 field6
  WITH FRAME frame1.
ENABLE ALL WITH FRAME frame1.
WAIT-FOR CHOOSE OF btn-exit.
```

- SELF – это системная переменная типа WIDGET-HANDLE, указывающая на объект интерфейса, по которому исполняется триггер.

2.8. Объекты интерфейса типа TOGGLE-BOX

Тип TOGGLE-BOX используется для представления логических значений. Он сопоставляется логическим переменным либо в операторе DEFINE, либо в интерактивных операторах с помощью VIEW-AS фразы. Например:

```
DEFINE VARIABLE sun AS LOGICAL VIEW-AS TOGGLE-BOX.
```

Типичные события, связанные с TOGGLE-BOX:

```
ENTRY - фокусировка объекта интерфейса
LEAVE - снятие фокусировки
VALUE-CHANGED - изменение значения
```

Типичные атрибуты: HANDLE, HELP, LABEL, MODIFIED, SCREEN-VALUE, SENSITIVE, VISIBLE, CHECKED.

2. Событийное программирование и объекты интерфейса

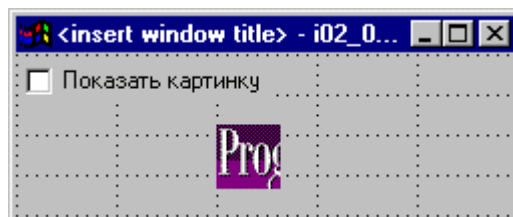
Новым в этом списке является атрибут CHECKED, он выдает логическое значение, соответствующее объекту интерфейса.


Пример 2.8.1

```
/*      DEFINE WIDGETS      */
DEFINE VARIABLE sun AS LOGICAL VIEW-AS TOGGLE-BOX.
DEFINE VARIABLE rain AS LOGICAL VIEW-AS TOGGLE-BOX.
DEFINE BUTTON btn-exit LABEL "exit".
/*      DEFINE FRAME      */
DEFINE FRAME frame1
    SKIP(2) sun rain
    SKIP(2) btn-exit WITH CENTERED SIDE-LABELS.
/*      DEFINE TRIGGERS      */
ON VALUE-CHANGED OF sun
DO:
    ASSIGN sun.
    IF sun
    THEN MESSAGE "SUNNY"
        VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
    ELSE MESSAGE "NUSTY"
        VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
    END.
ON VALUE-CHANGED OF rain
DO:
    IF rain:CHECKED
    THEN MESSAGE "RAINY"
        VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
    ELSE MESSAGE "DRY"
        VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
    END.
/*      MAIN LOGIC      */
DISPLAY sun rain WITH FRAME frame1.
ENABLE ALL WITH FRAME frame1.
WAIT-FOR CHOOSE OF btn-exit.
```

Пример 2.8.2 (AB)

Продолжим работу над маленьким приложением с картинкой, дополнив его объектом типа TOGGLE-BOX.



1. Через  прочитаем из файла созданную в одном из предыдущих примеров процедуру (i02_0502.w) и модифицируем ее.

2. Добавим в окно объект TOGGLE-BOX () и изменим его метку.

3. В начале выполнения программы наш TOGGLE-BOX должен иметь значение "yes". В Property Sheet этого объекта выберем кнопку Advanced для перехода к окну дополнительных атрибутов. В этом окне изменим Initial Value на "yes".

4. Напишем триггер на событие VALUE-CHANGED для TOGGLE-BOX ():

image-1:VISIBLE = NOT image-1:VISIBLE.

5. Сохраним измененную процедуру в файле i02_0802.w.

6. Запустим программу на исполнение.

2.9. Объекты интерфейса типа RADIO-SET

Тип RADIO-SET используется для представления данных любого базового типа, имеющих ограниченное число возможных значений, и сопоставляется переменным с помощью VIEW-AS фразы.

События, связанные с RADIO-SET: ENTRY, LEAVE, VALUE-CHANGED.

Атрибуты: HANDLE, HELP, LABEL, MODIFIED, SCREEN-VALUE, SENSITIVE, VISIBLE.

Методы:

Метод	Описание
ADD-LAST()	Добавляет еще одну кнопку.
DELETE()	Удаляет кнопку.
DISABLE()	Отменяет доступность кнопки.
ENABLE()	Восстанавливает доступность кнопки.
REPLACE()	Замещает кнопку.

Пример 2.9.1

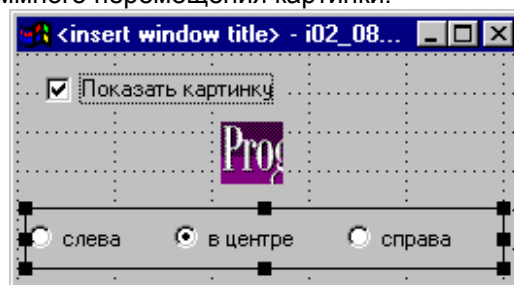
```

/*      DEFINE WIDGETS      */
DEFINE VARIABLE mark AS INTEGER INITIAL 5
VIEW-AS RADIO-SET
  RADIO-BUTTONS "EXCELLENT", 5, "GOOD", 4 ,
    "SATISFACTORY", 3, "NOT CREDIT", 2.
DEFINE BUTTON btn-exit LABEL "exit".
/*      DEFINE FRAME      */
DEFINE FRAME frame1
  SKIP(2) mark /* VIEW-AS RADIO-SET HORIZONTAL */
  SKIP(2) btn-exit WITH CENTERED SIDE-LABELS.
/*      DEFINE TRIGGERS      */
ON VALUE-CHANGED OF mark DO:
  ASSIGN mark.
  MESSAGE "MARK: " + STRING(mark).
END.
/*      MAIN LOGIC      */
DISPLAY mark WITH FRAME frame1.
MESSAGE "MARK: " + STRING(mark).
ENABLE ALL WITH FRAME frame1.
WAIT-FOR CHOOSE OF btn-exit.


```

Пример 2.9.2 (AB)

Дополним последний созданный нами в AB пример объектом интерфейса RADIO-SET, предназначенным для программного перемещения картинки.



1. Через  откроем процедуру i02_0802.w.

2. Добавим в окно объект RADIO-SET () и в Property Sheet укажем в качестве кнопок для него: "слева", 1, "в центре", 100, "справа", 200

3. Из Property Sheet перейдем по кнопке Advanced к окну дополнительных атрибутов и изменим Initial Value на 100, так как изначально наша картинка расположена в центре.

4. Войдем в редактор и напишем триггер на событие VALUE-CHANGED для RADIO-SET:
 ASSIGN radio-set-1
 image-1.X = radio-set-1.

5. Сохраним измененную процедуру в файле i02_0902.w и запустим программу на исполнение.

2.10. Объекты интерфейса типа SLIDER

Тип SLIDER используется для представления значений типа INTEGER в виде числовой шкалы.

Типичные события, связанные со SLIDER: ENTRY, LEAVE, VALUE-CHANGED.

Типичные атрибуты: HANDLE, HELP, LABEL, MODIFIED, SCREEN-VALUE, SENSITIVE, VISIBLE.

Пример 2.10.1

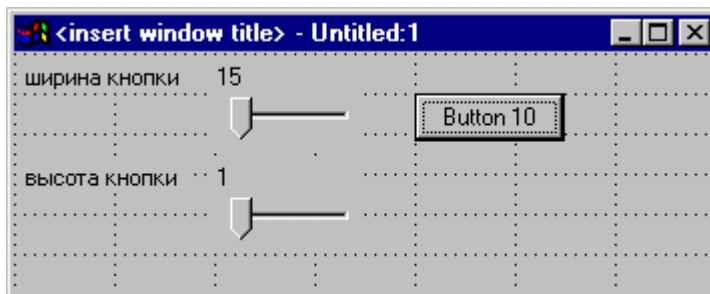
```

/*      DEFINE WIDGETS      */
DEFINE VARIABLE x AS INTEGER INITIAL 1 VIEW-AS SLIDER
      MIN-VALUE 1 MAX-VALUE 1000 HORIZONTAL SIZE-CHARS 20 BY 2.
DEFINE BUTTON btn-home LABEL "home".
DEFINE BUTTON btn-exit LABEL "exit".
/*      DEFINE FRAME      */
DEFINE FRAME frame1
      SKIP(5) x NO-LABEL AT ROW 1 COLUMN 9
      SKIP(2) btn-home
      SKIP(1) btn-exit WITH CENTERED .
/*      DEFINE TRIGGERS      */
ON VALUE-CHANGED OF x ASSIGN x.
ON CHOOSE OF btn-home DO:
      DEFINE VARIABLE i AS INTEGER.
      DO i = x TO 1 BY -1:
            x = i.
            DISPLAY x WITH FRAME frame1.
      END.
END.
/*      MAIN LOGIC      */
ENABLE ALL WITH FRAME frame1.
WAIT-FOR CHOOSE OF btn-exit.



```


Пример 2.10.2 (AB)

Создадим приложение, в котором объекты интерфейса типа SLIDER будут использоваться для изменения размеров кнопки.



1. Откроем новое окно.

2. Разместим в окне кнопку стандартного размера () и два объекта типа slider (). Через Property Sheet изменим диапазон возможных значений первого - от 15 до 25, а второго - от 1 до 5. Установим для каждого флажок Horizontal.

3. Создадим рядом с ними два объекта типа text ().

- Text - это объект интерфейса, который обладает минимумом атрибутов, не имеет событий и методов. Используется обычно для создания текста, на котором нельзя сфокусироваться.

Значениями этих двух объектов сделаем строки "ширина кнопки" и "высота кнопки"

4. Напишем триггеры на событие VALUE-CHANGED для sliders:

триггер для slider-1: ASSIGN slider-1
 button-10:WIDTH = slider-1.

триггер для slider-2: ASSIGN slider-2
 button-10:HEIGHT = slider-2.

5. Сохраним процедуру в файле i02_1002.w и запустим программу на исполнение.

2.11. Объекты интерфейса типа EDITOR

EDITOR выглядит как прямоугольная область на экране (при необходимости появляется SCROLLBAR) и используется для представления длинных строк.

Типичные события: ENTRY, LEAVE.

Типичные атрибуты: HANDLE, HELP, LABEL, MODIFIED, SCREEN-VALUE, SENSITIVE, VISIBLE.

Методы:

Метод	Описание
DELETE-CHAR()	Удаляет символ.
DELETE-LINE()	Удаляет линию.
INSERT-STRING(string)	Вставляет строку с текущей позиции курсора.
READ-FILE(file-name)	Удаляет текст и вставляет содержимое файла.
SAVE-FILE(file-name)	Сохраняет текст в файле.
INSERT-FILE(file-name)	Вставляет содержимое файла с текущей позиции курсора.
SEARCH(string,flag)	Ищет указанную строку, начиная с текущей позиции курсора.

Пример 2.11.1.

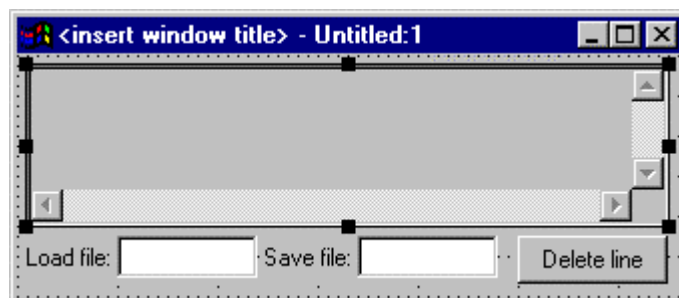
```

/*      DEFINE WIDGETS      */
DEFINE VARIABLE address AS CHARACTER VIEW-AS EDITOR
      INNER-CHARS 20 INNER-LINES 3.
DEFINE BUTTON btn-exit LABEL "exit".
/*      DEFINE FRAMES      */
DEFINE FRAME frame1
      address NO-LABEL SKIP
      btn-exit
      WITH TITLE "ADDRESS" AT ROW 2 COLUMN 3.
/*      DEFINE TRIGGERS      */
ON LEAVE OF address DO:
      ASSIGN address.
      MESSAGE address
      VIEW-AS ALERT-BOX INFORMATION BUTTONS OK
      TITLE "YOUR ADDRESS".
END.
/*      MAIN LOGIC      */
ENABLE ALL WITH FRAME frame1.
WAIT-FOR CHOOSE OF btn-exit.

```

Пример 2.11.2 (AB)

Создадим приложение, которое содержит run-time редактор, позволяющий работать с текстовыми файлами.



1. Откроем новое окно.



2. Разместим в окне объект интерфейса типа EDITOR (), два объекта FILL-IN (изменим их метки на "Load file" и "Save file") и кнопку с меткой "Delete line".

3. Для поля "Load file" напишем триггер на событие RETURN: это событие относится к клавиатурным и выбрать его можно следующим образом - через кнопку New в Section Editor вызвать окно Choose Event и нажать кнопку Keyboard Event... ; после открытия соответствующего окна следует нажать на клавиатуре клавишу RETURN):

```

IF NOT editor-1:READ-FILE(fill-in-2:SCREEN-VALUE)
  THEN MESSAGE "IT IS NOT POSSIBLE TO READ THIS FILE".

```

4. Для поля "Save file" напишем триггер на событие RETURN:
 IF NOT editor-1:SAVE-FILE(fill-in-3:SCREEN-VALUE)
 THEN MESSAGE "IT IS NOT POSSIBLE TO SAVE THIS FILE".
5. Напишем триггер на событие CHOOSE для кнопки "Delete line":
 IF NOT editor-1:DELETE-LINE()
 THEN MESSAGE "IT IS NOT POSSIBLE TO DELETE THIS LINE".
6. Сохраним процедуру в файле i02_1102.w и запустим программу на исполнение.

2.12. Объекты интерфейса типа SELECTION-LIST

Тип SELECTION-LIST применяется для представления строк. Он используется для выбора одной строки или нескольких строк (опции SINGLE или MULTIPLE во VIEW-AS фразе). Список строк может выглядеть как SCROLLBARS, а также изменяться во время исполнения программы.

Основные события: ENTRY, LEAVE, VALUE-CHANGED.

Основные атрибуты: HANDLE, HELP, LABEL, MODIFIED, SCREEN-VALUE, SENSITIVE, VISIBLE,

LIST-ITEMS - список строк, разделенных запятыми,

LIST-ITEMS-PAIRS – список пар метка-значение,

NUM-ITEMS - число выбираемых строк

Методы:

Метод	Описание
ADD-FIRST()	Добавляет строку в начало списка.
ADD-LAST()	Добавляет строку в конец списка.
DELETE()	Удаляет указанную строку.
ENTRY()	Возвращает значение указанной строки.
INSERT()	Вставляет новую строку перед указанной.
IS-SELECTED()	Возвращает значение true, если указанная строка является текущей.
LOOKUP()	Возвращает номер указанной строки.
REPLACE()	Замещает указанную строку новой.

Пример 2.12.1

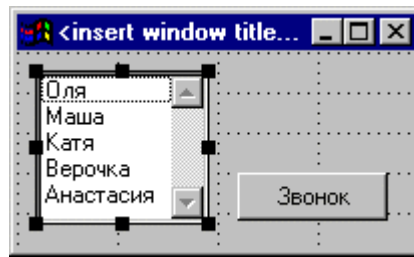
Программа позволяет просматривать список и дополнять его.

```


/*      DEFINE WIDGETS      */
DEFINE VARIABLE sel AS CHARACTER LABEL "list of possible years"
  INITIAL "1900" VIEW-AS SELECTION-LIST SINGLE
  LIST-ITEMS "1900","1920","1940","1960","1980","2000","2020" SORT
  SCROLLBAR-VERTICAL INNER-CHARS 15 INNER-LINES 7.
DEFINE VARIABLE y AS INTEGER LABEL "INSERT YEAR" FORMAT "9999".
DEFINE BUTTON btn-exit LABEL "exit".
/*      DEFINE FRAME      */
DEFINE FRAME frame1
  sel SKIP(2) y
  btn-exit WITH CENTERED NO-BOX NO-UNDERLINE SIDE-LABEL.
/*      DEFINE TRIGGERS      */
ON VALUE-CHANGED OF sel
DO:
  ASSIGN sel.
  MESSAGE "YOU CHOOSE " sel
  VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
END.
ON RETURN OF y
DO:
  ASSIGN y.
  sel:ADD-LAST(STRING(y)).
END.
/*      MAIN LOGIC      */
DISPLAY sel WITH FRAME frame1.
ENABLE ALL WITH FRAME frame1.
WAIT-FOR CHOOSE OF btn-exit.
  
```

Пример 2.12.2 (AB)

Создадим приложение, в котором используется SELECTION-LIST с парами метка-значение.



1. Откроем новое окно.

2. Разместим в окне объект интерфейса типа SELECTION-LIST (). В окне свойств объекта есть рабочее поле для создания списка значений, а над ним - radio-set для выбора вида этого списка. Выберем List-Items-Pairs и в рабочем поле напишем пары:

Оля,268-74-46,
Маша,678-77-45,
...
Анастасия,177-96-85

3. Создадим кнопку с меткой “Звонок” и напишем триггер для нее:
MESSAGE “Звонок по номеру “ Select-1:SCREEN-VALUE.

4. Продублируем триггер – те же действия должны выполняться при двойном щелчке мышью по элементу списка. Для этого возьмем оператор MESSAGE в Clipboard и здесь же, в Section Editor, перейдем к Select-1 (OF:). Событие Value-Changed нас не устраивает, поэтому щелкнем по кнопке New, в открывшемся окне укажем группу событий Portable Mouse Events, и из предложенного списка выберем MOUSE-SELECT-DBLCLICK. Вставим содержимое триггера.

5. Сохраним процедуру в файле i02_1202.w и запустим программу на исполнение.

2.13. Объекты интерфейса типа COMBO-BOX

COMBO-BOX совмещает в себе основные качества объектов FILL-IN и SELECTION-LIST. Он используется для ввода данных с возможностью выбора из готовых вариантов.

Типичные события: ENTRY, LEAVE, VALUE-CHANGED.

Типичные атрибуты: HANDLE, HELP, LABEL, MODIFIED, VALUE-CHANGED, SCREEN-VALUE, SENSITIVE, VISIBLE, LIST-ITEMS.


Методы:

Метод	Описание
ADD-FIRST()	Добавляет строку в начало списка.
ADD-LAST()	Добавляет строку в конец списка.
DELETE()	Удаляет указанную строку.
ENTRY()	Возвращает значение указанной строки.
INSERT()	Вставляет новую строку перед указанной.
LOOKUP()	Возвращает номер указанной строки.
REPLACE()	Замещает указанную строку новой.

Пример 2.13.1 (AB)

Создадим приложение, в котором используется COMBO-BOX с парами метка-значение. Объект COMBO-BOX будет использоваться для выбора цвета окраски фрейма.

1. Откроем новое окно.

2. Разместим в окне объект интерфейса типа COMBO-BOX(). В окне свойств объекта есть рабочее поле для создания списка пар, а над ним - radio-set для выбора вида этого списка. Выберем List-Items-Pairs и в рабочем поле напишем пары:

синий, 1, зеленый, 2, красный, 12

(в Приложении 4 приводится список кодов цветов, по умолчанию установленных в Progress). В поле Define As выберем тип INTEGER.

2. Событийное программирование и объекты интерфейса

3. Создадим следующий триггер для COMBO-BOX на событие VALUE-CHANGE:

ASSIGN COMBO-BOX-1.

FRAME DEFAULT-FRAME:BGCOLOR = COMBO-BOX-1.

4. Сохраним процедуру в файле i02_1301.w и запустим программу на исполнение.

2.14. ActiveX объекты

Progress позволяет использовать ActiveX объекты в программах на 4GL. В директории Progress\src\samples\activex приведены примеры использования ActiveX объектов. Особенно удобны в использовании те ActiveX объекты, которые вынесены на панель AppBuilder. Рассмотрим несколько примеров их использования.

Пример 2.14.1 (AB)

Создадим приложение, в котором используется объект таймер.



1. Откроем новое окно.



2. Разместим в окне объект таймер (). В окне свойств объекта установим значение атрибута Interval равным 600.

Property Editor - PSTimer	
(About)	...
Enabled	True
HonorProKeys	True
HonorReturnKey	False
Interval	600
Name	PSTimer
Taq	

3. Создадим FILL-IN поле без метки.

4. Напишем для таймера следующий триггер на событие OCX.Tick (найдем это событие через New в редакторе триггеров):

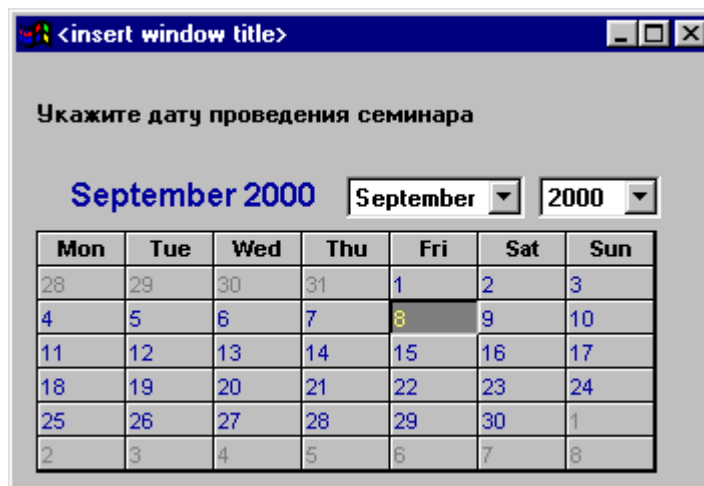
FILL-IN-1 = STRING(TIME,"HH:MM:SS").

DISPLAY FILL-IN-1 WITH FRAME DEFAULT-FRAME.

5. Сохраним процедуру в файле i02_1401.w и запустим программу на исполнение.

Пример 2.14.1 (AB)

Создадим приложение, в котором используется объект календарь.

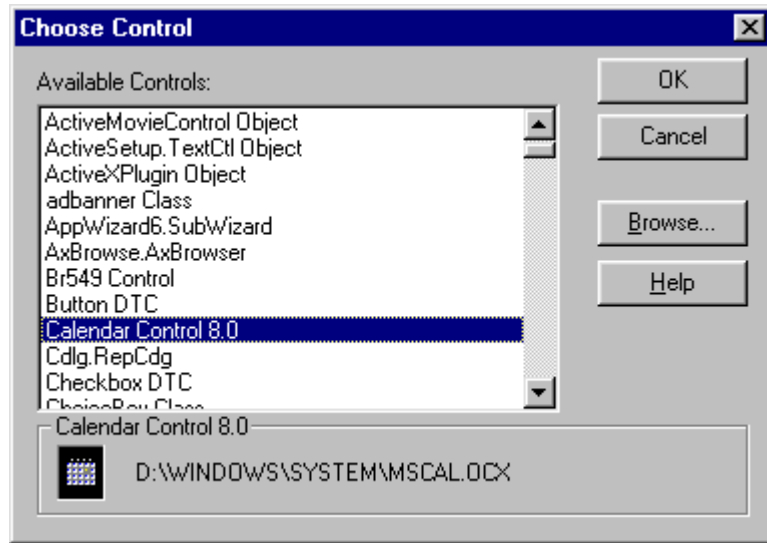


2. Событийное программирование и объекты интерфейса

1. Откроем новое окно.

2. Разместим в окне OCX-объект (). В появившемся списке выберем Calendar Control 8.0.

3. Разместим в окне текстовый объект с надписью "Укажите дату проведения семинара".



4. Напишем следующий триггер на событие OCX.DbClick:

```
MESSAGE "ВЫ ВЫБРАЛИ " + chCtrlFrame:Calendar:DAY + "/" +  
chCtrlFrame:Calendar:MONTH + "/" +  
chCtrlFrame:Calendar:YEAR.
```

5. Сохраним процедуру в файле i02_1402.w и запустим программу на исполнение.

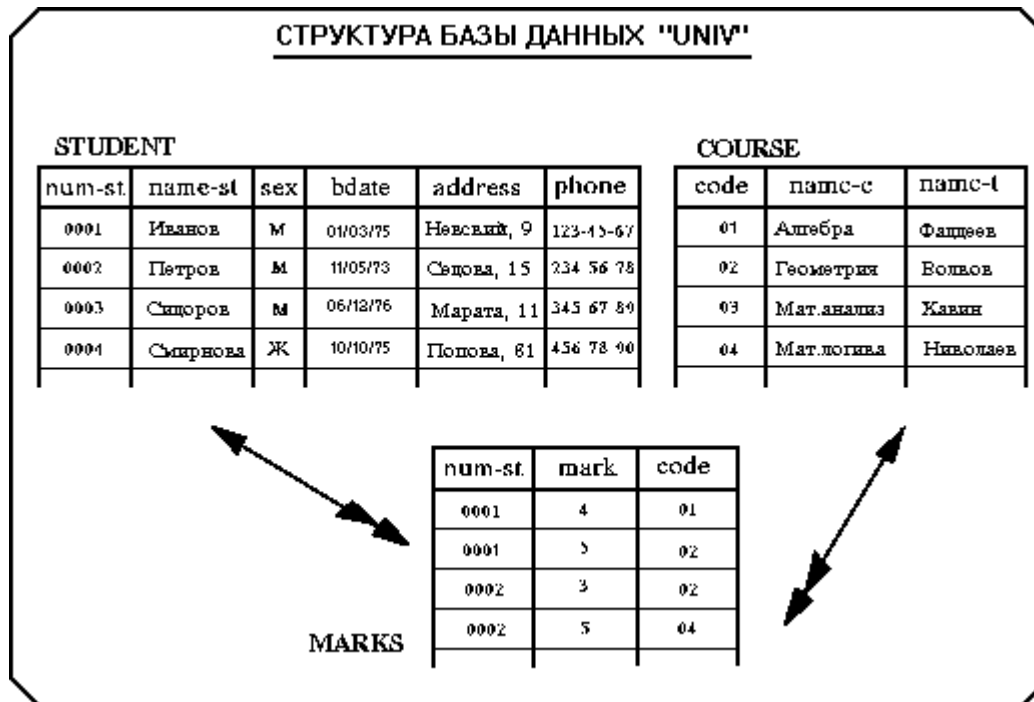
II. БАЗЫ ДАННЫХ

3. ОРГАНИЗАЦИЯ БАЗ ДАННЫХ

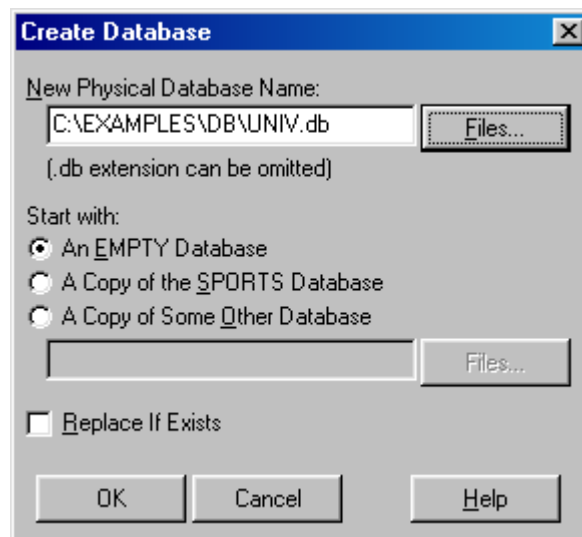
3.1. Создание базы данных

База данных в Progress - это место для хранения данных, индексов и секвенций. Создание таблиц, индексов и секвенций осуществляется в интерактивном режиме посредством компоненты DATA DICTIONARY.

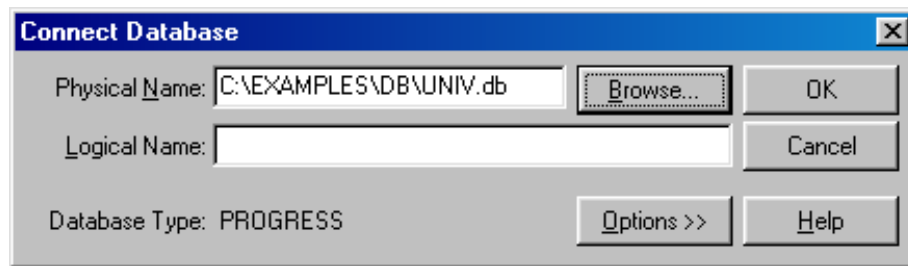
В этой главе мы начнем создавать базу данных UNIV, которая будет использоваться в большинстве примеров данного курса:



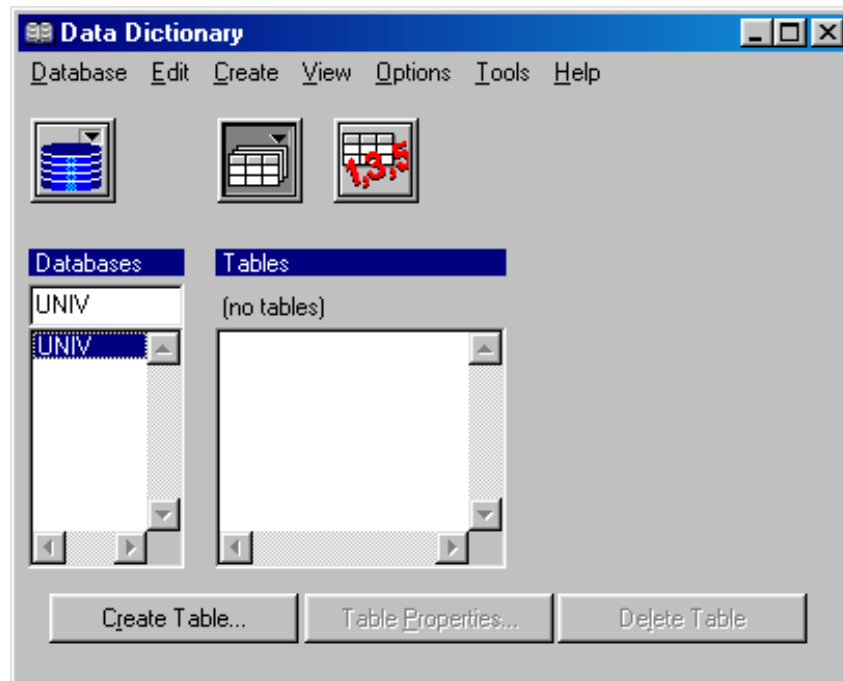
В окне Create Database выберем режим создания пустой (empty) базы данных, а в поле New Physical Name укажем полное имя файла базы данных.



В окне Connect Database поля, отличные от Physical Name, пока не заполняем.



В случае нормального выполнения команды Connect, увидим следующее диалоговое окно:



На физическом уровне база данных представляет собой несколько файлов, описание которых можно найти в приложении 5.

3.2. Создание таблиц

Создадим таблицу student: Create table....

При создании таблицы указываются следующие характеристики:

Table Name	- имя таблицы,
Dump File	- имя файла для dump (по умолчанию - имя таблицы);
Label	- метка, которая будет использована в error messages;
Description	- описание таблицы;
Hidden	- специфицируется "невидимая таблица" (по умолчанию "no");

Возможности, скрытые за полями и кнопками Area, Replication, Frozen, Triggers..., Validation..., String Attrs... будут рассмотрены позже.

3.3. Создание полей

При создании полей таблицы посредством DATA DICTIONARY в интерактивном режиме определяются следующие характеристики полей:

- Field Name - имя поля.
- Date Type - тип поля.
- Format - формат вывода поля.
- Label - метка поля (если оставить "?", то в качестве метки будет использоваться имя поля).
- Column Label - метка поля в несколько строк (строки отделяются друг от друга символом "!").
- Initial - начальное значение.
- Order - порядок изображения полей (по умолч. - через 10 номеров).
- Decimals - количество цифр в дробной части (определяется только для полей типа decimal).
- Description - неформальное описание поля.
- Help Text - строка подсказки при вводе в поле.
- Mandatory - обязательность заполнения поля. По умолчанию - "no".
- Case-Sensitive - чувствительность к прописным и строчным буквам.
- Extent - число элементов, если поле является массивом.

Для таблицы student создадим следующие поля:

Name	Type	Format	Label
Num_st	INTEGER	99999	номер студ.
Name_st	CHARACTER	x(15)	имя
Sex	LOGICAL	м/ж	пол
Address	CHARACTER	x(15)	адрес
Bdate	DATE	99/99/99	дата рожд.
Phone	CHARACTER	9-99-99-99	телефон

3.4. Описание индексов

Индексы определяются также посредством DATA DICTIONARY. Их использование в Progress обеспечивает:

- ускорение доступа к записи;
- автоматическое упорядочение записей при выводе;
- контроль за уникальностью значений;
- ускорение поиска соответствующих записей в связанных таблицах.

Имя индекса может совпадать с именем поля в таблице. Индексы могут быть составными - состоять из нескольких полей (компонент). При определении индекса в интерактивном режиме уточняются следующие характеристики индекса:

Index Name - имя индекса.
 Description - неформальное описание индекса.
 Primary - индекс, используемый по умолчанию.
 Active - активный индекс.
 Unique - уникальность индекса.
 Word index - специальный индекс для выборки по словам внутри символьного поля, см. Прилож.6.
 Abbreviated - возможность выборки по префиксу символьного поля.
 Ascending - построение индекса по возрастанию.
 Descending - построение индекса по убыванию.

Неактивный индекс может потребоваться для ускорения работы с большими объемами данных. Последующая активизация индекса возможна при помощи утилиты proutil с опцией idxbuild.

Для таблицы **student**, описанной в разделе 3.3., определим через DATA DICTIONARY индексы со следующими свойствами:

Index name	Primary	Unique	Field	Ascending	Word Index
Num_st	Yes	Yes	Num_st	Yes	No
Name_st	No	No	Name_st	Yes	No
Address	No	No	Address	Yes	Yes

3.5. Секвенции

При работе с базой данных могут понадобиться числовые последовательности, которые затем будут интерпретироваться самыми разнообразными способами (номера зачетных книжек студентов, шифры для работы на компьютере и т.д.). Такие последовательности (**sequence**) Progress может генерировать автоматически.

При описании секвенции нужно указать:

Sequence name - имя секвенции
 Initial name - начальное значение (по умолчанию - 0);
 Increment by - шаг секвенции (по умолчанию - 1);
 Upper limit - максимальное значение (по умолчанию - ?);
 Cycle limit - является ли секвенция циклической.

Для работы с секвенциями существуют специальные операторы и функции:

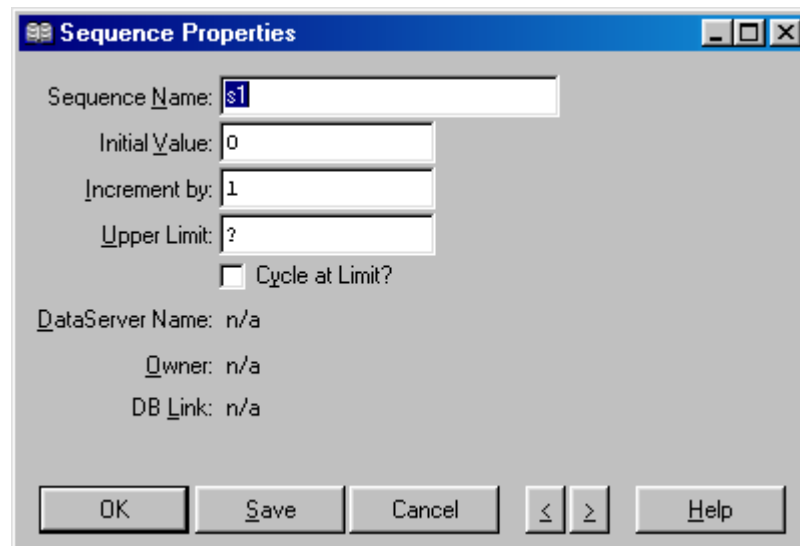
CURRENT-VALUE (name-seq) = i - оператор, устанавливающий текущее значение секвенции равным i;

NEXT-VALUE (name-seq) - функция, выдающая очередное значение секвенции (при этом изменяется текущее значение).

CURRENT-VALUE (name-seq) - функция, выдающая текущее значение секвенции.

В разделе 4.1. будет приведен пример использования этих операторов и функций.

Создадим в Data Dictionary секвенцию s1 с начальным значением 0 и шагом 1. В дальнейшем она будет использоваться для генерации номеров студентов в таблице **student**.



3.6. Триггеры базы данных

Существуют два вида **триггеров** базы данных - session и schema. **Session** триггеры создаются как часть Progress-процедуры и выполняются только в рамках данного приложения. **Schema** триггеры создаются через DATA DICTIONARY и выполняются в каждой клиентской сессии, вызвавшей соответствующее событие в базе данных.

На уровне таблицы существует серия событий, по которым строятся триггеры. Например:

CREATE	- добавление новой записи;
DELETE	- удаление записи;
FIND	- чтение записи;
WRITE	- изменение содержимого записи.

События с префиксом "REPLICATION-" предназначены для создания shema-триггеров, в которых программист может описать механизм репликаций при обработке распределенной базы данных. REPLICATION- триггер выполняется после одноименного стандартного триггера (например, REPLICATION-CREATE – после CREATE).

На уровне полей существует единственное событие, по которому может быть создан триггер:

ASSIGN	- изменение содержимого поля.
--------	-------------------------------

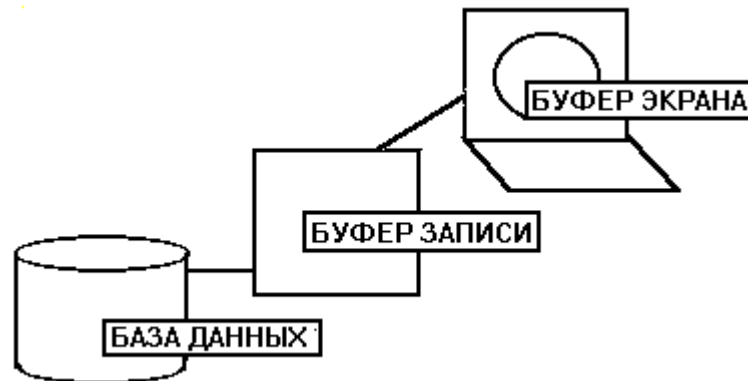
Если на одно и то же событие для одной и той же таблицы (или поля) существуют и session, и schema триггеры, они будут выполняться последовательно - сначала session, затем schema триггер (исключение – триггеры на событие FIND). Если же schema триггер был специфицирован как замещаемый (overridable), то session триггер может заместить schema триггер.

В разделе 4.7. будут приведены примеры использования триггеров.

4. РАБОТА С ДАННЫМИ

4.1. Добавление записей к таблице

Существуют три уровня локализации данных в Progress:



Операторы, обрабатывающие данные, переносят данные между этими тремя уровнями.

Следующая процедура демонстрирует добавление одной записи в таблицу **student**.

```
CREATE student.
UPDATE student.
```

Оператор CREATE создает новую запись в таблице и устанавливает все поля в соответствии с их начальными значениями, а также создает копию этой записи в буфере записи. Оператор UPDATE реализует запросы на ввод данных и помещает введенные данные как в буфер экрана, так и в буфер записи, а по концу процедурного блока обновляет их в базе. Оба оператора могут использоваться с опцией NO-ERROR.

Эта же процедура может быть записана с помощью оператора INSERT, который является сокращением CREATE + UPDATE.

```
INSERT student.
```

Добавление нескольких записей реализуется следующей процедурой:

```
REPEAT:
  INSERT student WITH 1 COLUMN.
END.
```

Напомним, что выход из такого цикла осуществляется по нажатию на клавишу END-ERROR. При этом Progress отменит все изменения, произведенные во время последней итерации.

Пример 4.1.1

В этом примере продолжается создание новых записей в таблице **student**. Для заполнения поля **num_st** здесь используется секвенция s1. Поскольку в таблице уже есть несколько записей о студентах, установим текущее значение секвенции равным номеру последнего добавленного в таблицу студента.

```
CURRENT-VALUE(s1) = 6.
REPEAT:
  CREATE student.
  num_st = NEXT-VALUE(s1).
  DISPLAY num_st.
  UPDATE name_st sex bdate address phone.
END.
```

4.2. Выборка данных из таблиц

Следующая процедура демонстрирует вывод содержимого таблицы с использованием FOR EACH оператора. Отметим, что этот оператор обладает свойствами блока.

```
FOR EACH student:
  DISPLAY student.
END.
```

Оператор FOR EACH читает записи из базы данных в буфер записи в соответствии с первичным индексом.

Индекс для чтения записей может быть указан явно в заголовке оператора FOR EACH с помощью конструкта USE-INDEX:

```
FOR EACH student USE-INDEX name_st:
    DISPLAY student.
END.
```

Если подходящего индекса для чтения записей нет, то выражения, в соответствии с которыми осуществляется выбор записей, могут быть явно указаны в заголовке FOR EACH оператора:

```
FOR EACH student BY phone:
    DISPLAY student.
END.
```

Следует заметить, что в этом случае по заданному выражению сначала будет построен временный индекс, и лишь затем записи будут выбраны из базы.

В приведенном выше примере записи выводятся по возрастанию значений указанных полей, однако порядок следования может быть изменен на противоположный, если в заголовке FOR EACH блока указано ключевое слово DESCENDING:

```
FOR EACH student BY phone DESCENDING:
    DISPLAY student.
END.
```

В операторе DISPLAY можно указывать не всю запись, а лишь некоторые поля. Однако нельзя перемешивать в одном операторе DISPLAY имена таблиц и полей.

```
FOR EACH student BY phone DESCENDING:
    DISPLAY num_st name_st phone.
END.
```

Использование конструкта WHERE в заголовке FOR EACH оператора позволяет задавать фильтр для выводимых записей:

```
FOR EACH student WHERE sex:
    DISPLAY num_st name_st phone sex.
END.
```

Конструкт WHERE может использоваться в сочетании с BY и USE-INDEX. Однако BY должен следовать за WHERE. Сочетание с USE-INDEX возможно в любом порядке.

Следующие примеры демонстрируют несколько вариантов использования оператора FOR EACH.

Пример 4.2.1

Программа выводит записи о студентах, фамилии которых начинаются на "А".

```
FOR EACH student WHERE name_st BEGINS "A":
    DISPLAY num_st name_st phone sex.
END.
```

Пример 4.2.2

Программа выводит записи о студентах, фамилии которых заканчиваются на "ов".

```
FOR EACH student WHERE name_st MATCHES "*ов":
    DISPLAY num_st name_st phone sex.
END.
```

В шаблоне знак "*" означает любое количество символов (знак "." заменяет один символ).

Пример 4.2.3

Использование конструкта WHERE в сочетании с USE-INDEX.

```
FOR EACH student
    WHERE num_st > 5 and num_st < 10 USE-INDEX name_st:
    DISPLAY num_st name_st phone sex.
END.
```

Примеры 4.2.4 и 4.2.5 демонстрируют использование конструкта WHERE в сочетании с WORD-индексом (напомним, что по полю **address** был построен WORD-индекс).

Синтаксис для его использования:

```
WHERE field-name CONTAINS 'string-expression'
```


В string-expression могут содержаться круглые скобки и следующие знаки операций: **&** ! ^ , здесь **&** представляет логическое AND, а ! и ^ - логическое OR. Слова в string-expression не заключаются в кавычки и могут содержать символ * для обозначения любой подстроки.

В приложении 6 рассказано, что подразумевается в Progress под словами и разделителями, и как их можно переопределить.

Пример 4.2.4

Программа выводит записи о студентах, в адресе которых присутствует число 2.

```
FOR EACH student WHERE address CONTAINS "2":
    DISPLAY student.
END.
```

Пример 4.2.5

Программа выводит записи о студентах, в адресе которых присутствует слово 'Рига' или 'Петербург'.

```
FOR EACH student WHERE address CONTAINS "Рига!Петербург":
    DISPLAY student.
END.
```

Чтение записей может быть организовано и с помощью операторов FIND FIRST, FIND NEXT, FIND LAST и FIND PREV. Операторы FIND ищут нужную запись в таблице в соответствии с первичным индексом и размещают ее в буфере записи. Операторы FIND, как и оператор FOR EACH, могут использоваться в сочетании с конструктами WHERE и USE-INDEX.

Например:

```
FIND student WHERE num_st = 5.
FIND NEXT student USE-INDEX name_st.
```

Если оператор FIND не находит соответствующей записи в таблице - Progress вырабатывает программное прерывание. Реакция процедуры на это прерывание зависит от контекста, в котором используется оператор FIND. Это либо сообщение об ошибке, либо выход из блока, содержащего оператор FIND. Директива NO-ERROR, как обычно, может быть использована для того, чтобы Progress не выдавал сообщения об ошибке. Проанализировать такую ошибку можно не только через системную переменную ERROR-STATUS, но и через логическую функцию AVAILABLE, которая выдает значение **true** (если соответствующая запись есть в буфере) или **false** (в противном случае).

Например:

```
FIND student WHERE num_st = 5 NO-ERROR.
IF NOT AVAILABLE student THEN MESSAGE "Нет студента с таким номером".
```

Пример 4.2.6

Выборка из таблицы множества записей путем использования оператора FIND NEXT в цикле REPEAT.

```
REPEAT:
    FIND NEXT student.
    DISPLAY student.
END.
```

Заметим, что оператор FIND NEXT начинает поиск с первой записи в файле, а оператор FIND PREV - с последней.

Операторы FIND эффективно используются в сочетании с функциями RECID и ROWID (последняя появилась только в Progress V8). Эти функции возвращают уникальный адрес записи в таблице. Аргументом является запись, расположенная в буфере записи, результат имеет тип, соответственно, RECID или ROWID. Синтаксис:

```
RECID(table-name)
ROWID(table-name)
```

Пример 4.2.7

Поиск в таблице student одного из самых старших студентов.

```
DEFINE VARIABLE recnum AS ROWID.
DEFINE VARIABLE bd AS DATE.
FIND FIRST student WHERE bdate <> ?.
recnum = ROWID(student).
bd = bdate.
REPEAT:
```

```

FIND NEXT student WHERE bdate <> ?.
IF bdate < bd THEN DO:
    bd = bdate.
    recnum = ROWID(student).
END.
END.
FIND student WHERE ROWID(student) = recnum.
DISPLAY name_st bdate.

```

Работа с записями может быть организована с помощью конструктора QUERY так, как это показано в примере 4.2.8. Назначение QUERY - формирование списка адресов записей в соответствии с record-phraze оператора OPEN QUERY. Доступ к записи осуществляется при помощи оператора GET с одной из опций: FIRST, LAST, NEXT или PREV. Оператор GET в соответствии со списком, сформированным в QUERY, выбирает запись из таблицы и помещает ее в буфер записи. Следует отметить, что в отличие от оператора FIND оператор GET не вырабатывает программного прерывания, и выход из блока, в котором используется оператор GET, нужно предусматривать явно.

Пример 4.2.8

```

DEFINE QUERY st_query FOR student.
OPEN QUERY st_query FOR EACH student.
GET FIRST st_query.
DO WHILE AVAILABLE(student):
    DISPLAY student.
    GET NEXT st_query.
    PAUSE.
END.

```

- В операторе DEFINE QUERY для сложных запросов следует использовать опцию SCROLLING для навигации по двум направлениям.

Работа с QUERY может быть организована через специальные методы, предусмотренные для объекта QUERY. Следующий пример демонстрирует использование таких методов.

Пример 4.2.9

```

DEFINE VARIABLE s_query AS HANDLE.
DEFINE QUERY st_query FOR student.
s_query = QUERY st_query:HANDLE.
s_query:QUERY-PREPARE("for each student").
s_query:QUERY-OPEN.
s_query:GET-FIRST.
DO WHILE NOT (s_query:QUERY-OFF-END):
    DISPLAY student WITH 1 COLUMN.
    s_query:GET-NEXT.
    PAUSE.
END.

```

В следующем примере демонстрируется возможное использование конструктора QUERY в АВ.

Пример 4.2.10 (АВ)

Просмотр полей из таблицы student.

номер ст.:	0001	First
имя ст.:	Иванов Саша	Next
адрес:	Невский 24-15	Prev
дата рожд.:	01/02/70	Last

1. Откроем новое окно в АВ.



2. Разместим в окне поля из таблицы student ().

3. Разместим в окне стандартные кнопки с метками First, Prev, Next и Last для навигации по записям таблицы (список стандартных кнопок появляется при нажатии на правую кнопку мыши при выборе объекта button).

4. Откроем Property для фрейма и обратим внимание на Query и его имя (default-frame).

5. Просмотрим триггеры для кнопок.

6. Запишем программу в файл под именем i04_0210.w. Запустим программу на исполнение.

Следующие примеры демонстрируют еще один объект интерфейса - BROWSE, который используется для вывода записей. Объект BROWSE применяется в тесной связи с QUERY, так как он выводит именно те записи, которые были выбраны оператором OPEN QUERY.

Основные события, связанные с BROWSE:

ENTRY - фокусировка объекта интерфейса
 LEAVE - снятие фокусировки
 ITERATION-CHANGED
 VALUE-CHANGED
 ROW-LEAVE
 ROW-ENTRY
 HOME
 END

Некоторые атрибуты: HANDLE, HELP, TITLE, SENSITIVE, VISIBLE, **MAX-DATA-GUESS** - определяет количество записей в query (для согласования вертикального бегунка), **NUM-LOCKED-COLUMNS** - количество фиксированных левых колонок (они будут оставаться на виду при горизонтальном скроллинге).

Типичные методы:

Метод	Описание
SELECT-PREV-ROW()	Выбирает предыдущую строку.
SELECT-NEXT-ROW()	Выбирает следующую строку.
MOVE-COLUMN(n,n)	Меняет местами колонки.
INSERT-ROW("AFTER" "BEFORE")	Вставляет пустую строку (только для редактируемого browse).
DELETE-SELECTED-ROWS()	Удаляет выделенные строки из browse и query.

Следующий пример демонстрирует, как работа с объектом BROWSE может быть организована в программе.

Пример 4.2.11

```
/* DEFINE QUERY */
DEFINE QUERY q_st FOR student .
/* DEFINE WIDGETS */
DEFINE BROWSE b_st QUERY q_st DISPLAY num_st name_st
      WITH 5 DOWN SEPARATORS.
DEFINE BUTTON exit .
/* DEFINE FRAME */
DEFINE FRAME f_st
      b_st SKIP(2)
      num_st name_st
      address bdate
      sex phone
      exit
      WITH 1 COLUMN SIDE-LABELS
      COLUMN 10 NO-BOX.
/* DEFINE TRIGGERS */
ON VALUE-CHANGED OF b_st
DO:
      DISPLAY student WITH FRAME f_st.
END.
/* MAIN LOGIC */
OPEN QUERY q_st FOR EACH student .
ENABLE ALL WITH FRAME f_st.
WAIT-FOR CHOOSE OF exit.
```

Следующие примеры демонстрируют приемы работы с объектами типа BROWSE в AB.

Пример 4.2.12 (AB)

Просмотр списка студентов.

номер ст.	имя ст.
0003	Семенов Ваня
0004	Денисов Петр
0005	Антонов Семен

номер ст.:	0005
имя ст.:	Антонов Семен
дата рожд.:	12/11/75
адрес:	Майский пр.12-
пол:	М
телефон:	3-33-22-33

1. Откроем новое окно в АВ.



2. Выберем в палитре объект типа BROWSE () и свяжем его с полями num_st и name_st из таблицы student.

3. Запустим незаконченную программу на исполнение и обратим внимание, что навигация по записям осуществляется как с помощью мыши, так и с помощью клавиш управления курсором.



4. Выберем все поля из таблицы student () и разместим их в окне в виде fill-in объектов.

5. Напишем триггер на событие VALUE-CHANGED для BROWSE:
DISPLAY student WITH FRAME default-frame.

6. В разделе Main Block после оператора RUN ENABLE_UI добавим:
APPLY "VALUE-CHANGED" TO browse-1.

7. Запишем программу в файл под именем i04_0212.w. Запустим программу на исполнение.

Пример 4.2.13 (AB)

Следующий пример демонстрирует, как для одного и того же BROWSE открываются различные QUERY.

num-st	name-st	дата рожд.
0001	Иванов Саша	01/02/70
0002	Сидорова Даша	01/03/75
0005	Антонов Семен	12/11/75

BY ASCENDING num-st
BY ASCENDING name-st
BY ASCENDING bdate

1. Откроем новое окно в АВ.



2. Выберем в палитре объект типа BROWSE () и свяжем его с полями num_st, name_st и bdate из таблицы student.

3. Разместим в окне в кнопку с меткой "BY ASCENDING num_st " и напишем триггер для нее:
OPEN QUERY browse-1 FOR EACH student.
VIEW browse-1.

4. Разместим в окне в кнопку с меткой "BY ASCENDING name_st " и напишем триггер для нее:
OPEN QUERY browse-1 FOR EACH student USE-INDEX name_st.
VIEW browse-1.

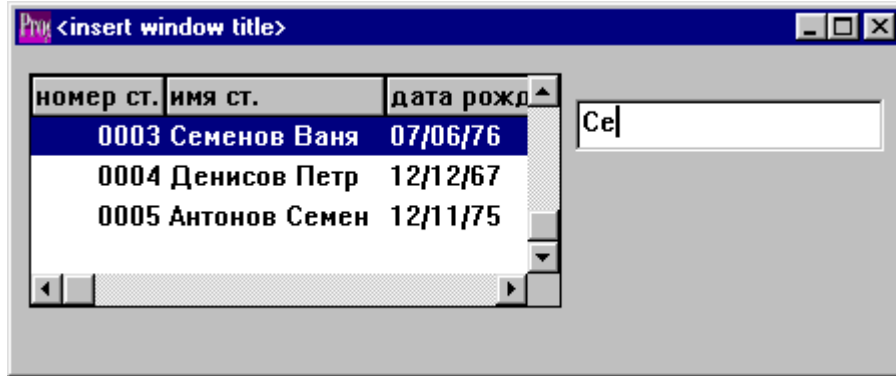
5. Разместим в окне в кнопку с меткой "BY ASCENDING bdate " и напишем триггер для нее:
OPEN QUERY browse-1 FOR EACH student BY bdate.
VIEW browse-1.

6. В разделе Main Block после оператора RUN ENABLE_UI добавим:
browse-1:VISIBLE = NO.

7. Запишем программу в файл под именем i04_0213.w. Запустим программу на исполнение.

Приведенный ниже пример демонстрирует работу BROWSE в сочетании с FILL-IN, используемым в качестве локатора.

Пример 4.2.14 (AB)



1. Откроем новое окно в AB.



2. Выберем в палитре объект типа BROWSE () и свяжем его с полями name_st, num_st и bdate из таблицы student.

3. Разместим в окне в fill-in объект без метки и напишем триггер для него на событие ANY-KEY:

FIND FIRST student WHERE

name_st BEGINS fill-in-2:SCREEN-VALUE + KEYFUNCTION(LASTKEY)

NO-ERROR.

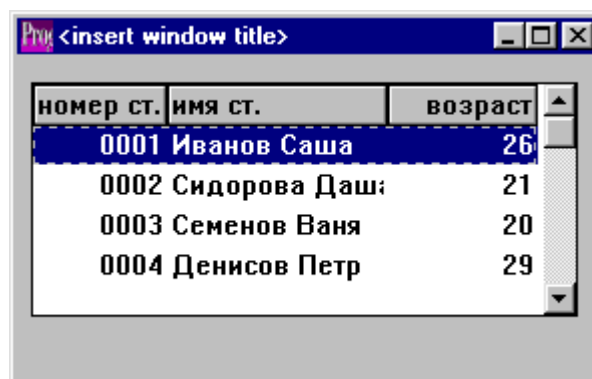
IF AVAILABLE student THEN REPOSITION browse-2 TO ROWID ROWID(student).

4. Существует еще один - встроенный - способ навигации по BROWSE. При активном browse нажатие символа на клавиатуре вызывает прокрутку до строки, в которой левый столбец начинается с указанного символа. Для того, чтобы проверить этот способ, поменяем местами поля num_st и name_st в BROWSE: Property -> Fields. Запустим программу на исполнение.

5. Запишем программу в файл под именем i04_0214.w.


Пример 4.2.15 (AB)

Следующий пример демонстрирует BROWSE, в котором осуществляется вывод полей num_st, name_st и вычисляемого поля с возрастом студента.



1. Откроем новое окно в AB.

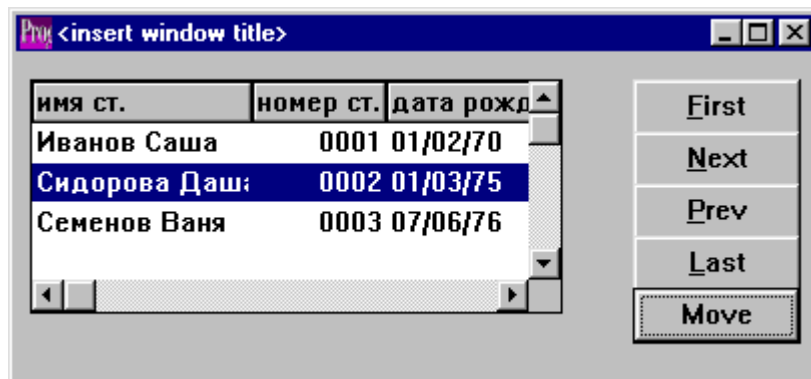


2. Выберем в палитре объект типа BROWSE () и свяжем его с полями num_st и name_st из таблицы student. Добавим вычисляемое поле (Calculated Field) с возрастом студента: YEAR(TODAY) - YEAR(bdate).

3. Запишем программу в файл под именем i04_0215.w. Запустим программу на исполнение.

Методы SELECT-PREV-ROW(), SELECT-NEXT-ROW() и MOVE-COLUMN(n,n) могут использоваться для навигации по записям внутри BROWSE и для перестановки столбцов, как это показано в следующем примере. В этом же примере предлагается познакомиться с приемами интерактивного изменения условий сортировки и выборки данных для представления их в BROWSE.

Пример 4.2.16 (AB)



1. Откроем новое окно.



2. Выберем в палитре объект типа BROWSE () и свяжем его с полями num_st, name_st и bdate из таблицы student.

3. Разместим в окне в кнопку с меткой "First " и напишем триггер для нее:
APPLY "HOME" TO browse-1.

4. Разместим в окне в кнопку с меткой "Next " и напишем триггер для нее:
browse-1:SELECT-NEXT-ROW().

5. Разместим в окне в кнопку с меткой "Prev " и напишем триггер для нее:
browse-1:SELECT-PREV-ROW().

6. Разместим в окне в кнопку с меткой "Last " и напишем триггер для нее:
APPLY "END" TO BROWSE-1.

7. Разместим в окне кнопку с меткой "Move " и напишем триггер для нее:
browse-1:MOVE-COLUMN(1,2).

8. Запустим программу на исполнение.

10. Установим для BROWSE сортировку по полю bdate: Property -> Query -> Sort. Запустим программу на исполнение.

11. Установим для browse выборку по условию num-st > 5 AND num-st < 15: Property -> Query -> Where. Запустим программу на исполнение.

12. Запишем программу в файл под именем i04_0216.w.

4.3. Корректировка записей

Корректировка записей осуществляется оператором UPDATE, который выводит запись из буфера записи в буфер экрана и по окончании редактирования возвращает запись в базу.

Следующие процедуры демонстрируют корректировку записей в FOR EACH блоке:

```
FOR EACH student:
    UPDATE student.
END.
```

и в REPEAT блоке:

```
REPEAT:
    FIND NEXT student.
    UPDATE address phone.
END.
```

Оператор UPDATE может использоваться с опцией NO-ERROR. Следующий пример демонстрирует, как в этом случае можно идентифицировать допущенные ошибки.

- Заметим, что использование опции NO-ERROR в таком большом операторе как "UPDATE student NO-ERROR." приведет к забавному результату - все накопившиеся сообщения об ошибках пользователь получит только по завершению редактирования записи. Для сравнения можно посмотреть на работу этой же программы с измененным оператором - "UPDATE num_st NO-ERROR."

Пример 4.3.1

```

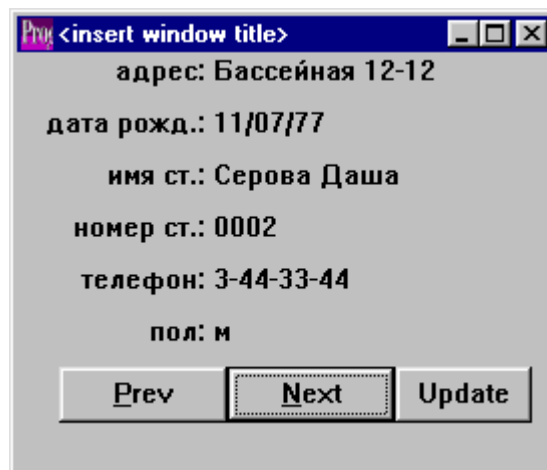
DEFINE VARIABLE I AS INTEGER.
CREATE STUDENT.
UPDATE STUDENT NO-ERROR.
IF ERROR-STATUS:ERROR
THEN DO I = 1 TO ERROR-STATUS:NUM-MESSAGES:
  MESSAGE IF ERROR-STATUS:GET-NUMBER(I)= 80
    THEN "MONTH MUST BE BETWEEN 1 AND 12!"
    ELSE IF ERROR-STATUS:GET-NUMBER(I)= 132
      THEN "YOU CAN NOT USE THIS NUMBER FOR STUDENT"
      ELSE "UNKNOWN ERROR"
  VIEW-AS ALERT-BOX INFORMATION BUTTON OK.
END.

```

Редактирование данных рекомендуется проводить в диалоговых окнах (DIALOG-BOX). Диалоговое окно представляет собой фрейм с некоторыми оконными свойствами и является **модальным** - обязывает пользователя либо довести редактирование до конца, либо полностью отказаться от него.


Пример 4.3.2 (AB)

Редактирование таблицы student.



1. Откроем новое окно в AB.



2. Выберем все поля из таблицы student () и разместим их в окне в виде text объектов (войти в Property каждого поля и отметить опцию View-as-Text или снять опцию Enable).

- Групповые действия с объектами интерфейса можно выполнить и с помощью окна Group Properties, которое вызывается через Window -> Group Properties Window. Группа объектов в этот момент должна быть выделена. В данном примере можно установить Enable – по для всех полей.

3. Разместим в окне две кнопки с метками Prev и Next (возьмем готовые шаблоны: щелкнем правой кнопкой мыши по кнопке button на палитре объектов и в открывшемся меню выберем сначала Prev, а затем и Next) для навигации по записям таблицы.

4. Разместим в окне кнопку с меткой Update и напишем триггер для нее:

```

UPDATE student WITH VIEW-AS DIALOG-BOX.
DISPLAY student WITH FRAME DEFAULT-FRAME.

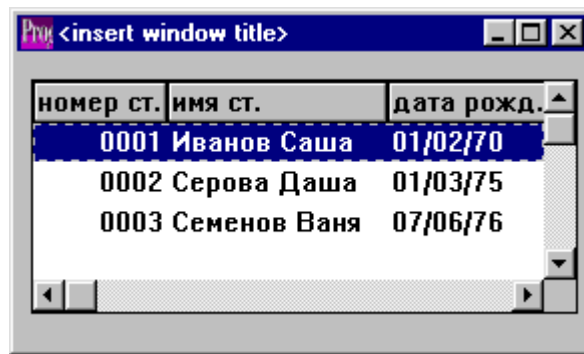
```

По нажатию пользователем кнопки Update редактирование записи будет производиться в непоименованном диалоговом фрейме (VIEW-AS DIALOG-BOX). По нажатию клавиши Enter в последнем поле будет приниматься изменение записи, а по нажатию Esc – отменяться.

5. Запишем программу в файл под именем i04_0302.w. Запустим программу на исполнение.

Пример 4.3.3 (AB)

Следующий пример демонстрирует создание редактируемого BROWSE.



1. Откроем новое окно.



2. Выберем в палитре объект типа BROWSE () и свяжем его с полями num_st, name_st и bdate из таблицы student.

3. Сразу после выбора полей для BROWSE в окне Column Editor отметим как Enabled поля name_st и bdate (в списке Fields in Browse возле этих полей появятся звездочки).

4. Запишем программу в файл под именем i04_0303.w. Запустим программу на исполнение и убедимся, что поля name_st и bdate могут быть отредактированы.

В редактируемых BROWSE могут использоваться события LEAVE и ENTRY на уровне полей BROWSE.

Пример 4.3.4 (AB)

1. Откроем процедуру из файла i04_0303.w.

2. Во встроенном редакторе (в списке объектов OF:) выберем объект bdate и напишем триггер для него на событие ENTRY:

MESSAGE "Вы уверены, что нужно редактировать дату рождения?".

3. Напишем триггер на событие LEAVE для того же объекта:

IF bdate:MODIFIED IN BROWSE BROWSE-1 THEN bdate:BGCOLOR = 3.

4. Запишем программу в файл под именем i04_0304.w. Запустим программу на исполнение.

Пример 4.3.5 (AB)

Пример демонстрирует редактирование записей таблицы student в диалоговом фрейме (DIALOG-BOX с кнопками завершения и отката, оформленном в виде include-файла:

DEFINE BUTTON btn-ok LABEL "OK" AUTO-GO.

DEFINE BUTTON btn-cancel LABEL "Cancel" AUTO-ENDKEY.

DEFINE FRAME dial-frame

student.num_st student.name_st

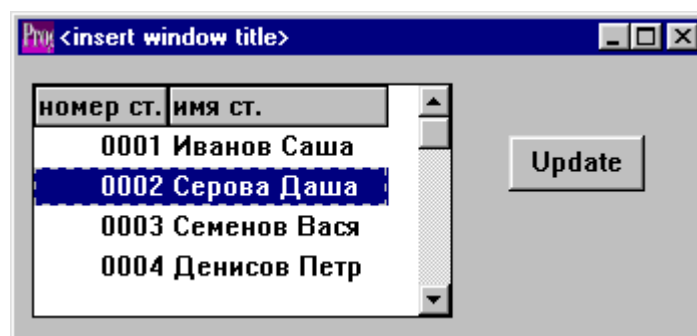
student.address student.bdate

student.sex student.phone


btn-ok btn-cancel WITH 1 COLUMN VIEW-AS DIALOG-BOX.

Этот текст сохраним в файле i04_dial.i, в дальнейшем он будет использоваться для редактирования таблицы student.

1. Откроем новое окно.





2. Выберем в палитре объект типа BROWSE () и свяжем его с полями num_st , name_st из таблицы student.
3. В раздел Definition добавим следующий include: {c:\examples\part4\i04_dial.i}
4. Разместим в окне кнопку Update и напишем триггер для нее на событие CHOOSE:


```
GET CURRENT browse-1 SHARE-LOCK.
ENABLE ALL WITH FRAME dial-frame.
UPDATE student WITH FRAME dial-frame.
DISPLAY num_st name_st WITH BROWSE browse-1.
```

 - Оператор "GET CURRENT browse-1 SHARE-LOCK." позволяет повысить "статус" текущей записи до уровня, позволяющего редактировать эту запись.
5. Запишем программу в файл под именем i04_0305.w. Запустим программу на исполнение.

4.4. Удаление записей

Удаление записей осуществляется оператором DELETE, который удаляет запись как из буфера записи, так и из таблицы.

Можно удалить только ту запись, которая была расположена в буфере записи одним из операторов: CREATE, FIND, FOR EACH или INSERT. Например, так удаляются студенты с заданными номерами:

```
FOR EACH student
    WHERE student.num_st < 20 AND student.num_st > 15:
    DELETE student.
END.
```

Оператор DELETE может использоваться с опциями VALIDATE и NO-ERROR:

```
DELETE table-name VALIDATE(condition,msg-expression) NO-ERROR
```

Опция VALIDATE используется для спецификации логического выражения (condition). Если выражение вырабатывает значение true - запись удаляется, в противном случае выдается предусмотренное сообщение об ошибке (msg-expression).

Пример 4.4.1

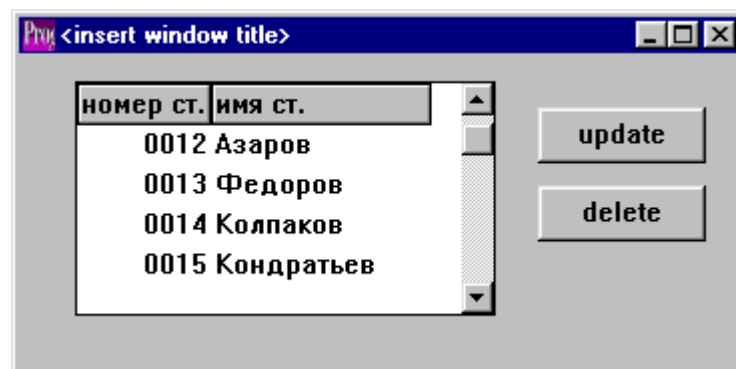
Приведенная ниже программа позволяет удалить лишь некоторых студентов.

```
FOR EACH student:
    DISPLAY student.
    MESSAGE "Are you sure?" VIEW-AS
    ALERT-BOX QUESTION BUTTONS YES-NO UPDATE b AS LOGICAL.
    IF b
    THEN DELETE student
    VALIDATE(sex, "Lady don't touch!").
END.
```

Пример 4.4.2 (AB)

Следующий пример демонстрирует, как можно удалять записи из таблицы student, используя при этом BROWSE для просмотра краткой информации о студенте.

1. Откроем процедуру из файла i04_0305.w.
2. Разместим в окне кнопку Delete и напишем триггер для нее на событие CHOOSE:



```

MESSAGE "Delete student?"
  VIEW-AS ALERT-BOX QUESTION
  BUTTONS yes-no UPDATE continue-ok AS LOGICAL.
  IF continue-ok
  THEN DO:
    GET CURRENT browse-1 SHARE-LOCK.
    DELETE student.
    browse-1:DELETE-SELECTED-ROWS().
  END.

```

3. Запишем программу в файл под именем i04_0402.w. Запустим программу на исполнение.

4.5. Связные таблицы

Определим новые таблицы в нашей базе: **course** и **marks**.

Таблица **course** состоит из следующих полей:

Name-field	Type	Format	Label
Code	INTEGER	99	код предмета
Name_c	CHARACTER	x(20)	назв. предмета
Name_t	CHARACTER	x(20)	фамилия лектора

Индекс к таблице **course**:

Index name	Primary	Unique	Field	Ascending	Word Index
Code	Yes	Yes	Code	Yes	No

Создадим в базе данных секвенцию S2 с начальным значением 0 и шагом 1. В дальнейшем она будет использоваться для генерации кодов в таблице **course**.

Таблица **marks** состоит из следующих полей:

```

num_st      номер студента;
code        код предмета, который студент сдавал на экзамене;
mark        отметка, которую он получил.

```

Определим следующим образом поля таблицы **marks**:

Name-field	Type	Format	Label
Num_st	INTEGER	99999	номер ст.
Code	INTEGER	99	код курса
Mark	INTEGER	9	отметка

- с помощью кнопки Copy Field... описание поля num_st может быть скопировано из таблицы student, а описание поля code - из таблицы course

Индекс к таблице:

Index name	Primary	Unique	Field	Ascending	Word Index
Num_st	Yes	No	Num_st	Yes	No

Предусмотрим контроль ввода пользователем значения оценки (mark). Для этого войдем в свойства поля mark и нажмем кнопку Validation. В открывшемся окне Field Validation напишем выражение *mark > 1 AND mark < 6* и сообщение на случай ошибки: "Оценка должна быть в интервале от 2 до 5".

Пример 4.5.1

Программа, заполняющая таблицы **course** и **marks**.

```

/* create course */
REPEAT:
  CREATE course.
  course.code = NEXT-VALUE(s2).
  DISPLAY course.code.
  UPDATE course.name_c course.name_t.
END.
/* create marks */
FOR EACH student:
  FOR EACH course:

```

```

CREATE marks.
marks.num_st = student.num_st.
marks.code = course.code.
DISPLAY marks.num_st marks.code.
UPDATE marks.mark.
END.
END.

```

Таблицы MARKS и COURSE, MARKS и STUDENT связаны между собой отношением типа 1 (один к одному). Такая связь может быть реализована оператором FIND. Рассмотрим следующий пример:

Пример 4.5.2

Программа, выдающая имена студентов, названия сданных ими курсов и оценки.

```

FOR EACH marks:
  FIND student OF marks.
  FIND course OF marks.
  DISPLAY name_st name_c mark.
END.

```

При этом следует иметь ввиду, что использование конструкта OF в операторе FIND возможно, если общие поля в обеих таблицах имеют одни и те же имена и типы (num_st, code). Кроме того, первый параметр OF должен иметь уникальный индекс по смежному полю (в примере - индексы num_st, code).

Progress предоставляет удобное средство для проверки существования связанных записей в таблицах. Это функция CAN-FIND, которая, в частности, может использоваться в качестве VALEXP при определении соответствующих полей через DATA DICTIONARY. Например, для проверки наличия связи типа 1 (от MARKS к COURSE) при описании поля code в таблице MARKS можно определить VALEXP:

```
CAN-FIND(course OF marks)
```

Связь типа M (один ко многим) соединяет таблицы STUDENT и MARKS (COURSE и MARKS), так как одной записи таблицы STUDENT могут быть сопоставлены ноль, одна или несколько записей таблицы MARKS. Такая связь реализуется с помощью оператора FOR EACH:

Пример 4.5.3

Программа для каждого студента выдает его оценки.

```

FOR EACH student:
  FOR EACH marks OF student:
    DISPLAY name_st mark.
  END.
END.

```

Как и в случае оператора FIND, использование конструкта OF в FOR EACH возможно лишь тогда, когда смежное поле имеет одинаковое имя в обеих таблицах. Наличие индекса желательно, но не обязательно.

Для проверки наличия смежных записей и в этом случае может использоваться функция CAN-FIND с модифицированным аргументом:

```
CAN-FIND(FIRST marks OF student)
```

или

```
CAN-FIND(LAST marks OF student)
```

Операторы FOR EACH и FIND могут объединяться в одном операторе так, как это сделано в примере 4.5.4. При этом PROGRESS трактует ",EACH table-name" как "FOR EACH table-name", а ",table-name" - как "FIND table-name".

Пример 4.5.4

Программа, выдающая фамилии лекторов, которым студенты сдавали экзамены.

```

FOR EACH student, EACH marks OF student, course OF marks:
  DISPLAY student.name_st course.name_t.
END.

```

Объединение таблиц дает возможность сортировать записи по полям из различных таблиц.

Пример 4.5.5

Программа выдает оценки, которые поставил преподаватель.

```
FOR EACH course, EACH marks OF course BY name_t BY mark:
  DISPLAY name_t mark.
END.
```

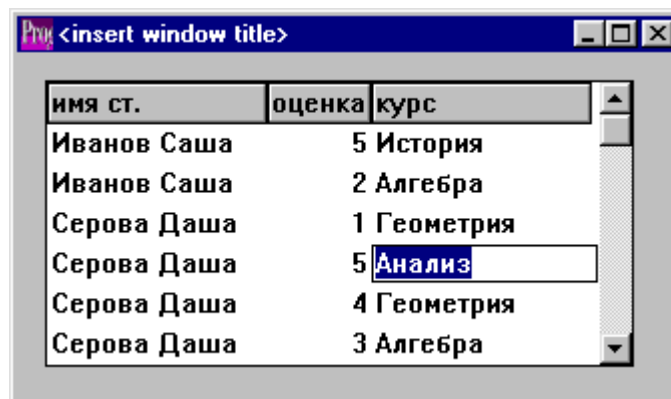
Следующий пример демонстрирует конструктор QUERY и соответствующий ему BROWSE, построенные по связным таблицам.

Пример 4.5.6

```
/****** DEFINE QUERIES *****/
DEFINE QUERY new-query FOR student, marks.
/****** DEFINE WIDGETS *****/
DEFINE BROWSE new-browse QUERY new-query
  DISPLAY name_st SPACE(3) mark WITH 12 DOWN.
DEFINE BUTTON btn-exit LABEL "Exit".
/****** DEFINE FRAMES *****/
DEFINE FRAME frame1
  new-browse AT ROW 1 COLUMN 2
  btn-exit AT ROW 1 COLUMN 50
  WITH NO-BOX CENTERED.
/****** MAIN LOGIC *****/
OPEN QUERY new-query FOR EACH student,
  EACH marks OF student.
ENABLE ALL WITH FRAME frame1.
WAIT-FOR CHOOSE OF btn-exit.
```

Пример 4.5.7 (AB)

Пример демонстрирует, как в AB строится редактируемый BROWSE по связным таблицам.



1. Откроем новое окно в AB.
2. Выберем в палитре объект типа BROWSE и свяжем его с полями name_st, mark и name_c из таблиц student, marks и course. Выберем опцию Enable All.
3. Обратим внимание на QUERY и на то, как меняется содержимое BROWSE при редактировании полей.
4. Запишем программу в файл под именем i04_0507.w. Запустим программу на исполнение.

Пример 4.5.8 (AB)

Этот пример демонстрирует как при удалении студента из таблицы student можно обеспечить одновременное удаление его оценок из таблицы marks.

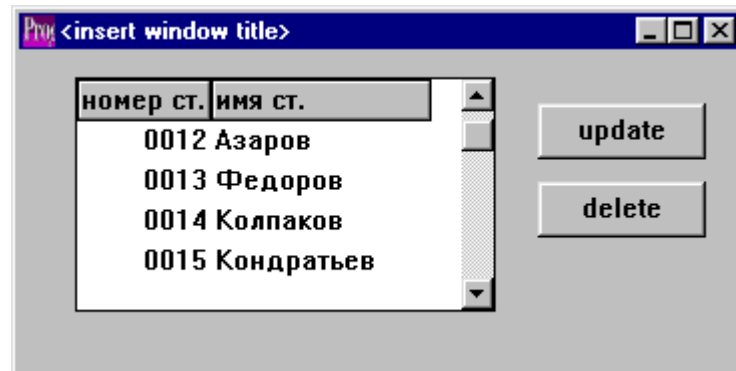
1. Откроем процедуру из файла i04_0402.w.
2. Выберем кнопку Delete и изменим триггер для нее на событие CHOOSE:


```
MESSAGE "Delete student?"
VIEW-AS ALERT-BOX QUESTION
BUTTONS yes-no UPDATE continue-ok AS LOGICAL.
```

```

IF continue-ok THEN DO:
  FOR EACH marks OF student:
    DELETE marks.
  END.
  GET CURRENT browse-1 SHARE-LOCK.
  DELETE student.
  browse-1:DELETE-SELECTED-ROWS().
END.

```

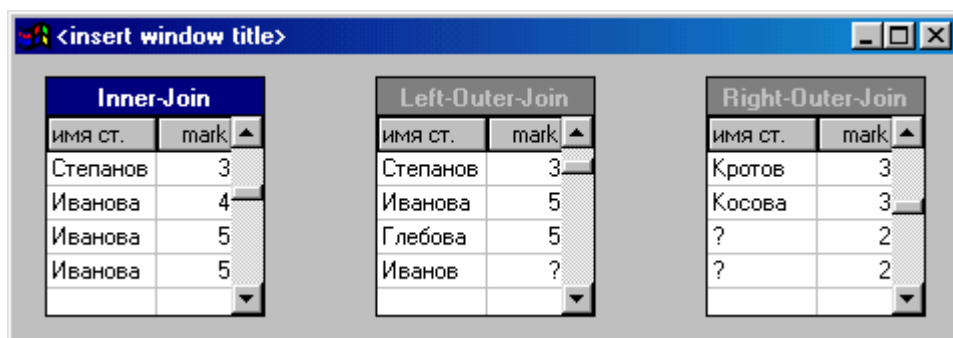


3. Запишем программу в файл под именем i04_0508.w. Запустим ее на исполнение.

В приведенных выше примерах демонстрировалось внутреннее соединение таблиц (INNER-JOIN). Начиная с Progress V8, для конструкта QUERY можно организовать и внешнее соединение таблиц (как левостороннее, так и правостороннее). Прежде, чем привести убедительный пример на эту тему, добавим по крайней мере одну новую запись в таблицу **student** (причем оценки для этого студента добавлять не будем) и несколько новых записей в таблицу **marks** (причем студенты с указанными номерами не должны присутствовать в таблице **student**).

Пример 4.5.9 (AB)

Приведенная ниже программа демонстрирует три возможных варианта соединения таблиц в конструкте QUERY : INNER-JOIN, LEFT OUTER-JOIN и RIGHT OUTER-JOIN.



1. Откроем новое окно в AB.



2. Выберем в палитре объект типа BROWSE () и свяжем его с полями name-st и mark из таблиц **student** и **marks**. Создадим заголовок: в окне свойств BROWSE-1 в поле Title напишем Inner-JOIN и установим Title Bar = yes.

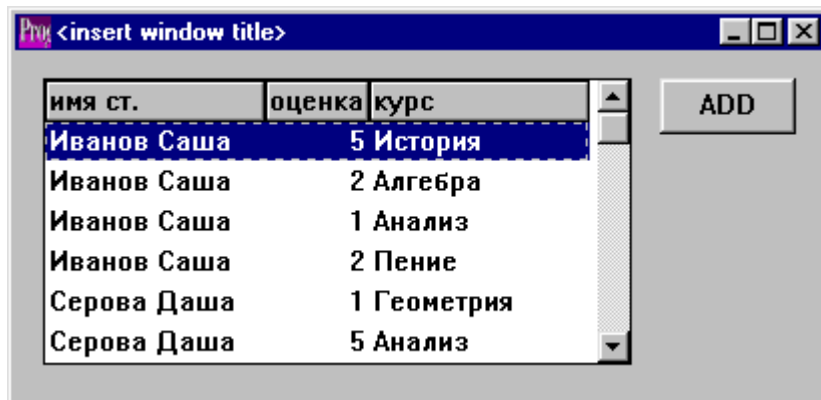
3. Выберем в палитре объект типа BROWSE и свяжем его с полями name-st и mark из таблиц **student** и **marks**. Здесь же, в окне Query Builder, укажем OUTER-JOIN для query: в верхней части окна выберем Options и в предложенной табличке двойным щелчком изменим в колонке Join значение INNER на OUTER. Создадим заголовок для browse: в окне свойств BROWSE-2 в поле Title напишем Left-Outer-Join и установим Title Bar = yes.

4. Выберем в палитре объект типа BROWSE и свяжем его с полями mark и name-st из таблиц **marks** и **student** (в отличие от двух первых browses здесь основной таблицей будет marks). Через Options, как и в предыдущем пункте, укажем для query OUTER-JOIN. Создадим заголовок: в окне свойств BROWSE-3 в поле Title напишем Right-Outer-Join и установим Title Bar = yes.

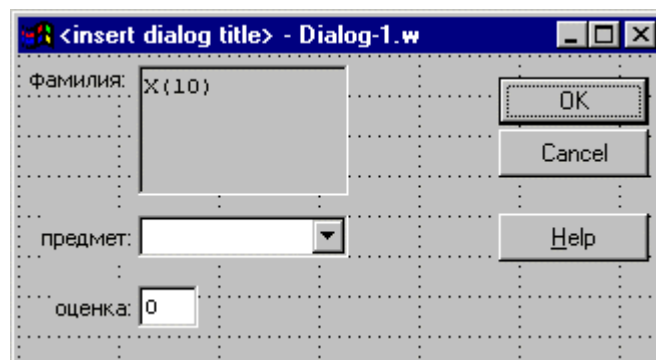
5. Запишем программу под именем i04_0509.w. и запустим ее на исполнение.

Пример 4.5.10 (AB)

Следующий пример демонстрирует еще один способ редактирования связанных таблиц.



1. Откроем процедуру из файла i04_0507.w.
2. В раздел Definition добавим следующие описания:
`DEFINE VARIABLE st_rowid AS ROWID.`
`DEFINE VARIABLE mr_rowid AS ROWID.`
3. Добавим кнопку Add и напишем триггер для нее:
`st_rowid = ROWID(student).`
`RUN C:\EXAMPLES\PART4\DIALOG-1.W(INPUT-OUTPUT st_rowid, OUTPUT mr_rowid).`
`OPEN QUERY browse-1 FOR EACH student,`
`EACH marks OF student, EACH course OF marks.`
`VIEW BROWSE-1.`
`REPOSITION browse-1 TO ROWID st_rowid, mr_rowid.`
4. Откроем новый контейнер типа DIALOG (через New) и разместим в нем:



- Browse по полю name_st таблицы student (в окне свойств откажемся от меток - поставим флажок no-labels);
- текстовое поле (слева от Browse) со значением "фамилия:";
- объект типа COMBO-BOX;
- текстовое поле (слева от COMBO-BOX) со значением "предмет:";
- поле mark из таблицы marks (изменим метку на "оценка").

В свойствах объекта COMBO-BOX отметим опцию List-Item-Pairs, а в поле Define as выберем тип INTEGER.

5. В раздел Definition добавим следующие описания:
`DEFINE INPUT-OUTPUT PARAMETER st_rowid AS ROWID.`
`DEFINE OUTPUT PARAMETER mr_rowid AS ROWID.`
6. В раздел Main Block перед оператором RUN ENABLE_UI добавим:
`COMBO-BOX-1:DELETE(1).`
`FOR EACH course:`
`COMBO-BOX-1:ADD-FIRST(course.name_c, course.code).`
`END.`
`COMBO-BOX-1 = INTEGER(COMBO-BOX-1:ENTRY(1)).`
`CREATE marks.`

7. В раздел Main Block после оператора RUN ENABLE_UI:
REPOSITION browse-2 TO ROWID st_rowid.
8. В свойствах фрейма уберем флажок Open the Query.
9. Напишем триггер на событие CHOOSE для кнопки ОК:
marks.num_st = student.num_st.
ASSIGN COMBO-BOX-1.
marks.code = COMBO-BOX-1.
ASSIGN marks.mark.
st_rowid = ROWID(student).
mr_rowid = ROWID(marks).
10. Сохраним диалоговое окно в файле dialog-1.w.
11. Запишем основную процедуру в файл под именем i04_0510.w. Запустим программу на исполнение.

4.6. Буферы записей

В Progress каждая процедура, работающая с записями из базы данных, имеет по одному буферу записи на каждую используемую таблицу. Это умалчиваемые буфера, которые не требуют специального описания. Дополнительные буфера могут быть определены следующим образом:

```
DEFINE BUFFER buffer-name FOR table-name.
```

Далее, в операторах обращения к таблице можно использовать buffer-name в качестве имени таблицы.

Следующий пример демонстрирует работу с одной таблицей через три буфера - умалчиваемый и два дополнительных.

Пример 4.6.1

Программа позволяет выбрать две записи о студентах и сравнить их характеристики.

```
/* DEFINE VARIABLES */
DEFINE VARIABLE mark_count AS INTEGER.
DEFINE VARIABLE mark_sum AS INTEGER.
DEFINE VARIABLE ave1 AS DECIMAL.
DEFINE VARIABLE ave2 AS DECIMAL.
/* DEFINE BUFFERS */
DEFINE BUFFER student1 FOR student.
DEFINE BUFFER student2 FOR student.
/* DEFINE QUERY */
DEFINE QUERY q-st FOR student.
/* DEFINE WIDGETS */
DEFINE BROWSE b-st QUERY q-st DISPLAY num_st name_st bdate WITH 5 DOWN.
DEFINE BUTTON but-1st LABEL "first student".
DEFINE BUTTON but-2st LABEL "second student".
DEFINE BUTTON but-old LABEL "who is older?".
DEFINE BUTTON but-well LABEL "who is better in learning?".
DEFINE BUTTON exit.
/* DEFINE FRAME */
DEFINE FRAME f-st
  b-st skip(1) but-1st but-2st but-old but-well exit
  skip(1) student1.name_st
  skip(1) student2.name_st WITH NO-BOX NO-LABEL.
/* DEFINE TRIGGERS */
ON CHOOSE OF but-1st DO:
  student1.name_st:BGCOLOR = 15.
  student2.name_st:BGCOLOR = 15.
  FIND student1 WHERE ROWID(student1) = ROWID(student).
  DISPLAY student1.name_st VIEW-AS TEXT WITH FRAME f-st.
END.
ON CHOOSE OF but-2st DO:
  student1.name_st:BGCOLOR = 15.
  student2.name_st:BGCOLOR = 15.
  FIND student2 WHERE ROWID(student2) = ROWID(student).
  DISPLAY student2.name_st VIEW-AS TEXT WITH FRAME f-st.
END.
```

```

ON CHOOSE OF but-old DO:
    IF student1.bdate < student2.bdate THEN student1.name_st:BGCOLOR = 3.
    ELSE student2.name_st:BGCOLOR = 3.
END.
ON CHOOSE OF but-well DO:
    ASSIGN mark_count = 0 mark_sum = 0.
    FOR EACH marks OF student1:
        mark_count = mark_count + 1.
        mark_sum = mark_sum + mark.
    END.
    ASSIGN ave1 = mark_sum / mark_count
    mark_count = 0 mark_sum = 0.
    FOR EACH marks OF student2:
        mark_count = mark_count + 1.
        mark_sum = mark_sum + mark.
    END.
    ave2 = mark_sum / mark_count.
    IF ave1 > ave2 THEN student1.name_st:BGCOLOR = 6.
    ELSE student2.name_st:BGCOLOR = 6.
END.
/* MAIN LOGIC */
OPEN QUERY q-st FOR EACH student.
ENABLE ALL WITH FRAME f-st.
WAIT-FOR CHOOSE OF exit.

```

В Progress каждая процедура, работающая с записью из базы данных, по умолчанию имеет свой буфер на каждую используемую таблицу. Для того, чтобы несколько процедур использовали один и тот же буфер, его следует объявить как SHARED (разделяемый).

Пример 4.6.2

```

DEFINE NEW SHARED BUFFER buf_student FOR student.
DEFINE NEW SHARED QUERY q_student FOR buf_student.
DEFINE BUTTON b_quit LABEL "Quit".
DEFINE BUTTON b_ascend LABEL "Name_st".
DEFINE BUTTON b_descend LABEL "Address".
DEFINE BUTTON b_num LABEL "Num_st".
DEFINE FRAME butt-frame b_ascend b_descend b_num b_quit WITH ROW 1.
ON CHOOSE OF b_ascend
DO:
    OPEN QUERY q_student FOR EACH buf_student USE-INDEX name_st.
    RUN c:\examples\part4\r-query.p.
END.
ON CHOOSE OF b_descend
DO:
    OPEN QUERY q_student FOR EACH buf_student BY buf_student.address.
    RUN c:\examples\part4\r-query.p.
END.
ON CHOOSE OF b_num
DO:
    OPEN QUERY q_student FOR EACH buf_student.
    RUN c:\examples\part4\r-query.p.
END.
ENABLE ALL WITH FRAME butt-frame.
WAIT-FOR CHOOSE OF b_quit.

```

Процедура r-query.p:

```

DEFINE SHARED BUFFER buf_student FOR student.
DEFINE SHARED QUERY q_student FOR buf_student.
DEFINE FRAME f_stud WITH CENTERED 15 DOWN ROW 3 USE-TEXT.
GET FIRST q_student.
DO WHILE AVAILABLE(buf_student):
    DISPLAY buf_student.name
        buf_student.num_st
        buf_student.address

```

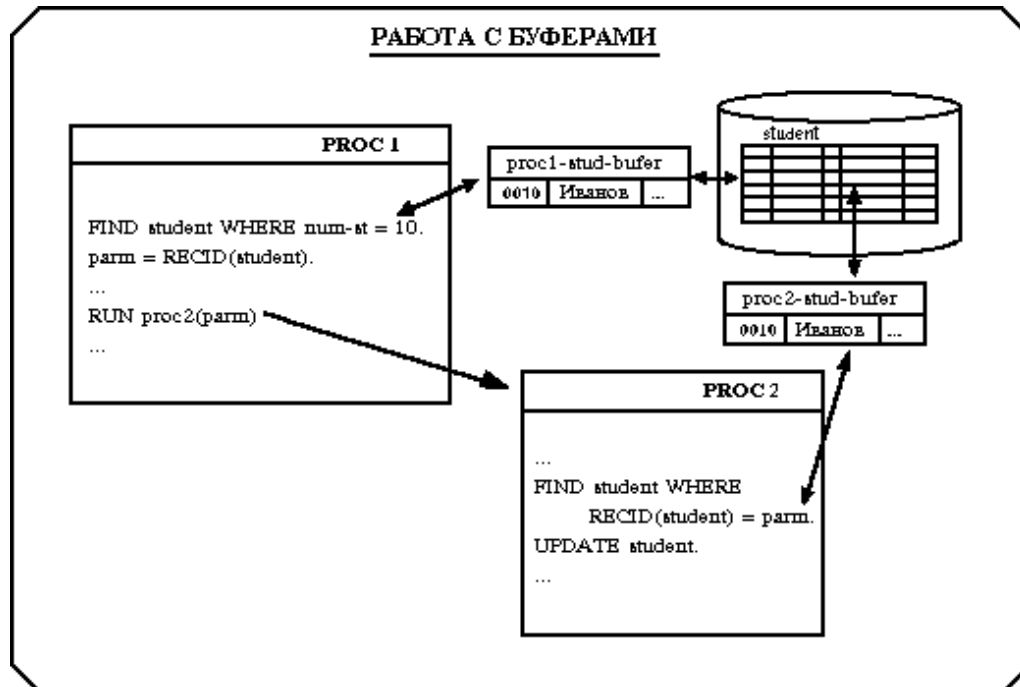


```

WITH FRAME f_stud CENTERED DOWN ROW 3 USE-TEXT.
DOWN 1 WITH FRAME f_stud.
GET NEXT q_student.
END.

```

В случае, когда две процедуры должны работать с одной и той же записью в разных буферах, следует позаботиться о том, чтобы вторая процедура прочитала в буфер ту же запись, с которой работала первая процедура.



Следующий пример демонстрирует один из возможных приемов работы в подобных случаях.

Пример 4.6.3

Программа вычисляет средний балл указанного студента и помечает тех студентов, чей средний балл меньше трех.

```

/* DEFINE VARIABLES */
DEFINE VARIABLE row_stud AS ROWID.
/* DEFINE QUERY */
DEFINE QUERY q-st FOR student.
/* DEFINE WIDGETS */
DEFINE BROWSE b-st QUERY q-st DISPLAY num_st name_st FORMAT "X(20)" WITH 5 DOWN.
DEFINE BUTTON exit.
DEFINE BUTTON aver LABEL "average".
/* DEFINE FRAME */
DEFINE FRAME f-st b-st aver exit WITH NO-BOX.
/* DEFINE TRIGGERS */
ON CHOOSE OF aver
DO:
    row_stud = ROWID(student).
    RUN c:\examples\part4\buf.p(row_stud).
    DISPLAY num_st name_st WITH BROWSE b-st.
END.
/* MAIN LOGIC */
OPEN QUERY q-st FOR EACH student.
ENABLE ALL WITH FRAME f-st.
WAIT-FOR CHOOSE OF exit.

```

Подпрограмма buf.p:

```

DEFINE INPUT PARAMETER row_stud AS ROWID.
DEFINE VARIABLE mark_count AS INTEGER.
DEFINE VARIABLE mark_sum AS INTEGER.
DEFINE VARIABLE str AS CHARACTER.
FIND student WHERE ROWID(student) = row_stud.

```

```

FOR EACH marks OF student:
    mark_count = mark_count + 1.
    mark_sum = mark_sum + mark.
END.
IF mark_sum / mark_count < 3 THEN DO:
    str = "Это неуспевающий студент (BAD)!".
    name_st = name_st + " BAD!".
    END.
MESSAGE "Средний балл студента - "
    + string(mark_sum / mark_count) SKIP str
VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.

```

Все упомянутые выше буфера были статическими. Между тем, начиная с версии 9 Progress, допускает, так называемые, динамические буфера, которые позволяют специфицировать таблицы во время исполнения приложения. Для этого в Progress предусмотрены объекты BUFFER и BUFFER-FIELD и ряд методов, позволяющих работать с этими объектами (последние применимы и к статическим буферам). Ниже приведен пример, демонстрирующий использование таких методов с динамическим буфером.

Пример 4.6.4

Программа позволяет уточнить названия полей любой таблицы базы данных.

```

DEFINE VARIABLE somebuf AS HANDLE.
DEFINE VARIABLE somefields AS HANDLE EXTENT 10.
DEFINE VARIABLE tab AS CHARACTER FORMAT "X(10)".
DEFINE FRAME someframe.
REPEAT:
SET tab LABEL "WHAT TABLE DO YOU WANT TO DISPLAY?".
CREATE BUFFER somebuf FOR TABLE tab.
    DEFINE VARIABLE i as INTEGER.
    REPEAT i = 1 TO somebuf:NUM-FIELDS:
        somefields[i] = somebuf:BUFFER-FIELD(i).
    DISPLAY somefields[i]:NAME
        WITH title somebuf:NAME CENTERED somebuf:NUM-FIELDS + 1 DOWN .
    END.
HIDE FRAME someframe.
END.

```

Пример 4.6.5

Приведенная ниже программа демонстрирует динамические объекты QUERY и BUFFER, которые используются для вывода данных из любой таблицы базы данных.

```

DEFINE VARIABLE somequery AS HANDLE.
DEFINE VARIABLE somebuf AS HANDLE.
DEFINE VARIABLE somefields AS HANDLE EXTENT 10.
DEFINE VARIABLE tab AS CHARACTER FORMAT "X(10)".
REPEAT:
SET tab LABEL "WHAT TABLE DO YOU WANT TO DISPLAY?".
CREATE QUERY somequery.
CREATE BUFFER somebuf FOR TABLE tab.
somequery:SET-BUFFERS(somebuf).
somequery:QUERY-PREPARE("for each " + tab).
somequery:QUERY-OPEN.
somequery:GET-NEXT.
DO WHILE NOT (somequery:QUERY-OFF-END):
    DEFINE VARIABLE i as INTEGER.
    REPEAT i = 1 TO somebuf:NUM-FIELDS:
        somefields[i] = somebuf:BUFFER-FIELD(i).
    DISPLAY somefields[i]:NAME somefields[i]:BUFFER-VALUE
        WITH title somebuf:NAME CENTERED somebuf:NUM-FIELDS + 1 DOWN .
    END.
    somequery:GET-NEXT.
END.
somequery:QUERY-CLOSE.
HIDE FRAME someframe.
END.

```

4.7. Триггеры базы данных

Триггер базы данных - это блок операторов 4GL, который выполняется, когда бы ни случилось упомянутое в нем событие. Например, при записи в базу происходит событие WRITE и выполняется триггер, предусмотренный для этого события (если такой имеется). В базе данных Progress предусмотрены следующие события, для которых могут быть написаны триггеры:

CREATE - генерируется при создании записи (например, операторами CREATE и INSERT). При этом Progress сначала создает запись, а затем исполняет триггер, предусмотренный для этого события и триггер *REPLICATION-CREATE* (если он был создан).

DELETE - генерируется при удалении записей (например, оператором DELETE). Progress сначала исполняет триггер, предусмотренный для этого события, потом триггер *REPLICATION-DELETE* (если он был создан) и лишь затем удаляет запись из базы.

FIND - генерируется при чтении записей из базы одним из возможных операторов выборки (например, FIND, GET, FOR EACH).

WRITE - генерируется при изменении содержимого записи. Сначала происходит изменение записи, затем Progress исполняет триггер WRITE, а потом - триггер *REPLICATION-WRITE* (если он был создан).

ASSIGN - единственное событие на уровне поля таблицы. Триггер исполняется сразу после того, как значение поля было изменено.

Триггеры в Progress могут использоваться для следующих целей:

- автоматической генерации значений
- обеспечения сложного протоколирования
- задания сложных правил целостности
- задания сложных правил доступа к данным
- обеспечения контроля над некоторыми событиями
- синхронной репликации таблиц

Ниже приведены примеры использования триггеров.

Пример 4.7.1

В следующем примере schema-триггер на событие CREATE для таблицы STUDENT используется для генерации значений поля num_st.

1. В DATA DICTIONARY создадим schema-триггер на событие CREATE для таблицы STUDENT:
 TRIGGER PROCEDURE FOR CREATE OF STUDENT.
 num_st = NEXT-VALUE(S1).

Запишем созданный триггер в файл с именем tr1.p.

2. Убедимся, что триггер работает с помощью следующей процедуры:
 REPEAT:
 INSERT STUDENT.
 END.

Пример 4.7.2

Протоколирование с помощью триггеров.

1. В Data Dictionary создадим вспомогательную таблицу AUDIT_TABLE со следующими полями: label_column (тип CHARACTER), old_value (тип CHARACTER), new_value (тип CHARACTER), user_id (тип CHARACTER), date (типа DATE).

2. Создадим следующий schema-триггер на событие ASSIGN для поля code таблицы course:
 TRIGGER PROCEDURE FOR ASSIGN OF course.code OLD VALUE old_code.
 DO:
 CREATE audit_table.
 ASSIGN label_column = "code"
 old_value = string(old_code)
 new_value = string(code)
 user_id = userid
 date = today.
 END.

Запишем созданный триггер в файл с именем tr2.p.

3. В AppBuilder создадим редактируемый браузер для таблицы COURSE.

4. Отредактируем в нескольких записях поле code.
5. В AppBuilder создадим браузер для таблицы AUDIT_TABLE.
6. Убедимся, что изменения запротоколированы в таблице AUDIT_TABLE.

Пример 4.7.3

Задание ссылочной целостности для таблиц course и marks. В программе описан session-триггер на событие ASSIGN для поля code таблицы COURSE.

```

/* DEFINE QUERY */
DEFINE QUERY q_cr FOR course.
DEFINE QUERY q_mr FOR marks.
/* DEFINE VARIABLE */
DEFINE VARIABLE oldcode AS INTEGER.
/* DEFINE WIDGETS */
DEFINE BROWSE b_cr QUERY q_cr DISPLAY course.code name_c name_t ENABLE ALL
    WITH 5 DOWN SEPARATORS.
DEFINE BROWSE b_mr QUERY q_mr DISPLAY num_st mark marks.code
    WITH 25 DOWN SEPARATORS.
DEFINE BUTTON exit .
/* DEFINE FRAMES */
DEFINE FRAME f_cr
    b_cr b_mr EXIT
    WITH CENTERED.
/* DEFINE TRIGGERS */
ON ASSIGN OF course.code OLD VALUE oldcode DO:
    FOR EACH marks WHERE marks.code = oldcode:
        marks.code = course.code.
    END.
    b_mr:REFRESH() IN FRAME f_cr.
END.
/* MAIN LOGIC */
OPEN QUERY q_cr FOR EACH course.
OPEN QUERY q_mr FOR EACH marks.
ENABLE ALL WITH FRAME f_cr.
WAIT-FOR CHOOSE OF exit.

```

Пример 4.7.4

Задание ссылочной целостности для таблиц marks и student..

1. Создадим в Data Dictionary следующий schema-триггер для таблицы student на событие DELETE:


```

            TRIGGER PROCEDURE FOR DELETE OF STUDENT.
            IF CAN-FIND(FIRST marks OF student)
            THEN RETURN ERROR.
            
```

Запишем созданный триггер в файл с именем tr3.p.

2. Добавим в таблицу student несколько новых студентов:

```

REPEAT:
    INSERT student.
END.

```

3. Разместим в окне AppBuilder объект BROWSE по таблице STUDENT и кнопку с меткой DELETE. Во встроенном редакторе напишем следующий триггер для кнопки на событие CHOOSE:

```

GET CURRENT browse-1 SHARE-LOCK.
DELETE student.
browse-1:REFRESH().

```

4. Убедимся, что теперь можно удалить новых студентов, но невозможно удалить студентов, у которых есть отметки.

5. Сохраним программу в файле под именем i04_0704.w.

Пример 4.7.5

Обеспечение контроля над некоторыми событиями (запрет на удаление записей из таблицы STUDENT в нерабочие дни и нерабочие часы).

- В Data Dictionary создадим следующий schema-триггер для таблицы STUDENT на событие DELETE: TRIGGER PROCEDURE FOR DELETE OF STUDENT.
DO:
 DEFINE VARIABLE hour AS INTEGER.
 DEFINE VARIABLE minute AS INTEGER.
 minute = TRUNC(TIME / 60, 0).
 hour = TRUNC(minute / 60, 0).
 IF WEEKDAY(TODAY) = 1 OR WEEKDAY(TODAY) = 7 OR hour < 8 OR hour > 18
 THEN RETURN ERROR.
END.

Запишем созданный триггер в файл с именем tr4.p.

- Запустим программу из примера i04_0704.w (сделаем это в подходящее время или изменим нерабочие часы в триггере) и убедимся, что триггер работает.

4.8 Рабочие и временные таблицы

Рабочие и временные таблицы представляют собой вспомогательные структуры для хранения данных, подобные таблицам базы данных, но создаваемые только на время выполнения приложения.

В следующей таблице приводятся сравнительные характеристики таблиц базы данных, рабочих и временных таблиц:

	Таблицы базы данных	Рабочие таблицы	Временные таблицы
Размещение	в базе данных на диске	в оперативной памяти	во временной базе данных на диске
Индексы	строятся	нет	строятся
Удаление записей	оператором DELETE	оператором DELETE или по концу процедуры	оператором DELETE или по концу процедуры
Перемещение записей	используются буферы	без буферизации	используются буферы
Хранение триггеров и validation	да	нет	нет
Поддержка механизма транзакций	автоматическая поддержка механизма транзакций	нет	поддержка локальных транзакций
Многопользовательский режим работы	да	нет	нет
Поддержка SQL-запросов	да	нет	нет

Рабочие таблицы

Используются для создания сложных отчетов (с двумя и более проходами по таблице), для сложных сортировок и т.д. Рабочая таблица должна быть определена в использующей ее процедуре. При необходимости работать с ней более широко в рамках приложения, следует определить ее как SHARED. Так как рабочая таблица не может иметь индексов, вместо оператора FIND используют FIND FIRST. При описании структуры рабочей таблицы можно использовать конструктор LIKE для копирования структуры полей базы данных, или определять новые поля аналогично тому, как описываются переменные в операторе DEFINE.

Синтаксис для описания рабочих таблиц:

```
DEFINE [[NEW] SHARED] {WORK-TABLE} work-table-name [LIKE table-name]
[FIELD field-name {{AS type}}{LIKE field}} [field-options]...
```

Пример 4.8.1

Вычисление среднего балла каждого студента.

```
DEFINE WORK-TABLE w-table
  FIELD w-num LIKE marks.num_st
  FIELD w-mark AS INTEGER FORMAT "999"
  FIELD w-count AS INTEGER FORMAT "99".
FOR EACH marks:
  FIND FIRST w-table WHERE w-table.w-num = marks.num_st NO-ERROR.
  IF NOT AVAILABLE w-table THEN DO:
    CREATE w-table.
    w-table.w-num = marks.num_st.
  END.
```

```

w-table.w-mark = w-table.w-mark + marks.mark.
w-table.w-count = w-table.w-count + 1.
END.
FOR EACH w-table BY w-mark / w-count:
    DISPLAY w-num w-mark / w-count.
END.

```

Временные таблицы

Временные таблицы хранятся на диске в рабочей директории (или в директории, указанной стартовым параметром -T) и используются только одним приложением. Определяется временная таблица в той процедуре, в которой будет использоваться.

Синтаксис для описания:

```

DEFINE [[NEW] SHARED] TEMP-TABLE temp-table-name
    [LIKE table-name [USE-INDEX index-name [AS PRIMARY]]...]
    [FIELD field-name {{AS type}}{LIKE field}} [field-options]]...
    [INDEX index-name [IS [UNIQUE] [PRIMARY]]
    {index-field [ASCENDING|DESCENDING]}...]...

```

Если временная таблица копирует структуру таблицы базы данных, она принимает и ее индексы. Первичным индексом временной таблицы будет или первичный индекс копируемой таблицы, или перекрывающий его PRIMARY-индекс, указанный в описании. В остальных случаях будет использоваться физический порядок ввода записей. WORD-индексы во временных таблицах не разрешены.

Пример 4.8.2 (AB)

Вычисление среднего балла студентов.

1. Откроем новое окно.

2. Войдем во встроенный редактор и в область Definition скопируем через Clipboard описание рабочей таблицы w-table из предыдущего примера и заменим описатель WORK-TABLE на TEMP-TABLE:

```

DEFINE TEMP-TABLE w-table
    FIELD w-num LIKE marks.num_st
    FIELD w-mark AS INTEGER FORMAT "999"
    FIELD w-count AS INTEGER FORMAT "99".

```

3. Перейдем в раздел Main Block и перед "RUN enable_UI." вставим операторы заполнения таблицы из предыдущего примера:

```

FOR EACH marks:
    FIND FIRST w-table WHERE w-table.w-num = marks.num_st NO-ERROR.
    IF NOT AVAILABLE w-table THEN DO:
        CREATE w-table.
        w-table.w-num = marks.num_st.
    END.
    w-table.w-mark = w-table.w-mark + marks.mark.
    w-table.w-count = w-table.w-count + 1.
END.

```

4. Возьмем с палитры и вставим в окно browse. В Query Builder не станем выбирать никакие таблиц базы данных Univ, а щелкнем по кнопке "Freeform Query". После подтверждения наших намерений создать действительно query свободной формы, в окне появится browse несколько непривычного вида - без полей, со вспомогательным текстом.

5. При активном browse войдем во встроенный редактор. В списке стандартных событий для Freeform Browse помимо VALUE-CHANGE предлагаются

```

OPEN-QUERY
DISPLAY

```

В "триггер" для OPEN-QUERY вставим имя нашей временной таблицы:

```

OPEN QUERY {&SELF-NAME} FOR EACH w-table.

```

В "триггер" для DISPLAY вставим поля из нашей временной таблицы:

```

w-num w-mark w-count

```

6. Запишем программу в файл под именем i04_0802.w. Запустим ее на исполнение.

5. ОБЕСПЕЧЕНИЕ ЦЕЛОСТНОСТИ БАЗ ДАННЫХ

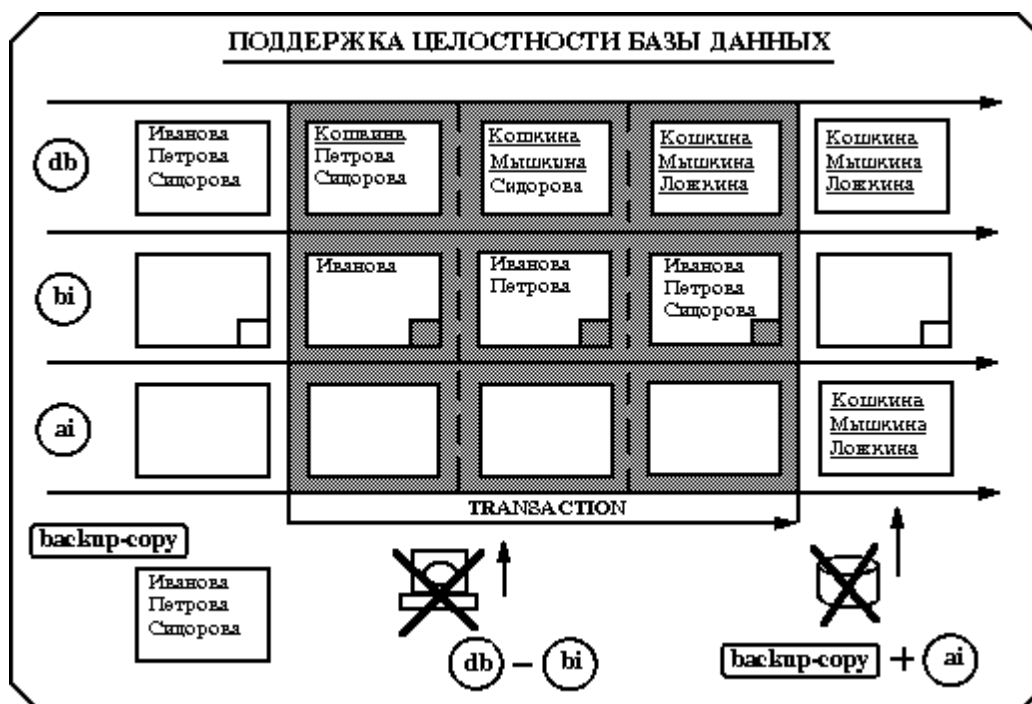
5.1. Механизмы поддержки целостности баз данных

В Progress существуют два механизма поддержки физической целостности данных - *roll-back recovery* и *roll-forward recovery* (before-imaging и after-imaging).

Before-imaging обеспечивает оперативное восстановление данных при внезапном прерывании процесса обработки в результате системных и машинных ошибок. Этот механизм позволяет после перезапуска сеанса работы приводить базу данных в целостное состояние без вмешательства программиста. Поддерживается before-imaging автоматически.

After-imaging запускается и сопровождается администратором на случай возможного физического разрушения базы данных. After-imaging-поддержка позволяет восстанавливать базу данных с точностью до последнего успешно выполненного блока изменений.

Работа этих механизмов вплотную связана с понятием транзакции в Progress.



5.2. Транзакции

Транзакция - это некоторый блок действий по изменению базы данных, до начала которого база находится в одном логически целостном состоянии, и по успешному завершению которого она переходит в другое логически целостное состояние. Такой блок действий или выполняется от начала до конца, или - если возникло специальное Progress-прерывание - отменяется, происходит "откат" - возвращение базы данных к состоянию на момент начала выполнения транзакции.

Под специальными Progress-прерываниями подразумеваются в первую очередь серьезные машинные и системные ошибки, которые чаще всего приводят к аварийному завершению сеанса работы с базой данных.

В каждый момент выполнения программы может существовать только одна активная транзакция.

Если нет активной транзакции, то транзакция стартует:

- в procedure-блоке,
- в trigger-блоке,
- в каждой итерации блоков FOR EACH, REPEAT, DO ON ERROR при наличии в них:
 - операторов корректировки базы данных
 - операторов чтения с опцией EXCLUSIVE-LOCK.
- в блоках DO, FOR EACH и REPEAT при наличии в их заголовках ключевого слова TRANSACTION.

В приведенных ниже примерах 5.2.1 - 5.2.2 мы рассмотрим работу механизма транзакций по умолчанию.

Пример 5.2.1

```

Tr [ REPEAT:
    CREATE student.
    num_st = NEXT-VALUE(s1).
    UPDATE student.
    END.

```

Здесь 2 транзакционных блока:

1. PROCEDURE - не запускает транзакцию, так как не содержит операторов корректировки, которые не включены уже в какой-либо внутренний транзакционный блок.
2. REPEAT - запускает транзакцию в каждой итерации, так как содержит операторы корректировки.

Если мы прервем выполнение процедуры, смоделировав Progress-прерывание (отключим компьютер или нажмем клавишу STOP) во время выполнения оператора UPDATE (пустая запись уже добавлена, а транзакция еще не закончилась), последняя введенная запись будет отсутствовать в базе данных. Убедимся в этом:

```

FOR EACH student:
    DISPLAY student.
END.

```

Пример 5.2.2

```

Tr [ INSERT student WITH 2 COLUMNS.
    REPEAT:
        INSERT marks.
        PAUSE.
    END.

```

← *Stop*

Здесь 2 транзакционных блока:

1. PROCEDURE - запускает транзакцию, так как содержит оператор корректировки базы данных;
2. REPEAT - не запускает транзакцию, так как уже есть активная транзакция.

Если мы прервем выполнение процедуры, смоделировав Progress прерывание во время паузы, в базе данных будут отсутствовать не только все добавленные оценки, но и добавленный студент.

Если нас не устраивают стандартные размеры транзакции, мы можем сами определить транзакционные блоки с помощью конструктора TRANSACTION:

Пример 5.2.3

Увеличение размера транзакции.

- 1) стандартная транзакция:

```

Tr [ REPEAT:
    INSERT student WITH 2 COLUMNS.
    REPEAT:
        INSERT marks.
    END.
    END.

```

- 2) увеличение транзакции:

```

Tr [ DO TRANSACTION:
    REPEAT:
        INSERT student WITH 2 COLUMNS.
        REPEAT:
            INSERT marks.
        END.
    END.
    END.

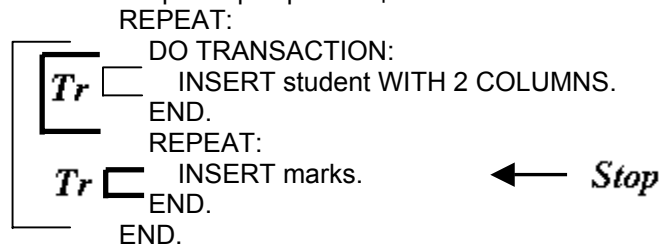
```

← *Stop*

Транзакцией будет не одна итерация, а весь REPEAT-блок. Если прервать выполнение процедуры (STOP) во время добавления оценок в таблицу marks, исчезнут все изменения базы данных, произведенные в программе.

Пример 5.2.4

Уменьшение размера транзакции.



Если прервать выполнение процедуры (STOP) во время добавления оценок в таблицу marks, то исчезнет последняя оценка, а последняя запись о студенте сохранится.

- Логическая функция TRANSACTION позволяет выявить наличие транзакции в данный момент.

Progress предлагает и более гибкий механизм откатов. Этот механизм касается обработки менее серьезных ошибок, при которых не происходит аварийное завершение программы, но целостность данных в базе тем не менее может быть нарушена.

Progress-прерывания, которые нарушают выполнение транзакции, можно разделить на два типа, и размер отката зависит от того, какого типа прерывание произошло.

1 тип:

- прерывание по машинной ошибке (например, ошибка питания);
- прерывание STOP:
 - любая системная ошибка (например, не найден файл на диске);
 - генерация прерывания оператором STOP или нажатием клавиши STOP;

2 тип:

- прерывание ERROR:
 - повторный ввод уникального индекса;
 - поиск несуществующей записи с помощью операторов FIND, FIND FIRST, FIND LAST;
 - генерация прерывания оператором RETURN ERROR в триггерном блоке или вызываемой процедуре;
 - генерация прерывания нажатием клавиши, определенной в программе как ERROR;
- прерывание ENDKEY:
 - конец ввода данных из файла;
 - поиск несуществующей записи с помощью операторов FIND NEXT, FIND PREV;
 - генерация прерывания нажатием клавиши, определенной в программе как ENDKEY.

До сих пор мы говорили только о прерываниях первого типа. Со вторым типом прерываний вплотную связано понятие субтранзакции.

Субтранзакция - это совокупность действий по изменению базы данных и/или переменных внутри блоков при активной транзакции. Механизм субтранзакций позволяет при не слишком серьезных ошибках (прерываниях второго типа) осуществлять более мелкие откаты.

Субтранзакция стартует тогда, когда есть активная транзакция, а Progress встретил субтранзакционный блок.

Субтранзакционными блоками являются:

- каждая итерация блоков FOR EACH, REPEAT, DO TRANSACTION, DO ON ERROR, DO ON ENDKEY;
- trigger-блок;
- procedure-блок, который запускается оператором RUN из транзакционного блока.

В отличие от транзакции, которая всегда только одна, субтранзакций может быть одновременно сколько угодно.

Прерывания первого типа приводят к откату всей активной транзакции (независимо от того, завершились или нет внутренние субтранзакции), а прерывания второго типа приводят к откату ближайшей субтранзакции.

Использование одного из прерываний второго типа - прерывания ERROR уже демонстрировалось в примерах 4.7.4 и 4.7.5. Напомним, что в этих примерах демонстрировались schema-триггеры, в которых использовался оператор RETURN ERROR. Генерация этого прерывания приводила к откату ближайшей субтранзакции и, в частности, отмене оператора DELETE, вызвавшего исполнение

schema-триггера. В следующих примерах обсуждается реакция транзакций и субтранзакций на прерывания первого и второго типа.

Пример 5.2.5

```

Tr [ FOR EACH course:
    UPDATE course.
    END.
    REPEAT:
        INSERT student.
        REPEAT:
            STTr [ INSERT marks.
                PAUSE.
                END.
            END.
        END.
    END.

```

← *Stop / Endkey*

Здесь 4 транзакционных блока:

1. PROCEDURE - не стартуется транзакция, так как не содержит операторов корректировки, которые не входили бы уже в какой-либо внутренний транзакционный блок;
2. FOR EACH - стартуется транзакция в каждой итерации, так как содержит оператор UPDATE;
3. внешний REPEAT - стартуется транзакция в каждой итерации, так как содержит оператор INSERT (а предыдущая транзакция уже завершилась);
4. внутренний REPEAT - стартуется субтранзакция в каждой итерации.

Если мы прервем выполнение процедуры, сгенерировав прерывание первого типа (нажмем клавишу STOP или отключим компьютер) во время выполнения субтранзакции, в базе данных будут отсутствовать не только добавленные оценки, но и последний добавленный студент, так как прерывание возникло во время выполнения транзакции во внешнем REPEAT-блоке. Все изменения в таблице course сохранятся, так как транзакция в блоке FOR EACH завершилась раньше.

Если мы прервем выполнение процедуры во время паузы, сгенерировав прерывание второго типа ENDKEY (нажмем клавишу Esc), исчезнет только последняя отметка - откатится субтранзакция.

Пример 5.2.6

Убедимся в том, что нажатие на клавишу STOP в субтранзакции - в подпрограмме *tr.p* - приведет к откату транзакции: исчезнет запись о последнем добавленном студенте и записи о его отметках. Прерывание второго типа в субтранзакции (например, неудачное выполнение FIND) вызовет откат субтранзакции.

```

Tr [ REPEAT:
    INSERT student
    RUN tr.p.
    END.
/* процедура tr.p: */
REPEAT:
    INSERT marks.
    FIND course WHERE course.code = marks.code.
    DISPLAY course.name_c.
    END.

```

Пример 5.2.7.

В приведенной ниже программе два trigger-блока, каждый из которых стартуется транзакцию. Добавим нового студента. Прервав выполнение программы нажатием клавиши STOP во время добавления нового курса, убедимся, что запись о студенте сохранилась - откат не затронет изменения в первом триггерном блоке. Если вместо клавиши STOP нажать END-ERROR, результат будет тем же, так как в данном случае нет субтранзакции, и любой откат затронет всю текущую транзакцию:

```

DEFINE BUTTON ins_st.
DEFINE BUTTON ins_cr.
DEFINE BUTTON exit_b.
FORM SKIP(1) ins_st SKIP(1) ins_cr SKIP(1) exit_b WITH FRAME a.
ON CHOOSE OF ins_st DO:
Tr [ INSERT student WITH FRAME st VIEW-AS DIALOG-BOX.
    END.
ON CHOOSE OF ins_cr DO:
Tr [ INSERT course WITH FRAME cr VIEW-AS DIALOG-BOX.

```

```
END.
ENABLE ALL WITH FRAME a.
WAIT-FOR CHOOSE OF exit_b.
```

Переменные, переопределенные во время выполнения транзакции (или субтранзакции), при откате из нее восстанавливают свое старое значение. Убедимся в этом на следующем примере:

Пример 5.2.8

```
DEFINE VARIABLE ctr AS INTEGER /*NO-UNDO*/ INITIAL 0.
REPEAT:
  CREATE course.
  ctr = ctr + 1.
  DISPLAY ctr.
  UPDATE course WITH NO-BOX.
END.
DISPLAY "Создано " ctr " новых курсов" WITH NO-BOX NO-LABELS.
```

После выхода из транзакции по нажатию клавиши END-ERROR значение переменной ctr окажется на единицу меньше, чем во время работы транзакции (для прерывания работы здесь используем не STOP, а END-ERROR, так как иначе мы не увидим значения переменной после отката).

Управлять откатами можно с помощью конструкта UNDO.

```
UNDO[метка1] [ , RETRY [метка1]
              , LEAVE [метка2]
              , NEXT [метка2]
              , RETURN [ ERROR | NO-APPLY ] ]
```

Использование его вызовет откат из субтранзакции в текущем или помеченном блоке, а далее возможны:

- повторение итерации текущего или помеченного блока (RETRY),
- переход к следующей итерации текущего или помеченного блока (NEXT),
- выход из текущего или помеченного блока (LEAVE),
- завершение процедуры или триггерного блока (RETURN).

1) Назначение отката, когда стандартные откаты не предусмотрены.

Пример 5.2.9

Добавляем запись о новом курсе и создаем записи об отметках студентов по этому курсу с проверкой существования студента.

```
REPEAT:
  INSERT course.
  REPEAT:
    CREATE marks.
    marks.code = course.code.
    UPDATE num_st mark.
    IF NOT CAN-FIND(student OF marks)
    THEN DO:
      MESSAGE "Такого студента не существует"
      VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
      UNDO.
    END.
  END.
END.
```

Здесь во внутреннем REPEAT-блоке стартует субтранзакция. Если во время ввода очередной оценки мы укажем несуществующего студента, то оператор UNDO произведет "откат" из субтранзакции, то есть отменит все изменения базы данных, произведенные в последней итерации внутреннего REPEAT-блока.

2) Изменение отката по прерыванию ERROR.

Стандартные действия, приписываемые блоку при прерывании ERROR:

```
ON ERROR UNDO, RETRY
```

(отмена последних изменений и повторение итерации ближайшего включающего блока).

Пример 5.2.10

Корректируем записи об отметках с просмотром фамилий соответствующих студентов.

```
FOR EACH marks:
  UPDATE marks.
  FIND student WHERE student.num_st = marks.num_st.
  DISPLAY name_st.
END.
```

Если студент не найден, произойдет откат и повторение итерации.

Обработка прерывания ERROR может быть назначена явно:

```
FOR EACH marks ON ERROR UNDO, LEAVE:
  UPDATE marks.
  FIND student WHERE student.num_st = marks.num_st.
  DISPLAY name_st.
END.
```

Если студент не найден, произойдет откат и выход из блока.

3) Изменение отката по прерыванию ENDKEY.

Стандартные действия, приписываемые блоку при прерывании ENDKEY:

```
ON ENDKEY UNDO, LEAVE
```

(отмена последних изменений и выход из ближайшего включающего блока).

Пример 5.2.11

```
FOR EACH course:
  UPDATE course.
  REPEAT:
    FIND NEXT marks OF course.
    DISPLAY marks.mark.
  END.
END.
```

После редактирования записи о предмете начинается просмотр оценок по этому предмету. Когда список оценок исчерпан, происходит переход к редактированию следующего предмета. Если мы назначим свою обработку прерывания по ENDKEY, результат существенно изменится - после просмотра последней оценки мы выйдем из программы, отменив изменения в записи о последнем предмете:

```
lab: FOR EACH course:
  UPDATE course.
  REPEAT ON ENDKEY UNDO lab, LEAVE lab:
    FIND NEXT marks OF course.
    DISPLAY marks.mark.
  END.
END.
```

4) Изменение отката по прерыванию STOP.

Стандартными действиями Progress при возникновении прерывания STOP будет откат транзакции и возврат в редактор или startup процедуру:

```
ON STOP UNDO, RETURN
```

Однако можно назначить и свою обработку. В следующем примере предусмотрена обработка случайного нажатия на клавишу STOP:

Пример 5.2.12

```
FOR EACH student ON STOP UNDO, RETRY:
  IF RETRY THEN DO:
    MESSAGE " STOP? " VIEW-AS ALERT-BOX QUESTION
    BUTTONS yes-no UPDATE stop_ok AS LOGICAL.
    IF stop_ok THEN QUIT.
  END.
  UPDATE student.
END.
```

5) Работа с клавишей END-ERROR.

Функциональная клавиша END-ERROR, которую мы обычно используем для выхода из REPEAT-цикла, в разных ситуациях (в зависимости от контекста) может работать по-разному:

как ENDKEY - UNDO, LEAVE	"выполнить откат и покинуть блок", когда исполняется первый диалоговый оператор в итерации;
как ERROR - UNDO, RETRY	"выполнить откат и повторить итерацию" при исполнении последующих операторов.

Пример 5.2.13

```
REPEAT:
  PROMPT-FOR student.num_st.
  FIND student USING num_st.
  UPDATE address phone.
END.
```

Нажав END-ERROR во время выполнения первого диалогового оператора внутри итерации (PROMPT-FOR), мы вернемся в редактор. Здесь END-ERROR работает как ENDKEY. Нажатие END-ERROR во время выполнения следующего диалогового оператора (UPDATE) приведет к приглашению заново ввести имя студента, то есть END-ERROR сработает как ERROR.

Механизм работы транзакций и субтранзакций в Progress заключается в использовании журналов транзакций и субтранзакций before-image и local-before-image. Первый содержит информацию об изменении базы данных во время работы транзакции. Для каждой базы данных постоянно существует один такой журнал (основной его файл имеет расширение bi). Журналы local-before-image используются для работы с субтранзакциями и создаются для каждого пользователя на текущий сеанс Progress.

5.3. Локализация данных в многопользовательских задачах

При работе нескольких пользователей с одной базой данных могут возникать конфликтные ситуации из-за одновременного обращения к данным. В Progress такие проблемы разрешаются с помощью механизма автоматической локализации данных на уровне записей.

Несколько пользователей могут одновременно читать одну и ту же запись. Два пользователя не могут одновременно изменять одну и ту же запись.

При работе с записями Progress использует два вида замков:

```
SHARE-LOCK
EXCLUSIVE-LOCK
```

Записи, которую программа читает с помощью операторов FOR EACH и FIND приписывается по умолчанию замок SHARE-LOCK (при этом другие пользователи не могут изменить эту запись, но могут читать).

Сфера действия SHARE-LOCK - в пределах области доступа (scope) - пока запись находится в буфере.

В следующей программе областью доступа очередной записи будет текущая итерация:

FOR EACH student:	← устанавливается SHARE-LOCK
DISPLAY student.	
END.	← снимается SHARE-LOCK

Записи, которую программа изменяет, приписывается режим EXCLUSIVE-LOCK (другие не могут даже читать).

Сфера действия EXCLUSIVE-LOCK:

- включается в момент обновления записи в буфере или во время чтения записи с опцией EXCLUSIVE-LOCK;

- снимается по концу транзакции (если транзакция завершилась, а работа с записью продолжается, EXCLUSIVE-LOCK будет заменен на SHARE-LOCK).

Рассмотрим следующие процедуры:

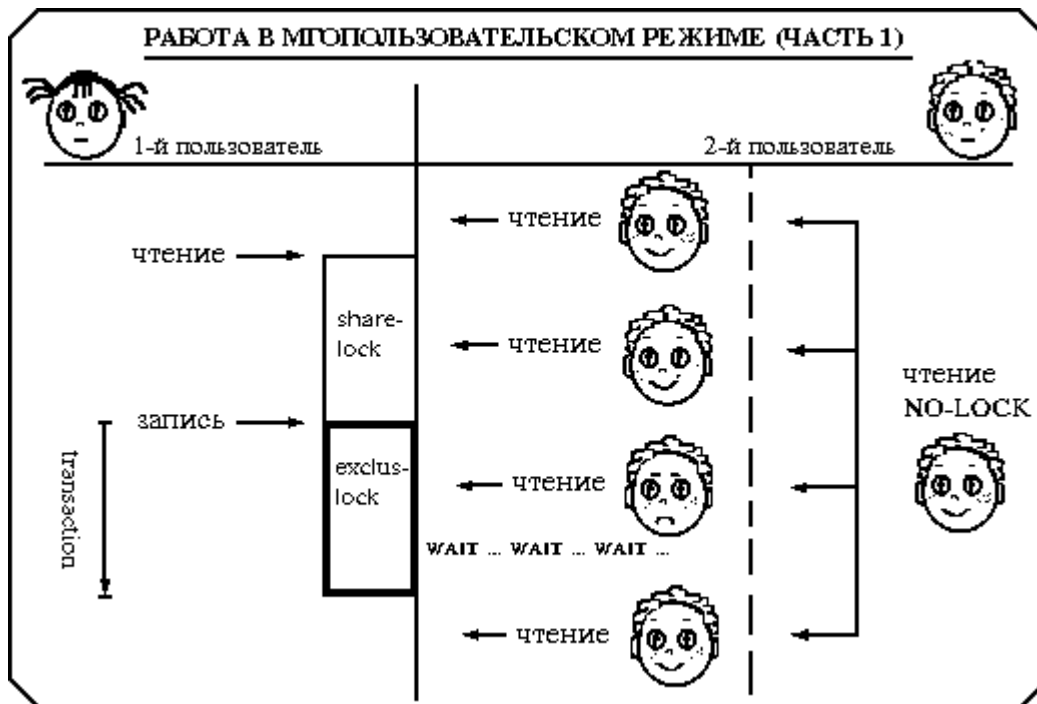
- 1)

FOR EACH student:	← устанавливается SHARE-LOCK
UPDATE student.	← замок заменяется на EXCLUSIVE-LOCK
END.	← снимается замок
- 2)

DO TRANSACTION:	
FOR EACH student:	← устанавливается SHARE-LOCK на очередную запись
UPDATE student.	← замок очередной записи заменяется на EXCLUSIVE-LOCK
END.	
END.	← все записи таблицы имеют замок EXCLUSIVE-LOCK
	← снимаются все замки
- 3)

DO TRANSACTION:	
FIND FIRST student:	← student - SHARE-LOCK
UPDATE student.	← student - EXCLUSIVE-LOCK
END.	← student - SHARE-LOCK
FOR EACH marks of student:	← marks - SHARE-LOCK
UPDATE marks.	← marks - EXCLUSIVE-LOCK
END.	← marks - снимается EXCLUSIVE -LOCK
	← student - снимается SHARE-LOCK

В следующем примере два пользователя будут одновременно работать с одной и той же записью. Для выполнения примера следует стартовать сервер на базу данных Univ, предварительно закрыв однопользовательский доступ к базе из текущего сеанса (disconnect), и два сеанса разработчика, установив в каждом из них connect к базе данных в многопользовательском режиме.



Запуск такой конфигурации в рабочем режиме, когда сервер и клиенты связываются по сетевым протоколам, очень коротко описан в приложении 1 (рекомендуем обратиться к администратору или к документации по администрированию). В учебном режиме, при наличии одного продукта Progress - ProVision, можно эмулировать многопользовательский режим работы с базой данных, стартовав на одном компьютере "мини-сервер" _mprosrv, входящий в состав ProVision, и два стандартных сеанса разработчика.

При выполнении примера первому пользователю рекомендуется приостанавливать работу после вывода на экран информации сначала о SHARE-LOCK, а затем о EXCLUSIVE-LOCK выбранной им записи. Второй пользователь должен в обоих случаях запустить на выполнение свою процедуру заново.

Пример 5.3.1.Программа первого пользователя:

```

DEFINE VARIABLE ans AS LOGICAL.
PROMPT-FOR student.num_st with frame a.
FIND student USING num_st.
DISPLAY "Student" name_st "is SHARE-LOCK now." with frame b no-labels.
DISPLAY name_st with frame c.
UPDATE phone with frame c.
IF phone ENTERED THEN DO:
    DISPLAY "Student" name_st "is EXCLUSIVE-LOCK now." with frame d no-labels.
    SET ans LABEL "UPDATE OK? (yes/no)".
    IF NOT ans THEN UNDO, NEXT.
END.

```

Программа второго пользователя:

```

FOR EACH student:
    DISPLAY name_st phone.
END.

```

Добавив в оператор чтения опцию NO-LOCK, второй пользователь получит возможность читать записи базы данных независимо от того, какие замки им приписаны:

```

FOR EACH student NO-LOCK:
    DISPLAY name_st phone.
END.

```

Таким образом, существует три режима работы программы с записью:

- с замком EXCLUSIVE-LOCK - режим EXCLUSIVE-LOCK;
- с замком SHARE-LOCK - режим SHARE-LOCK;
- без учета замков - режим NO-LOCK.

Согласование режимов локализации при работе нескольких пользователей с одной и той же записью:

если на записи:	для других возможны:
NO-LOCK (нет замков)	EXCLUSIVE-LOCK SHARE-LOCK NO-LOCK
SHARE-LOCK	SHARE-LOCK NO-LOCK
EXCLUSIVE-LOCK	NO-LOCK

Во всех остальных случаях пользователи получают соответствующие сообщения.

При работе с записью в режиме NO-LOCK следует иметь в виду, что если она будет вовлечена в транзакцию (получит замок EXCLUSIVE-LOCK), то по завершению транзакции (если не произойдет освобождения записи) EXCLUSIVE-LOCK сменится на SHARE-LOCK.

Обычная проблема при многопользовательской работе с базами данных - ***Deadly Embrace*** - "***смертельное объятие***". Каждая СУБД решает эту проблему по-своему.

Рассмотрим ситуацию, когда два пользователя собираются редактировать одну и ту же запись, и оба уже работают с ней в режиме SHARE-LOCK.

Пример 5.3.2.Программа первого пользователя:

```

FIND FIRST student.
UPDATE student.

```

```

← SHARE-LOCK
← EXCLUSIVE-LOCK

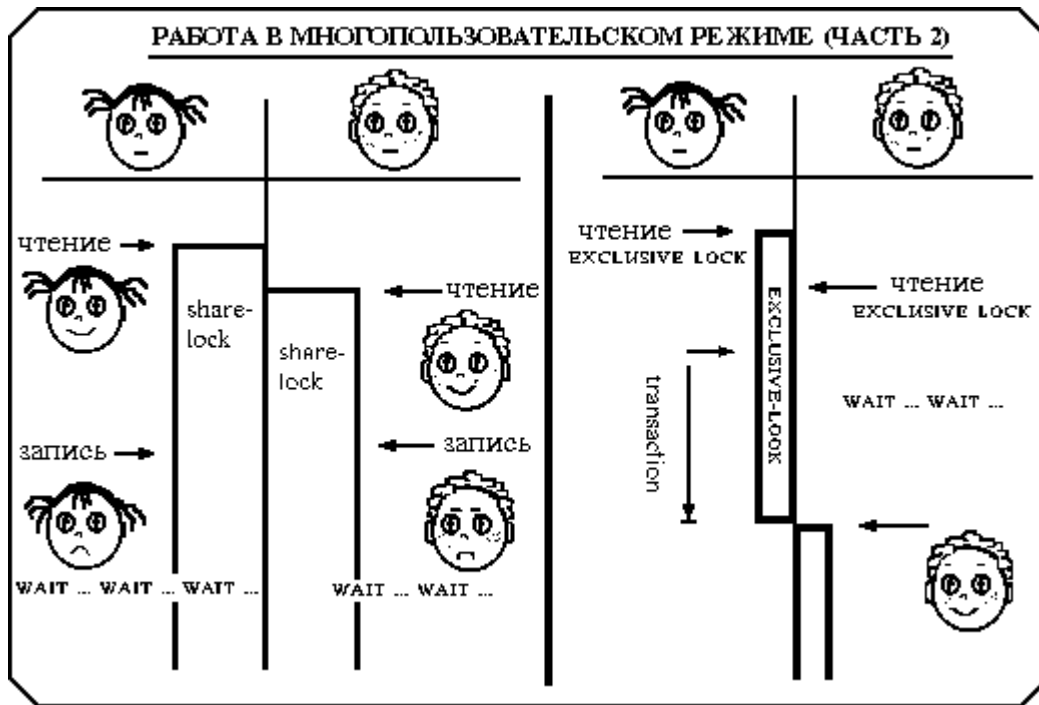
```

Программа второго пользователя:

```

FOR EACH student:
    UPDATE student.
END.
    
```

← SHARE-LOCK
← EXCLUSIVE-LOCK



После выполнения операторов чтения обе программы работают с первой записью таблицы student в режиме SHARE-LOCK. Завершив редактирование записи, убедимся, что ни одна из программ не сможет захватить запись как EXCLUSIVE-LOCK. Выход из этой ситуации - откат одной из программ.

Справиться с такой неприятностью можно разными способами, один из самых простых - указать режим EXCLUSIVE-LOCK при чтении записи, которую предстоит в дальнейшем изменять:

Программа первого пользователя:

```

FIND FIRST student EXCLUSIVE-LOCK.
UPDATE student.
    
```

Программа второго пользователя:

```

FOR EACH student EXCLUSIVE-LOCK:
    UPDATE student.
END.
    
```

Теперь один из пользователей захватит запись раньше, и другому будет предложено подождать.

Иногда удобно предусмотреть заранее и обработать программно такие ситуации, когда ожидание нежелательно. Рассмотрим следующий пример:

Пример 5.3.3.

```

/* DEFINE WIDGETS */
DEFINE BUTTON upd_button LABEL "Update".
DEFINE BUTTON btn_ok LABEL "OK" AUTO-GO.
DEFINE BUTTON btn_can LABEL "Cancel" AUTO-ENDKEY.
DEFINE QUERY qst FOR student.
DEFINE BROWSE brws_st QUERY qst DISPLAY num_st name_st WITH 5 DOWN.

/* DEFINE FRAMES */
DEFINE FRAME brws_frame
    brws_st upd_button WITH NO-LABELS.
DEFINE FRAME upd_frame
    
```



```

        student.name_st btn_ok btn_cancel WITH VIEW-AS DIALOG-BOX TITLE "Update student".
/* DEFINE TRIGGERS */
ON CHOOSE OF upd_button DO:
    GET CURRENT qst EXCLUSIVE-LOCK NO-WAIT.
    IF LOCKED student THEN MESSAGE "STUDENT LOCKED"
        VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
    ELSE DO:
        ENABLE ALL WITH FRAME upd_frame.
        UPDATE student.name_st WITH FRAME upd_frame.
        DISPLAY student.name_st WITH BROWSE brws_st.
        GET CURRENT qst NO-LOCK.
    END.
END.
/* MAIN LOGIC */
OPEN QUERY qst FOR EACH student.
ENABLE ALL WITH FRAME brws_frame.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.

```

Еще один алгоритм, рекомендуемый Progress'ом разработчикам для организации многопользовательского доступа к данным, состоит в следующем:

1. Прочитать запись в режиме NO-LOCK.
2. Предоставить пользователю возможность изменять значения полей в экранном буфере.
3. Поймать завершение работы пользователя с этой записью (с помощью конструкторов типа ON LEAVE OF field или ON CHOOSE OF ok-button).
4. Перечитать запись (FIND CURRENT или GET CURRENT) с опцией EXCLUSIVE-LOCK.
5. Проверить с помощью функции CURRENT-CHANGED отличаются ли перечитанные данные от тех, что лежат в буфере записи.
6. Если данные не изменились - завершить блок, если изменились - информировать об этом пользователя, вывести новые данные и повторить все заново.

При работе нескольких пользователей с одними и теми же объектами структуры базы данных только один из них в текущий момент может вносить изменения в схему базы (в DATA DICTIONARY).

Наличие файла с именем базы и расширением lk указывает на использование ее в данный момент.

Ограничение на количество замков задается серверным стартовым параметром -L.

III. ПРОГРАММИРОВАНИЕ ИНТЕРФЕЙСОВ И ОТЧЕТОВ

6. РАЗРАБОТКА ИНТЕРФЕЙСОВ

6.1. Фреймы

Одним из важнейших элементов интерфейса является фрейм. **Фрейм** - это прямоугольная область на экране внутри окна, которую Progress использует для вывода объектов. Progress использует фреймы для того, чтобы не нужно было позиционировать каждый выводимый объект. Progress сам позиционирует фреймы и объекты внутри него в соответствии с некоторыми правилами умолчания. Например, одно из них заключается в том, что метки выводятся над полями, другое - фрейм окружен прямоугольником и так далее. Все эти правила умолчания могут быть изменены по желанию программиста.

Фреймы автоматически сопоставляются следующим блокам:

```
REPEAT
FOR EACH
DO WITH FRAME
PROCEDURE
TRIGGER
```

Однако Progress сопоставляет этим блокам фреймы лишь в том случае, если в блоке происходит вывод какого-либо объекта. Исполним следующую процедуру:

Пример 6.1.1

```
DEFINE VARIABLE num AS INTEGER.
FOR EACH student:
  num = 0.
  DISPLAY name_st.
  FOR EACH marks OF student:
    num = num + 1.
  END.
  DISPLAY num LABEL "сдано экзаменов".
END.
```

Здесь три блока, однако фрейм будет сопоставлен лишь одному из них.

Фреймы, сопоставляемые блокам по умолчанию, принято называть непоименованными фреймами.

Как имена, так и характеристики сопоставляются фреймам с помощью конструктора WITH во фрейм-фразе операторов INSERT, DISPLAY, PROMPT-FOR, SET, UPDATE, в заголовках блоков или при явном описании фреймов в операторах DEFINE и FORM.

Перечислим некоторые из возможных характеристик фреймов:

CENTERED	- размещение фрейма в центре экрана;
FRAME <name>	- задание имени фрейма;
NO-BOX	- без рамки;
NO-LABELS	- без меток;
NO-UNDERLINE	- без подчеркивания меток;
ROW <expression>	- позиционирование фрейма по строке;
COLUMN <expression>	- позиционирование фрейма по столбцу;
SIDE-LABELS	- метки слева;
TITLE	- заголовок фрейма;
<expression> DOWN	- число итераций, допустимых в одном фрейме (без параметра - столько итераций, сколько позволяет экран).

При выводе полей внутри фрейма также можно явно указывать их характеристики:

AT <n>	- поле выводится начиная с n-ой позиции фрейма;
BLANK	- пробелы вместо значения поля;
FORMAT <s>	- формат изображения поля;
LABEL <s>	- задание метки поля;
COLUMN-LABEL <s1!s2>	- задание метки в несколько строк;
VALIDATE (<expression>,<s>)	- проверка области допустимых значений поля;
HELP <s>	- выдача подсказки при вводе информации в поле;
AUTO-RETURN	- ввод данных по заполнению поля.

Следующие примеры демонстрируют использование этих характеристик.

Пример 6.1.2

```

DISPLAY " Список студентов " WITH CENTERED NO-BOX.
FOR EACH student:
  DISPLAY name_st WITH CENTERED.
END.

```

Пример 6.1.3

```

FOR EACH student:
  DISPLAY name_st WITH TITLE "Список студентов." ROW 7 COLUMN 20.
END.

```

Пример 6.1.4

```

FOR EACH student:
  DISPLAY name_st WITH TITLE "Список студентов." ROW 7 SIDE-LABELS 5 DOWN.
END.

```

Пример 6.1.5

```

FOR EACH marks:
  UPDATE num_st code mark VALIDATE (mark >= 1 AND mark <= 5, " оценка от 1 до 5")
  HELP "введите оценку" WITH SIDE-LABELS.
END.

```

Пример 6.1.6

```

FOR EACH student:
  DISPLAY num_st COLUMN-LABEL "номер!зачетки" SPACE(5)
  name_st COLUMN-LABEL "имя!студента" WITH TITLE "Список студентов".
END.

```

Progress позволяет создавать также перекрывающиеся друг друга фреймы. К примеру, можно вывести всю информацию о студенте в одном фрейме, а в перекрывающем его фрейме - информацию о его оценках. Для этого второй фрейм нужно точно позиционировать и указать опцию OVERLAY:

Пример 6.1.7

```

FOR EACH student:
  DISPLAY student WITH 2 COLUMNS TITLE "student information".
  FOR EACH marks OF student:
    DISPLAY mark WITH OVERLAY TITLE "student marks" ROW 3 COLUMN 30.
  END.
END.

```

Следующие два примера иллюстрируют возможное описание фрейма через операторы FORM и DEFINE FRAME:

Пример 6.1.8

```

FORM
  student.num_st AT 3 NO-LABEL name_st AT 15 NO-LABEL sex at 40 SPACE(1)
  address AT 40 SPACE(1) phone AT 40 SPACE(1) bdate AT 40
  WITH FRAME a SIDE-LABELS ROW 3 COLUMN 10 TITLE "STUDENT".
FOR EACH student WITH FRAME a:
  DISPLAY student.
  PAUSE.
END.

```

Пример 6.1.9

```

DEFINE FRAME a
  student.num_st student.name_st
  WITH CENTERED TITLE "СПИСОК СТУДЕНТОВ" 15 DOWN ROW 5.
FOR EACH student WITH FRAME a:
  DISPLAY num_st name_st.
END.

```

Фреймы делятся на SINGLE и DOWN фреймы. DOWN фреймы - это те, в которых размещается множество итераций, а SINGLE - одна итерация.

По умолчанию фреймы, определенные через оператор FORM всегда SINGLE.

Во вложенных блоках только самый внутренний блок может иметь DOWN фрейм.

Фрейм выводится на экран либо оператором VIEW, либо при выводе данных внутри фрейма, либо установкой атрибута VISIBLE. Приведенные ранее примеры демонстрировали вывод фрейма при использовании оператора DISPLAY.

Следующий пример демонстрирует вывод фрейма через использование оператора VIEW:

Пример 6.1.10

```
DEFINE FRAME a student.name_st student .phone.
VIEW FRAME a.
PAUSE.
FOR EACH student WITH FRAME a:
    DISPLAY name_st phone.
    PAUSE.
END.
```

- Заметим, что использование оператора VIEW приводит фрейм к виду SINGLE.

Приведенный ниже пример демонстрирует использование операторов HIDE и CLEAR при работе с фреймами.

Пример 6.1.11

Оператор HIDE используется для стирания фреймов. Сравним результаты работы двух следующих процедур:

<pre>FOR EACH student: DISPLAY name_st. FOR EACH marks OF student: DISPLAY marks. END. END.</pre>	<pre>FOR EACH student: DISPLAY name_st. PAUSE. HIDE. FOR EACH marks OF student: DISPLAY marks. END. END.</pre>
---	--

Оператор CLEAR очищает фрейм от итераций. Сравним результат работы предыдущей процедуры со следующей:

```
FOR EACH student:
    DISPLAY name_st.
    PAUSE.
    CLEAR.
    FOR EACH marks OF student:
        DISPLAY marks.
    END.
END.
```

Фрейм - это объект интерфейса (типа container), которому, как обычно, соответствует свой набор атрибутов и методов. Синтаксис для использования атрибутов фрейма:

```
FRAME frame-name:attr-name
```

Синтаксис для методов:

```
FRAME frame-name:method-name(....)
```

Следующий пример демонстрирует некоторые атрибуты фрейма, приписанные ему как явно, так и по умолчанию.

Пример 6.1.12

```
/****** DEFINE FRAMES *****/
DEFINE FRAME Frame1 WITH SIDE-LABELS CENTERED ROW 5 TITLE "АТРИБУТЫ ФРЕЙМА".
/****** MAIN LOGIC *****/
DISPLAY FRAME Frame1:BOX LABEL "Ограничен ли фрейм рамкой?" SKIP
FRAME Frame1:SIDE-LABELS LABEL "Имеет ли фрейм боковые метки?" SKIP
FRAME Frame1:CENTERED LABEL "Центрирован ли фрейм в окне?" SKIP
```

```

FRAME Frame1: COLUMN LABEL "В какой колонке позиционирован фрейм?" SKIP
FRAME Frame1: ROW LABEL "В какой строке позиционирован фрейм?"
WITH FRAME Frame1.

```

Приведенный ниже пример показывает, как можно управлять цветом и шрифтами внутри фрейма при помощи атрибутов BGCOLOR, FGColor, FONT.

Пример 6.1.13

```

DEFINE BUTTON quit_btn LABEL "Выход".
DEFINE VARIABLE fgc_frm AS INTEGER VIEW-AS SLIDER MAX-VALUE 15 MIN-VALUE 0.
DEFINE VARIABLE bgc_frm AS INTEGER VIEW-AS SLIDER MAX-VALUE 15 MIN-VALUE 0.
DEFINE VARIABLE font_frm AS INTEGER VIEW-AS SLIDER MAX-VALUE 15 MIN-VALUE 0.
DEFINE FRAME x "Цвет пепа" fgc_frm SKIP(1) "Цвет фона" bgc_frm SKIP(1) "Шрифт" font_frm
SKIP(2) quit_btn AT 12 SKIP(3) WITH NO-LABELS CENTERED WIDTH 50.
ON VALUE-CHANGED OF fgc_frm FRAME x:FGCOLOR = INT(fgc_frm:SCREEN-VALUE).
ON VALUE-CHANGED OF bgc_frm FRAME x:BGCOLOR = INT(bgc_frm:SCREEN-VALUE).
ON VALUE-CHANGED OF font_frm FRAME x:FONT = INT(font_frm:SCREEN-VALUE).
ASSIGN
    fgc_frm:MAX-VALUE = COLOR-TABLE:NUM-ENTRIES - 1
    bgc_frm:MAX-VALUE = COLOR-TABLE:NUM-ENTRIES - 1
    font_frm:MAX-VALUE = FONT-TABLE:NUM-ENTRIES - 1.
ENABLE ALL WITH FRAME x.
WAIT-FOR CHOOSE OF quit_btn.

```

Отметим некоторые функции, связанные с фреймами:

FRAME-DB - возвращает логическое имя базы данных, к которой принадлежит поле, на которое указывает курсор;
 FRAME-FILE - возвращает имя таблицы, содержащей поле, на которое указывает курсор;
 FRAME-FIELD - возвращает имя поля, на которое указывает курсор;
 FRAME-INDEX - возвращает индекс элемента массива, на который указывает курсор;
 FRAME-NAME - возвращает имя активного фрейма;
 FRAME-VALUE- возвращает значение поля, на которое указывает курсор.

6.2. Динамические объекты интерфейса

Progress позволяет генерировать объекты интерфейса во время исполнения приложения. Такие объекты принято называть динамическими. Синтаксис для описания динамических объектов:

```

CREATE widget-type | VALUE(expression)
    widget-handle-variable
    [ IN WIDGET-POOL pool-name ]
    [ ASSIGN attribute = value [ attribute = value ] ... ]
    [ trigger-phrase ]

```

widget-type - тип объекта (button, frame, ...);
 VALUE(expression) - динамическое определение типа объекта;
 widget-handle-variable - имя переменной, описанной как widget-handle;
 pool-name - имя предварительно описанного пула (см. далее);
 attribute-параметры - позволяют задать атрибуты порождаемого объекта;
 trigger-параметр - один или более триггеров для объекта.

Несомненным достоинством динамических объектов является возможность создания произвольного их числа во время исполнения приложения. Например, можно изобразить для каждой записи таблицы-справочника свою кнопку, позволяющую вызывать определенные действия с данной записью, независимо от размера справочника на текущий момент. Некоторые трудности при работе с динамическими объектами могут возникать в связи с необходимостью более подробно описывать такие объекты: их нужно явно позиционировать во фрейме, заботиться об удалении и т.д.

Пример 6.2.1

Программа демонстрирует размещение во фрейме произвольного числа кнопок.

```

DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE num_buts AS INTEGER.
DEFINE VARIABLE butt_hand AS WIDGET-HANDLE.
DEFINE FRAME butt_frame WITH WIDTH 35 CENTERED TITLE "Buttons".
SET num_buts.

```

```

DO i = 1 TO num_buts:
  CREATE BUTTON butt_hand
  ASSIGN LABEL = STRING( i )
  FRAME = FRAME butt_frame:HANDLE
  ROW = TRUNC((i - 1) / 3,0) + 1
  COLUMN = (((i - 1) MOD 3) * 10) + 5
  SENSITIVE = TRUE.
END.
FRAME butt_frame:HEIGHT-CHARS = (num_buts / 3) + 3.
VIEW FRAME butt_frame.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.

```

Пример 6.2.2

Программа демонстрирует создание динамических text и fill-in объектов, представляющих поля базы данных (студенты и их оценки).

```

/***** DEFINE WIDGETS *****/
DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE j AS INTEGER.
DEFINE VARIABLE name_handle AS WIDGET-HANDLE.
DEFINE VARIABLE mark_handle AS WIDGET-HANDLE.
/***** DEFINE FRAMES *****/
DEFINE FRAME frame_stud WITH WIDTH 100 CENTERED TITLE "STUDENTS AND THEIR MARKS".
/***** MAIN LOGIC *****/
i = 0.
FOR EACH student:
  CREATE TEXT name_handle
  ASSIGN
    FRAME = FRAME frame_stud:HANDLE
    DATA-TYPE = "CHARACTER"
    FORMAT = "X(15)"
    SCREEN-VALUE = name_st
    ROW = 2 + i
    COLUMN = 2.
  j = 0.
  FOR EACH marks OF student:
    CREATE FILL-IN mark_handle
    ASSIGN
      FRAME = FRAME frame_stud:HANDLE
      DATA-TYPE = "INTEGER"
      FORMAT = "9"
      SCREEN-VALUE = STRING( mark )
      PRIVATE-DATA = STRING(ROWID( marks ))
      ROW = 2 + i
      COLUMN = 20 + j
      SENSITIVE = TRUE
    TRIGGERS:
    ON LEAVE
      DO: IF SELF:MODIFIED
        THEN
          DO:
            FIND marks WHERE
              ROWID(marks) = TO-ROWID(SELF:PRIVATE-DATA).
            mark = INTEGER(SELF:SCREEN-VALUE).
          END.
      END.
    END TRIGGERS.
    j = j + 5.
  END.
  i = i + 1.
END.
FRAME frame_stud:HEIGHT-CHARS = i + 2.
VIEW FRAME frame_stud.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.

```

Каждый динамический объект в момент создания приписывается некоторому пулу. Пул - это группа объектов с одной и той же областью действия, которая может быть удалена как единое целое. Progress создает один непоименованный пул для каждой клиентской сессии (работы приложения), и использует его как умалчиваемый для всех генерируемых во время сессии динамических объектов. Пул автоматически удаляется по концу сессии. В приложении могут быть созданы и поименованные пулы:

```
CREATE WIDGET-POOL pool-name [PERSISTENT] [NO-ERROR]
```

Такие пулы могут быть удалены автоматически (по концу сессии) или явно оператором:

```
DELETE WIDGET-POOL pool-name [NO-ERROR]
```

При удалении пула все динамические объекты, которые были к нему приписаны, также удаляются.

Пример 6.2.3

```

/***** DEFINE QUERY */
DEFINE QUERY s_query FOR student.
/***** DEFINE WIDGET *****/
DEFINE BROWSE s_browse QUERY s_query DISPLAY name_st WITH 5 DOWN.
DEFINE VARIABLE k AS INTEGER.
DEFINE VARIABLE course_handle AS WIDGET-HANDLE.
/***** DEFINE FRAMES *****/
DEFINE FRAME frame_course WITH TITLE "COURSES" WIDTH 30.
DEFINE FRAME frame_stud s_browse WITH CENTERED TITLE "STUDENTS ".
/***** DEFINE TRIGGERS *****/
ON VALUE-CHANGED OF s_browse
DO:
  DELETE WIDGET-POOL "MARKS" NO-ERROR.
  CREATE WIDGET-POOL "MARKS" PERSISTENT.
  k = 1.
  FOR EACH marks of student, course of marks:
    CREATE TEXT course_handle IN WIDGET-POOL "MARKS"
    ASSIGN
      FRAME = FRAME frame_course:HANDLE
      DATA-TYPE = "CHARACTER"
      FORMAT = "X(20)"
      SCREEN-VALUE = name_c + ": " + STRING(MARK)
      ROW = k
      COLUMN = 2.
    k = k + 2.
  END.
  FRAME frame_course:HEIGHT-CHARS = k.
  VIEW FRAME frame_course.
END.
/***** MAIN LOGIC *****/
FOR EACH mark:
  IF NOT CAN-FIND(course OF marks) THEN DELETE marks.
END.
OPEN QUERY s_query FOR EACH student.
ENABLE ALL WITH FRAME frame_stud.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

6.3. Окна

Progress автоматически создает окно, когда стартует приложение. Доступ к этому окну можно получить через системную переменную DEFAULT-WINDOW.

Большинство приложений используют только одно окно, определенное по умолчанию. В текстовом режиме использование других окон невозможно. В графическом же режиме Progress поддерживает множество окон. Можно создавать окна во время работы прикладной программы или изменять атрибуты окна, используемые по умолчанию. Доступ к текущему окну осуществляется через системную переменную CURRENT-WINDOW.

Перечислим некоторые атрибуты окна:

атрибут	Тип	описание
TITLE	STRING	заголовок окна
STATUS-AREA	LOGICAL	наличие статусной строки
MESSAGE-AREA	LOGICAL	наличие строки сообщений
POPUP-MENU	WIDGET-HANDLE	указатель на POPUP-MENU
MENUBAR	WIDGET-HANDLE	указатель на MENUBAR
FIRST-CHILD	WIDGET-HANDLE	указатель на первый фрейм
LAST-CHILD	WIDGET-HANDLE	указатель на последний фрейм

События, связанные с окном:

WINDOW-MINIMIZED
WINDOW-MAXIMIZED
WINDOW-RESTORED
WINDOW-CLOSE

Реакция приложения на эти события может быть организована через триггер:

ON WINDOW-CLOSE OF CURRENT-WINDOW QUIT.

или через WAIT-FOR оператор:

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.

Пример 6.3.1

```

DEFINE QUERY stq FOR student.
DEFINE BROWSE stb QUERY stq DISPLAY name_st WITH 15 DOWN.
DEFINE FRAME x
  stb.
DEFINE FRAME y
  student
  WITH SIDE-LABELS COLUMN 25 ROW 3 WIDTH 70.
ON ITERATION-CHANGED OF BROWSE stb
DO:
  DISPLAY student WITH FRAME y.
END.
ASSIGN DEFAULT-WINDOW:VIRTUAL-WIDTH-CHARS = 100
  DEFAULT-WINDOW:VIRTUAL-HEIGHT-CHARS = 15
  DEFAULT-WINDOW:WIDTH-CHARS = 60
  DEFAULT-WINDOW:HEIGHT-CHARS = 10
  DEFAULT-WINDOW:TITLE = "Student Browser"
  DEFAULT-WINDOW:BGCOLOR = 6.
OPEN QUERY stq FOR EACH student.
ENABLE stb WITH FRAME x.
APPLY "ITERATION-CHANGED" TO BROWSE stb.
ENABLE ALL WITH FRAME y.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

Создание дополнительных окон возможно с помощью оператора

```

CREATE WINDOW handle-variable
[ASSIGN attribute = value] [trigger-phrase].
```


Следующий пример демонстрирует работу с несколькими окнами:

Пример 6.3.2

```

DEFINE VAR dwin AS WIDGET-HANDLE.
DEFINE VAR cl AS INT EXTENT 7 INITIAL [4,12,14,10,11,9,13].
ON WINDOW-MAXIMIZED OF DEFAULT-WINDOW DO:
  DEF VAR i AS INT.
  DO i = 1 TO 7:
    CREATE WINDOW dwin
    ASSIGN BGCOLOR = cl[i]
    X = 1 + 90 * (i - 1)
    Y = 100
    VISIBLE = YES.
    PAUSE.
  END.
END.
ASSIGN DEFAULT-WINDOW:TITLE = "" DEFAULT-WINDOW:BGCOLOR = 0.
DISPLAY "КАК ОДНАЖДЫ ЖАК ЗВОНАРЬ ГОЛУБОЙ СТАЦИЛ ФОНАРЬ" WITH CENTERED.
WAIT-FOR WINDOW-CLOSE OF DEFAULT-WINDOW.

```

Пример 6.3.3

В этом примере демонстрируются приемы работы с окнами через PERSISTENT-процедуры.

```

DEFINE QUERY stud_query FOR student.
DEFINE BROWSE stud_browse QUERY stud_query
  DISPLAY num_st name_st WITH 10 DOWN.
DEFINE BUTTON b_exit LABEL "exit".
/***** DEFINE FRAMES *****/
DEFINE FRAME studframe
  stud_browse SPACE(2) b_exit.
/***** DEFINE TRIGGERS *****/
ON VALUE-CHANGED OF stud_browse IN FRAME studframe
DO:
  IF CAN-FIND(FIRST marks OF student WHERE mark < 3 )
  THEN
    MESSAGE "THIS IS BAD STUDENT! WOULD YOU LIKE TO LOOK HIS CARD"
    VIEW-AS ALERT-BOX QUESTION BUTTONS YES-NO
    UPDATE bul AS LOGICAL.
  IF bul
  THEN RUN C:\EXAMPLES\PART6\i06_0304.p PERSISTENT (ROWID(student)).
END.
/***** MAIN LOGIC *****/
ASSIGN
  CURRENT-WINDOW:MAX-HEIGHT-CHARS =
    FRAME studframe:HEIGHT-CHARS
  CURRENT-WINDOW:MAX-WIDTH-CHARS =
    FRAME studframe:WIDTH-CHARS.
OPEN QUERY stud_query FOR EACH student.
ENABLE ALL WITH FRAME studframe.
WAIT-FOR CHOOSE OF b_exit.

```

Процедура i06_0304.p:

```

DEFINE INPUT PARAM id AS ROWID.
DEFINE BUTTON b_cancel LABEL "cancel".
DEFINE VARIABLE studwin AS WIDGET-HANDLE.
/***** DEFINE FRAMES *****/
DEFINE FRAME studrecord
  student.num_st name_st address bdate sex phone b_cancel
  WITH 1 COLUMN.
/***** DEFINE TRIGGERS *****/
ON CHOOSE OF b_cancel IN FRAME studrecord
DO:
  DELETE PROCEDURE THIS-PROCEDURE.
END.

```

```

/***** MAIN LOGIC *****/
FIND student WHERE ROWID(student) = id.
CREATE WIDGET-POOL.
CREATE WINDOW studwin
    ASSIGN TITLE = "Student Card"
        MAX-HEIGHT-CHARS = FRAME studrecord:HEIGHT-CHARS
        MAX-WIDTH-CHARS = FRAME studrecord:WIDTH-CHARS.
THIS-PROCEDURE:CURRENT-WINDOW = studwin.
DISPLAY student WITH FRAME studrecord.
ENABLE ALL WITH FRAME studrecord.

```

6.4. Организация меню

Progress поддерживает два вида меню: MENUBAR и pop-up MENU.

MENUBAR - меню, ассоциируемое с окном и выводимое на его вершине. Оно состоит из заголовков подменю (SUB-MENU) или пунктов меню (MENU-ITEM), расположенных горизонтально. Выбор заголовка подменю приводит к появлению выпадающего меню (PULL-DOWN MENU).

Pop-up MENU - вертикальное меню, которое ассоциируется с объектом. Pop-up MENU появляется лишь в том случае, когда пользователь фокусирует соответствующий объект и щелкает по правой клавиши мыши или нажимает клавишу DEFAULT POP-UP (это может быть Shift-F10 или F4).

Любое меню - это иерархическая структура, которая состоит из подменю (SUB-MENU) и пунктов меню (MENU-ITEM).

Подменю определяется следующим образом:

```

DEFINE SUB-MENU colors
    MENU-ITEM one LABEL "красный"
    MENU-ITEM two LABEL "белый"
    MENU-ITEM three LABEL "желтый"
DEFINE SUB-MENU move
    MENU-ITEM m1 LABEL "Page-&Up"
        ACCELERATOR "Page-Up"
    MENU-ITEM m2 LABEL "Right"
    MENU-ITEM m3 LABEL "Left"
    MENU-ITEM m4 LABEL "Page-&DOWN"
        ACCELERATOR "Page-DOWN".

```

Опция ACCELERATOR обеспечивает доступ к пунктам меню через функциональные клавиши.

Ampersand (&) перед литерой означает, что эта литера будет выделена подчеркиком при выводе меню на экран и доступ к данному пункту меню будет обеспечен нажатием на клавиатуре этой литеры (или, если речь идет о верхнем – горизонтальном – уровне меню, нажатием этой литеры совместно с клавишей Alt). Допустимо выделение только одной литеры в метке. По умолчанию Progress выделяет первую литеру.

Оператор DEFINE MENU используется для описания MENUBAR подобным образом:

```

DEFINE MENU main MENUBAR
    SUB-MENU colors LABEL "цвета"
    SUB-MENU move LABEL "перемещения".

```

Привязка меню к соответствующему окну обеспечивается при помощи оператора ASSIGN и атрибута MENUBAR. Например:

```

ASSIGN DEFAULT-WINDOW:MENUBAR = MENU main:HANDLE.

```

Событие, связанное с меню - это выбор пункта меню. Его синтаксис:

```

ON CHOOSE OF MENU-ITEM menu-item-name

```

Пример 6.4.1

```

DEFINE SUB-MENU sounds
    MENU-ITEM sound_cat LABEL "Ко&т" ACCELERATOR "F9"
    MENU-ITEM sound_cow LABEL "Ко&пова" ACCELERATOR "Ctrl-W"
    MENU-ITEM sound_dog LABEL "Собака" ACCELERATOR "ALT-F7".

```

```

DEFINE MENU mainbar MENUBAR
  SUB-MENU sounds LABEL "Животные"
  MENU-ITEM exit_item LABEL "Выход".
ON CHOOSE OF MENU-ITEM sound_cat, MENU-ITEM sound_cow,
  MENU-ITEM sound_dog
DO:
  MESSAGE SELF:PRIVATE-DATA
  VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
END.
ON CHOOSE OF MENU-ITEM exit_item
DO:
  QUIT.
END.
ASSIGN MENU-ITEM sound_cat:PRIVATE-DATA = "Мя-я-у..."
  MENU-ITEM sound_cow:PRIVATE-DATA = "Му-у-у-у..."
  MENU-ITEM sound_dog:PRIVATE-DATA = "Гав! Гав! Гав!".
CURRENT-WINDOW:MENUBAR = MENU mainbar:HANDLE.
DISPLAY "Хотите услышать голоса домашних животных?" WITH CENTERED.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.

```

Пример 6.4.2 (AB)

Выполним приведенный выше пример средствами AB.

1. Откроем новое окно.



2. Войдем в окно свойств главного окна (через ) и определим следующее Menu Bar:

```

Животные
-Кот
-Корова
-Собака
Выход

```

Знак «-» перед метками означает понижение иерархии пункта меню. Для его появления следует нажать соответствующую кнопку в диалоговом окне меню.

Здесь же определим соответствующие "горячие клавиши" и "горячие буквы".

3. Опишем триггер на событие CHOOSE для пункта меню с меткой «Выход»:
QUIT.

4. Опишем триггер на событие CHOOSE для пункта меню с меткой «Кот»:
MESSAGE "Мя-у-у..".

5. Опишем триггер на событие CHOOSE для пункта меню с меткой «Корова»:
MESSAGE "Му-у-у..".

6. Опишем триггер на событие CHOOSE для пункта меню с меткой «Собака»:
MESSAGE "Гав! Гав! Гав!..".

7. Запишем программу в файл под именем **i06_0402.w**. Запустим программу на исполнение.

POP-UP меню также определяется с помощью оператора DEFINE MENU, но без опции MENUBAR. Привязка к объекту осуществляется через оператор ASSIGN.

Пример 6.4.3

```

/***** DEFINE WIDGETS *****/
DEFINE BUTTON n LABEL "Sequences".
DEFINE MENU num
  MENU-ITEM a LABEL "Armstrong's numbers"
  MENU-ITEM s LABEL "Prime numbers"
  MENU-ITEM p LABEL "Perfect numbers"
  RULE
  MENU-ITEM ex LABEL "Exit".
DEFINE FRAME num_frame
  n AT ROW 5 COLUMN 5.

```

```


/***** DEFINE TRIGGERS *****/
ON CHOOSE OF MENU-ITEM a, MENU-ITEM s, MENU-ITEM p
DO: MESSAGE SELF:PRIVATE-DATA
    VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
END.
/***** MAIN LOGIC *****/
ASSIGN n:POPUP-MENU = MENU num:HANDLE
MENU-ITEM a:PRIVATE-DATA = "1,2,3,4,5,6,7,8,9,153,370,371, ..."
MENU-ITEM s:PRIVATE-DATA = "1,2,3,5,7,11,13, ..."
MENU-ITEM p:PRIVATE-DATA = "6,28,496,8128, ...".
ENABLE n WITH FRAME num_frame.
WAIT-FOR CHOOSE OF MENU-ITEM ex.

```

Пример 6.4.4 (AB)

Выполним приведенный выше пример средствами AB.



1. Откроем новое окно (через ).
2. Разместим в окне кнопку с меткой "Занимательные ряды" и через окно свойств определим следующее PopUp Menu :
Числа Армстронга
Простые числа
Совершенные числа
RULE
Выход
3. Опишем триггер на событие CHOOSE для пункта меню с меткой «Выход»:
QUIT.
4. Опишем триггер на событие CHOOSE для пункта меню с меткой «Числа Армстронга»:
MESSAGE "1,2,3,4,5,6,7,8,9,153,370,371,..".
5. Опишем триггер на событие CHOOSE для пункта меню с меткой «Простые числа»:
MESSAGE "1,2,3,5,7,11,13,..".
6. Опишем триггер на событие CHOOSE для пункта меню с меткой «Совершенные числа»:
MESSAGE "6,28,496,8128,..".
7. Запишем программу в файл под именем **i06_0404.w**. Запустим программу на исполнение.

Пункты меню могут быть запрещены к фокусированию через атрибут SENSITIVE. Однако следует иметь ввиду, что если пункт меню был запрещен к фокусированию, он не может быть далее выбран. Убедимся в этом, добавив в какой-нибудь триггер предыдущего примера оператор
ASSIGN SELF:SENSITIVE = FALSE.

Можно использовать опцию TOGGLE-BOX для спецификации пункта меню в любом месте, за исключением первого уровня MENUBAR. Это дает пользователю дополнительные возможности для контроля состояния пункта меню. Например, прикладная программа может определить, был ли выбран соответствующий пункт меню через атрибут
CHECKED

Значение этого атрибута также может быть явно изменено в программе. Состояние соответствующего пункта меню визуализировано для пользователя как toggle-box. Когда пользователь изменяет содержимое toggle-box, Progress сопоставляет событие
VALUE-CHANGED

соответствующему пункту меню.

Пример 6.4.5 (AB)

1. Откроем новое окно.
2. Войдем в окно свойств главного окна и определим следующее Menu Bar:
Цвет-Шрифт
- Цвет пера
- Цвет фона
- Шрифт
Выход

Пункты "Цвет пера", "Цвет фона" и "Шрифт" отметим как TOGGLE-BOX.

Здесь же определим соответствующие "горячие клавиши" и "горячие буквы".

3. Опишем триггер на событие CHOOSE для пункта меню с меткой «Выход»:
QUIT.

4. Разместим в окне слайдер (SLIDER-1) для управления цветом пера. Через окно атрибутов сделаем его горизонтальным. Напишем триггер на событие VALUE-CHANGED для slider-1:

FRAME default-frame:FGCOLOR = INTEGER(SELF:SCREEN-VALUE).

5. Копированием через Clipboard создадим еще два слайдера (SLIDER-2, SLIDER-3) для управления цветом фона и шрифтами. В триггерах этих слайдеров поменяем FGColor на BGColor и FONT, соответственно.

6. В раздел Main Block перед оператором "RUN enable_UI." добавим:

ASSIGN slider-1:MAX-VALUE = COLOR-TABLE:NUM-ENTRIES - 1

slider-2:MAX-VALUE = COLOR-TABLE:NUM-ENTRIES - 1

slider-3:MAX-VALUE = FONT-TABLE:NUM-ENTRIES - 1.

7. Напишем триггер на событие VALUE-CHANGED для пункта меню с меткой «Цвет пера»:
slider-1:VISIBLE IN FRAME DEFAULT-FRAME = NOT slider-1:VISIBLE.

8. Напишем триггер на событие VALUE-CHANGED для пункта меню с меткой «Цвет фона»:
slider-2:VISIBLE IN FRAME DEFAULT-FRAME = NOT slider-2:VISIBLE.

9. Напишем триггер на событие VALUE-CHANGED для пункта меню с меткой «Шрифт»:
slider-3:VISIBLE IN FRAME DEFAULT-FRAME = NOT slider-3:VISIBLE.

10. Запишем программу в файл под именем **i06_0405.w**. Запустим программу на исполнение.

6.5 HELP - поддержка

В Progress существуют следующие способы организации HELP-поддержки для прикладных программ:

HELP-строка

плавающая подсказка ToolTip (с V8.2 Progress)

контекстно-зависимый HELP

ссылочный HELP

HELP-строка комментирует поля базы данных и объекты интерфейса. Она может быть определена в DATA DICTIONARY (через Field Properties), в AB (через Property Sheet для объекта), через атрибут HELP объекта или в HELP-опциях операторов FORM, DEFINE FRAME и интерактивных операторов. HELP-строка появляется в status-area родительского окна в момент выбора поля или объекта.

Пример 6.5.1

Программа, использующая HELP -строку.

DEFINE VARIABLE name AS CHARACTER.

SET name HELP "WHAT IS YOUR NAME?".

MESSAGE "HELLO, " + name + " !!!"

VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.

Плавающие подсказки (ToolTips) используются приложениями, работающими в среде Windows. ToolTip - это строка, которая может быть задана для объекта интерфейса в опции TOOLTIP оператора DEFINE, во VIEW-AS фразе, в AB (через Property Sheet), а также динамически через строковый атрибут TOOLTIP. Существует session-атрибут TOOLTIPS логического типа, который включает/выключает режим использования плавающих подсказок для всего сеанса.

Откроем в AB созданный в предыдущем разделе пример (6.4.5) и для слайдеров напишем в их Property Sheet какие-нибудь плавающие подсказки.

Контекстно-зависимая HELP-информация предоставляется пользователю по нажатию HELP-клавиши (обычно F1). Когда пользователь нажимает F1 во время выполнения программы, происходит поиск триггера для HELP-события ближайшего объекта, последовательно - для объектов data widgets, фрейма, окна, текущего приложения (ON HELP ANYWHERE). Если триггер не найден, выполняется код, хранящийся в файле applhelp.p в рабочем каталоге.

Пример 6.5.2

Applhelp.p может быть таким:

```

IF FRAME-FILE = "marks"
THEN IF FRAME-FIELD ="num_st"
    THEN FOR EACH student:
        DISPLAY student.
    END.
ELSE IF FRAME-FIELD = "code"
    THEN FOR EACH course:
        DISPLAY course.
    END.
ELSE MESSAGE "HELP для этого поля не предусмотрен"
    VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
ELSE MESSAGE "HELP для этой таблицы не предусмотрен"
    VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.

```

Перезапишем приведенный код в файл applhelp.p (в рабочую директорию). Исполним следующую процедуру, чтобы убедиться, что applhelp.p работает должным образом (будем нажимать F1 при обработке различных полей):

```

FOR EACH marks:
    UPDATE marks.
END.

```

Пример 6.5.3

В данной процедуре описан триггер для события HELP для поля code. Этот триггер перекроет программный код в applhelp.

```

DEFINE FRAME b
marks with 10 DOWN CENTERED TITLE "ОЦЕНКИ".
ON HELP OF FRAME b
    MESSAGE "ФРЕЙМ ДЛЯ РЕДАКТИРОВАНИЯ ОЦЕНОК".
ON HELP OF marks.code DO:
    FIND course WHERE course.code = INTEGER(FRAME-VALUE) no-error.
    IF AVAILABLE course THEN DISPLAY course.name_c course.name_t.
    ELSE MESSAGE "Нет курса с таким кодом".
END.
FOR EACH marks:
    UPDATE marks WITH FRAME b.
END.

```

Примером ссылочной HELP-информации является системный Progress-HELP. Разработчик может сам создать аналогичную HELP-систему для своего приложения под Windows, например, с помощью редактора Microsoft Word и утилиты hsw.exe, которая находится в Progress/bin (см. документацию PROGRESS HELP). Можно воспользоваться также любыми доступными системными средствами.

В следующем примере при выборе кнопки "HELP" (а также при нажатии F1) вызывается help для базы данных UNIV, подготовленный в файле univhelp.hlp.

Пример 6.5.4

```

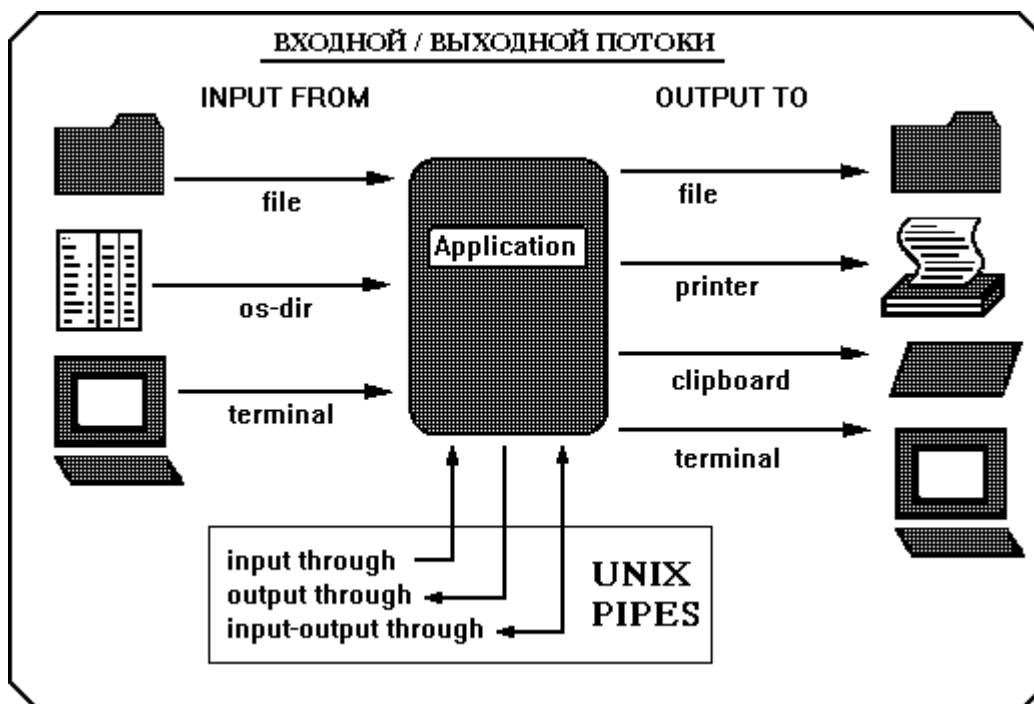
DEFINE VARIABLE helpfile AS CHARACTER INITIAL "c:\examples\part6\univhelp.hlp".
DEFINE BUTTON b_help LABEL "help".
DEFINE BUTTON b_exit LABEL "exit".
FORM
SKIP(2) SPACE(2) b_help SPACE(2) b_exit SPACE(2) SKIP(2) " F1 - help"
WITH FRAME a.
ON CHOOSE OF b_help
    SYSTEM-HELP helpfile CONTENTS.
ON HELP ANYWHERE
    SYSTEM-HELP helpfile CONTENTS.
ENABLE ALL WITH FRAME a.
WAIT-FOR CHOOSE OF b_exit.

```

7. СОЗДАНИЕ ОТЧЕТОВ

7.1. Переопределение входного и выходного потоков

Progress предоставляет возможность использовать потоки ввода/вывода различной направленности:



Выходной поток, по умолчанию, направляется на терминал. Оператор OUTPUT TO переопределяет направление выходного потока:

OUTPUT TO <filename>	- в файл;
OUTPUT TO PRINTER	- на принтер;
OUTPUT TO "CLIPBOARD"	- в MS-Windows Clipboard;
OUTPUT TO TERMINAL	- на терминал;
OUTPUT TO DEVICE	- на системное устройство.

Возможно использование формы OUTPUT TO VALUE(expression).

Область действия оператора OUTPUT TO - до ближайшего оператора OUTPUT CLOSE, OUTPUT TO или до конца процедуры.

Во время выполнения OUTPUT TO <filename> происходит создание нового файла. Если файл с таким именем уже существует, он будет "пересоздан". Для записи данных в конец существующего файла следует в оператор OUTPUT добавить опцию APPEND.

Пример 7.1.1

Следующая процедура выводит в файл содержимое таблицы student. Запустим ее на выполнение:

```
OUTPUT TO student.txt.
FOR EACH student:
  DISPLAY student.
END.
```

Вывод данных в файл или на принтер может осуществляться также оператором EXPORT. В операторе EXPORT можно указывать имя таблицы или отдельные поля. Опция DELIMITER определяет разделители, которые будут разграничивать данные в выходном файле или в распечатке (по умолчанию - пробел). Строковые данные при использовании оператора EXPORT будут заключены в кавычки.

Сравним результат работы следующей программы с результатом предыдущей:

```
OUTPUT TO student.txt.
FOR EACH student:
  EXPORT DELIMITER "," student.
END.
```

Входными данными, по умолчанию, считаются данные, вводимые с клавиатуры. Однако источник входного потока может быть переопределен с помощью оператора INPUT FROM:

INPUT FROM <filename>	- из файла;
INPUT FROM OS-DIR(directory)	- содержимое директории;
INPUT FROM TERMINAL	- с клавиатуры;
INPUT FROM DEVICE	- от системного устройства.

Возможно использование формы INPUT FROM VALUE(expression).

Ввод данных из файла может осуществляться любыми операторами ввода (например, оператором SET). Удобно использовать для ввода оператор IMPORT, который отличается от остальных тем, что не использует фрейм (и, следовательно, эффективнее), а также тем, что не сообщает об отказе ввода записи с повторным уникальным ключом. В операторе IMPORT можно указывать или имя таблицы, или отдельные поля, а также разделитель - DELIMITER (если в файле данные разделены не пробелами). Строковые значения в файле должны быть заключены в кавычки.

Конец строки в файле интерпретируется как конец записи. Конец файла трактуется как нажатие на клавишу END-ERROR. Если INPUT файлы содержат больше полей, чем это необходимо в процедуре, Progress игнорирует дополнительные данные. Если полей меньше - Progress устанавливает значения по умолчанию. Знак ^ используется в операторе ввода для пропуска полей.

Область действия оператора INPUT FROM - до ближайшего оператора INPUT CLOSE, INPUT FROM или конца процедуры.

Если во входном файле строковые данные не заключены в кавычки, можно использовать утилиту QUOTER, которая закавычивает строки, разделенные символом перевода каретки, запятыми, пробелами и т.д.:

```
QUOTER file1 > file2 [-d CHARACTER]
```

Где -d CHARACTER - определяет разделитель (по умолчанию - конец строки),

Пример 7.1.2

Следующая процедура позволит нам увеличить количество записей таблицы student вдвое путем ввода из файла student.txt "новых" записей. Следует лишь позаботиться об уникальности поля num_st - изменить номера студентов в файле на новые.

```
INPUT FROM student.txt.
REPEAT:
  CREATE student.
  IMPORT DELIMITER "," student.
END.
```

При выводе данных из директории (INPUT FROM OS-DIR(...)) можно получить информацию о именах файлов/директорий, полном пути к ним и атрибутах (таких как F – файл, D – директория, S – специальное устройство и т.д.)

Пример 7.1.3

Программа выводит на экран информацию о директории examples.

```
DEFINE VARIABLE file-name AS CHARACTER FORMAT "x(40)" LABEL "File".
DEFINE VARIABLE attr-list AS CHARACTER FORMAT "x(4)" LABEL "Attributes".
INPUT FROM OS-DIR("c:\examples").
REPEAT:
  SET file-name ^ attr-list.
END.
```

Если возникает необходимость работать с несколькими **потоками данных**, можно определить эти потоки следующим образом:

```
DEFINE STREAM stream-name.
```

Операторы назначения потока выглядят так:

```
INPUT STREAM stream-name FROM file-name.
INPUT STREAM stream-name FROM TERMINAL.
OUTPUT STREAM stream-name TO PRINTER.
OUTPUT STREAM stream-name TO file-name.
OUTPUT STREAM stream-name TO TERMINAL.
```


Операторы закрытия потока:

```
OUTPUT STREAM stream-name CLOSE.
INPUT STREAM stream-name CLOSE.
```

В операторах ввода/вывода при использовании потоков также следует специфицировать имя потока. Например:

```
DISPLAY STREAM stream1 student.
```

Пример 7.1.4

В этой процедуре предлагается ввести с клавиатуры имя файла и выводить в него информацию об указанных студентах.

```
DEFINE STREAM fl.
DEFINE VAR f-name AS CHARACTER FORMAT "X(12)".
DEFINE VARIABLE I AS LOGICAL.
SET f-name LABEL "Введите имя файла ".
OUTPUT STREAM fl TO VALUE(f-name).
FOR EACH student:
    DISPLAY name_st.
    MESSAGE "Выводить в файл информацию об этом студенте?"
    VIEW-AS ALERT-BOX QUESTION BUTTONS yes-no UPDATE I.
    IF I THEN EXPORT STREAM fl num_st name_st bdate.
END.
```

Можно осуществлять ввод/вывод данных в ASCII-формате интерактивно, в среде DATA ADMINISTRATION:

```
DATA ADMINISTRATION → ADMIN → Import Data
DATA ADMINISTRATION → ADMIN → Export Data
```

7.2. Отчеты

Progress позволяет создавать различные виды отчетов. Простые отчеты генерируются с помощью операторов вывода с указанием во фрейм-фразе структуры отчета и направляются на экран, на печать или в файл. Допустимы группирование данных (BREAK BY), сортировки внутри групп, вычисляемые колонки, контрольные суммы и другие обобщающие вычисления:

```
AVERAGE      - среднее значение;
COUNT        - количество;
MAXIMUM        - максимальное значение;
MINIMUM        - минимальное значение;
TOTAL          - сумма.
```

Пример 7.2.1

Следующая программа выводит номера студентов, их оценки и коды предметов, группируя по предметам в порядке убывания балла с подсчетом среднего балла по предмету.

```
FOR EACH marks BREAK BY code BY mark DESCENDING:
    DISPLAY num_st mark (AVERAGE BY code) FORMAT "9.99" code.
END.
```

Приведенная ниже программа является небольшим усложнением предыдущей. Здесь вместо кодов выводятся имена студентов и названия предметов, причем последние - только в начале соответствующей группы. В дополнение к среднему баллу подсчитывается и количество оценок.

```
FOR EACH marks BREAK BY code BY mark DESCENDING:
    IF FIRST-OF(code) THEN DO:
        FIND course OF marks NO-ERROR.
        IF NOT AVAILABLE course THEN UNDO.
        DISPLAY name_c.
    END.
    FIND student OF marks NO-ERROR.
    IF NOT AVAILABLE student THEN UNDO.
    DISPLAY name_st mark (AVERAGE COUNT BY marks.code) FORMAT "9.99".
END.
```

Оператор ACCUMULATE накапливает вычисляемые (агрегатные) значения в цикле по записям таблицы, а функция ACCUM позволяет использовать эти значения в дальнейшем.

Пример 7.2.2.

Выведем статистику по оценкам студентов.

```
FOR EACH student, EACH marks OF student:
    ACCUMULATE mark (AVERAGE COUNT).
END.
DISPLAY "STATISTICS FOR MARKS:" SKIP(1)
    " num marks   = " (ACCUM COUNT mark) SKIP
    " average mark = " (ACCUM AVERAGE mark)
    WITH NO-LABELS.
```

Модифицируем программу, добавив группирование по полю sex с вычислением количества записей в группе и подсчетом процентного соотношения полов:

```
FOR EACH student BREAK BY sex:
    ACCUMULATE student.num_st (SUB-COUNT BY sex).
    FOR EACH marks OF student:
        ACCUMULATE mark (AVERAGE COUNT).
    END.
    IF LAST-OF(sex) THEN
        DISPLAY sex " - "
            (ACCUM SUB-COUNT BY sex student.num_st) * 100 / COUNT-OF(sex) "%"
        WITH NO-LABEL.
    END.
DISPLAY "STATISTICS FOR MARKS:" SKIP(1)
    " num marks   = " (ACCUM COUNT mark) SKIP
    " average mark = " (ACCUM AVERAGE mark)
    WITH OVERLAY ROW 6 COLUMN 10 NO-LABELS.
```

Данные, определенные как SLIDER, SELECTION-LIST, TOGGLE-BOX или RADIO-SET при выводе на печать или в файл дадут пустое значение. Чтобы избежать этого, нужно пойти по одному из следующих путей:

1. использовать опцию STREAM-IO во фрейм-фразе;
2. указать опцию TEXT при выводе данных ;
3. использовать оператор PUT вместо DISPLAY (выводит на печать или во внешний файл - без создания фреймов - данные с фиксированных позиций).

Пример 7.2.3

В DATA DICTIONARY изменим описание поля sex:

Field Properties → View-As → VIEW-AS TOGGLE-BOX.

Посмотрим на результат работы процедуры:

```
OUTPUT TO student.txt.
FOR EACH student:
    DISPLAY name_st sex.
END.
```

Следующие три процедуры могут быть использованы для вывода содержимого поля sex:

```
OUTPUT TO student.txt.
FOR EACH student:
    DISPLAY name_st sex WITH STREAM-IO.
END.
```

```
OUTPUT TO student.txt.
FOR EACH student:
    DISPLAY name_st sex VIEW-AS TEXT.
END.
```

```
OUTPUT TO student.txt.
FOR EACH student:
    PUT name_st AT 1 sex AT 40.
END.
```

При выводе в файл или на печать данные типа FILL-IN окружены пустыми строками. Чтобы выходной документ получился более сжатым, следует использовать опцию USE-TEXT.

Пример 7.2.4

Сравним:

```
FOR EACH student:
  DISPLAY name_st address.
END.
```

и

```
FOR EACH student:
  DISPLAY name_st address WITH USE-TEXT.
END.
```

При работе с данными, представляющими собой длинные строки, удобно форматировать вывод, описывая их как EDITOR.

Пример 7.2.5

Сравним:

```
FOR EACH student:
  DISPLAY num_st name_st bdate phone address.
END.
```

и

```
FOR EACH student:
  DISPLAY num_st name_st bdate phone
  address VIEW-AS EDITOR INNER-CHARS 5 INNER-LINES 3.
END.
```

Следующий пример демонстрирует ряд возможностей для создания в Progress страничного форматированного отчета:

Пример 7.2.6

```
OUTPUT TO student.txt PAGE-SIZE 30.
DEFINE FRAME hdr HEADER
  "*****" SKIP
  "DATE: " TODAY "**** STUDENT REPORT ****" AT 25
  "page" AT 60 PAGE-NUMBER FORMAT ">9" SKIP
  "*****" SKIP
  WITH PAGE-TOP STREAM-IO.
DEFINE FRAME ftr HEADER
  "*****" SKIP
  "CONTINUE ON NEXT PAGE" AT 25
  WITH PAGE-BOTTOM STREAM-IO.
DEFINE FRAME a
  name_st NO-LABEL sex AT 25 bdate AT 35 SKIP
  address AT 25 NO-LABEL phone AT 45 SKIP(1)
  "_____ " SKIP
  WITH SIDE-LABELS STREAM-IO.

FOR EACH student USE-INDEX name_st:
  VIEW FRAME hdr.
  VIEW FRAME ftr.
  DISPLAY name_st sex bdate address phone WITH FRAME a.
END.
```

7.3. RESULTS - интерактивный построитель отчетов

Для создания отчетов в интерактивном режиме существует специальная компонента Progress - RESULTS. Эта среда позволяет формировать запрос (query) к базе данных и оформлять результат выборки различными способами - в виде Browse, Report или Form. RESULTS позволяет также экспортировать данные из базы, генерировать тексты программ на Progress 4GL и создавать наклейки (Labels).

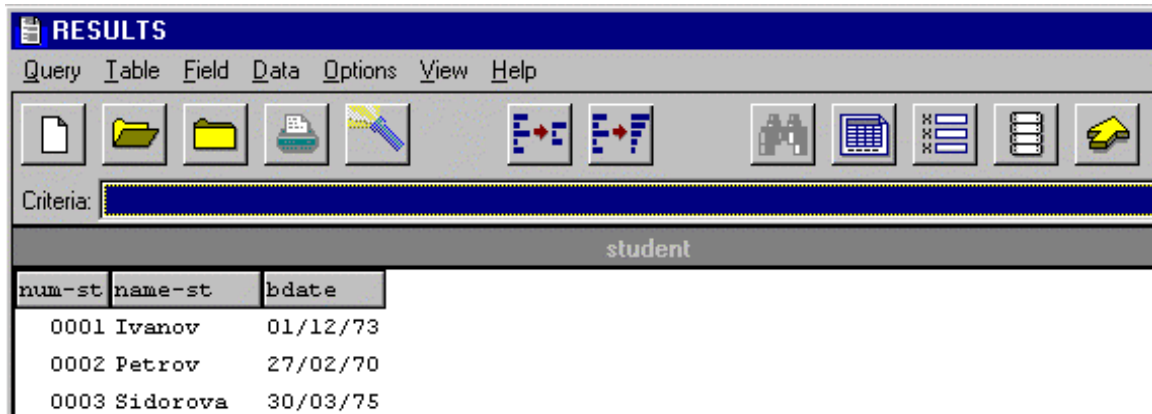
Войти в RESULTS можно через Desktop или через пункт меню Tools любой компоненты. При первом сеансе работы с базой через RESULTS создаются вспомогательные файлы конфигурации (в той же директории, где лежит база).


Пример 7.3.1

Создание простого запроса.

1. Войдем в RESULTS и откроем новый запрос в форме Browse: Query → New → Browse...


В окне Add/Remove Tables выберем таблицу student, а затем в окне Add/Remove Fields - поля num_st, name_st и bdate.



2. Установим критерием сортировки поле bdate: Data → Sort Ordering ().

3. Установим критерием выборки фильтр num_st < 8 : Data → Selection ().

4. Добавим в запрос остальные поля таблицы student: Field → Add/Remove Fields.

5. Изменим форму запроса: View → As Report (). Чтобы в этом случае увидеть результат

выборки, следует войти в режим Preview ().

6. Еще раз изменим форму запроса: View → As Form ().

7. Сохраним созданный запрос под любым именем.

Пример 7.3.2

Создание запроса по связным таблицам и генерация процедуры.

1. Откроем новый запрос в форме Report по таблицам course (поля name_c, name_t) и marks (поля num_st, mark): Query → New → Report...

2. Посмотрим, как выглядит этот запрос, будучи представленным в форме Form (редактирование окажется недоступным).

3. Добавим к запросу еще одну связную таблицу - student (поле name_st): Table → Add/Remove Tables...

4. Сгенерируем процедуру на Progress 4GL, которая формирует точно такой же запрос: Query → Generate...

5. Сохраним запрос. Закроем RESULTS и из процедурного редактора запустим на выполнение сгенерированную нами процедуру.

8. КОНСТРУИРОВАНИЕ ПРИКЛАДНОЙ ПРОГРАММЫ

Создадим простейшую прикладную программу, обслуживающую базу данных UNIV. Приемы создания приложения, предлагаемые в этой главе, характерны для стандартного событийного (событийно-ориентированного) стиля программирования.

1) Главное окно программы и меню.

Откроем новое окно. Вставим в текущий фрейм объект типа image и кнопку. Разместим на кнопке длинную надпись, используя наложение на нее двух текстовых полей. Для кнопки напишем триггер на событие CHOOSE:

```
MESSAGE "Автор программы - .... Дата создания - ...".
```

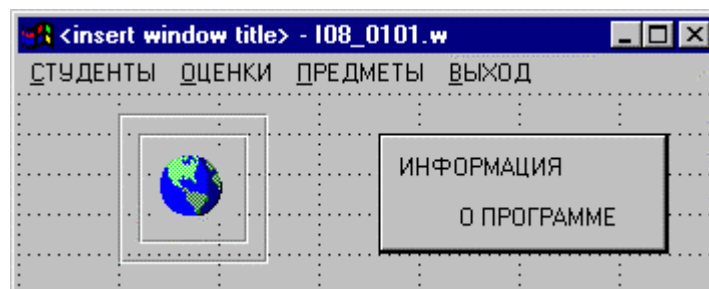
Создадим MENUBAR для нашего окна (Property Sheet → Menu Bar):

```
СТУДЕНТЫ
ОЦЕНКИ
- Оценки студентов
- Оценки по предметам
ПРЕДМЕТЫ
ВЫХОД
```

- Символы "-" определяют иерархию пунктов меню и вставляются с помощью управляющей кнопки ">>".

Для пункта меню "ВЫХОД" напишем триггер на событие CHOOSE:

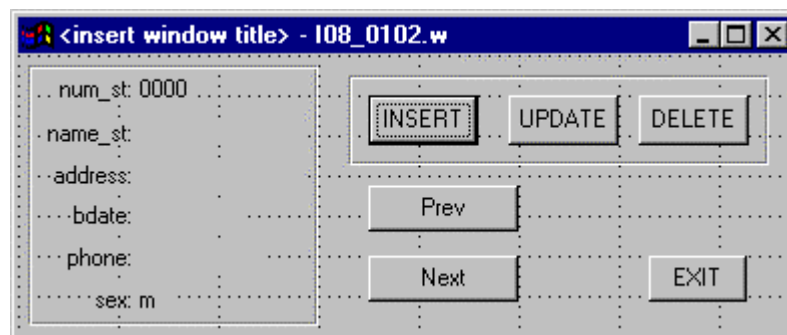
```
QUIT.
```



Запишем окно в файл под именем `c:\examples\part8\i08_0101.w`.

2) Пункт меню "СТУДЕНТЫ".

Откроем новое окно. Ранее вся работа велась во фрейме основного окна приложения. Новое окно - окно подпрограммы - должно храниться в отдельном файле и вызываться из основной программы оператором RUN.



Разместим в разделе Definitions вставку include-файла, содержащего описание фрейма для редактирования записей таблицы student (типа dialog-box):

```
{c:\examples\part4\i04_dial.i}
```

Разместим кнопку выхода с меткой "Exit" и напишем триггер для события CHOOSE:

```
APPLY "CLOSE" TO THIS-PROCEDURE.
```

Разместим во фрейме все поля таблицы student (пиктограмма Fields). В Property Sheet каждого поля укажем View-As-Text.

Для навигации по записям таблицы разместим стандартные кнопки Prev и Next.

Разместим кнопку INSERT для добавления новых записей и напишем триггер для нее:

```
CREATE student.  
ENABLE ALL WITH FRAME dial-frame.  
UPDATE student WITH FRAME dial-frame.  
DISPLAY student WITH FRAME DEFAULT-FRAME.
```

Разместим кнопку UPDATE для редактирования записей и триггер для нее:

```
ENABLE ALL WITH FRAME dial-frame.  
UPDATE student WITH FRAME dial-frame.  
DISPLAY student WITH FRAME DEFAULT-FRAME.
```

Разместим кнопку DELETE для удаления записей и триггер для нее:

```
DELETE student.  
APPLY "CHOOSE" TO Btn_Next.
```

Убедимся, что наша подпрограмма работает (Run).

Запишем созданное окно в файл **c:\examples\part8\i08_0102.w**, а пункту меню "СТУДЕНТЫ" в основном окне сопоставим триггер на событие CHOOSE:

```
RUN c:\examples\part8\i08_0102.w.
```

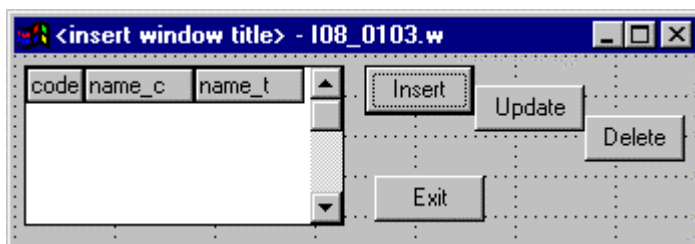
Исполним наше приложение (выберем Run в то время, когда активным будет основное окно приложения).

3) Пункт меню "ПРЕДМЕТЫ".

Откроем новое окно. Создадим это окно таким образом, чтобы оно "встраивалось" в вызывающее. Такое окно должно иметь атрибут Suppress Window = yes. В этом новом окне, как и в любом другом, будет свой фрейм. Во время исполнения программы этот фрейм будет встраиваться в окно вызывающей программы, перекрывая фрейм основного окна. Suppress-окно должно также храниться в отдельном файле и вызываться из основной программы оператором RUN .

Итак, в Property Sheet нового окна поставим флажок Suppress Window.

Копируем во фрейм кнопку выхода с меткой "Exit" из предыдущего окна. Разместим во фрейме также BROWSE по таблице course (все поля) и кнопки "Insert", "Update", "Delete".



Модифицируем Query для Browse - вместо NO-LOCK укажем SHARE-LOCK (войдем из Property Sheet для Browse в окно описания Query и найдем место для соответствующей установки - под списком таблиц). В результате каждая запись, выбранная пользователем в Browse, будет помещаться в буфер с замком, позволяющим редактирование и удаление. Поэтому в триггерах теперь не нужно писать GET CURRENT ... SHARE-LOCK. Это не лучший, но вполне допустимый способ.

Напишем триггер для кнопки "Insert" на событие CHOOSE:

```
INSERT course WITH VIEW-AS DIALOG-BOX.  
OPEN QUERY browse-1 FOR EACH course.  
DISPLAY browse-1 WITH FRAME DEFAULT-FRAME.
```

Напишем триггер для кнопки "Update" на событие CHOOSE:

```
UPDATE course WITH VIEW-AS DIALOG-BOX 1 COLUMN.  
DISPLAY course WITH BROWSE browse-1.
```

Напишем триггер для кнопки "Delete" на событие CHOOSE:

```
MESSAGE "Вы уверены?" VIEW-AS ALERT-BOX QUESTION
      BUTTONS YES-NO UPDATE m AS LOGICAL.
IF m THEN DO: DELETE course.
  OPEN QUERY browse-1 FOR EACH course.
  DISPLAY browse-1 WITH FRAME DEFAULT-FRAME.
END.
```

Убедимся, что подпрограмма работает (Run).

Запишем созданное окно в файл **c:\examples\part8\i08_0103.w**, а пункту меню "ПРЕДМЕТЫ" в основном окне сопоставим триггер на событие CHOOSE:

```
RUN c:\examples\part8\i08_0103.w.
```

Исполним приложение (выберем Run в то время, когда активным будет основное окно приложения).

4) Пункт меню "ОЦЕНКИ СТУДЕНТОВ".

Откроем новое окно. Копируем во фрейм нового окна кнопку выхода с меткой "Exit" из предыдущего окна.

Разместим в новом окне BROWSE с полем name_st из таблицы student (browse-2).

Разместим в окне еще один BROWSE с полями mark и name_c из таблиц marks и course (browse-3).

Напишем триггер на событие VALUE-CHANGED для объекта browse-2:

```
OPEN QUERY browse-3 FOR EACH marks OF student,EACH course OF marks.
VIEW browse-3.
```

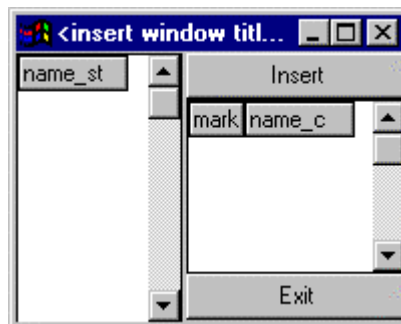
Добавим в Main Block после оператора "RUN enable_UI.":

```
APPLY "VALUE-CHANGED" TO browse-2.
```

Для добавления записей таблицы marks используем возможность создания в AppBuilder Dialog-Boxes, хранимых в виде отдельных файлов.

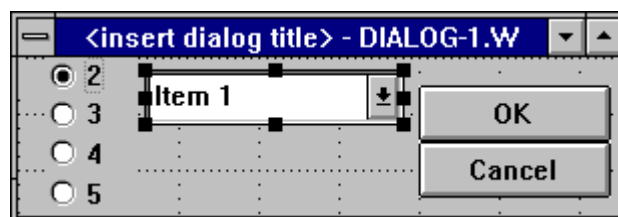
Разместим во фрейме кнопку "Insert". Напишем для нее триггер на событие CHOOSE:

```
RUN c:\examples\part8\dialog-1.w (student.num_st). /*dialog-1.w создадим после*/
OPEN QUERY browse-3 FOR EACH marks OF student,EACH course OF marks.
DISPLAY browse-3 WITH FRAME default-frame.
```



Запишем созданное окно в файл **c:\examples\part8\i08_0104.w**,

Создадим Dialog-Box "Dialog-1" (через New). Удалим кнопку Help, оставим кнопки завершения OK (Auto-Go) и отмены CANCEL (Auto-End-Key).



В раздел Definitions добавим описание параметра через который будем получать номер текущего студента (так как сфера действия буфера записи - текущая процедура):

DEFINE INPUT PARAMETER prm AS INTEGER.

Добавим во фрейм объект типа radio-set для отображения оценки. В его свойствах определим Buttons: "2", 2, "3", 3, "4", 4, "5", 5 и укажем на связь с marks.mark (через кнопку Database Field в окне свойств).

Добавим объект типа combo-box. В свойствах COMBO-BOX отметим опцию List-Item-Pairs, а тип данных установим в соответствии с определениями поля code из таблицы course (integer).

В разделе Main Block до оператора "RUN enable_UI." сформируем для него список значений:

```
COMBO-BOX-1:DELETE(1).
FOR EACH course:
COMBO-BOX-1:ADD-FIRST(course.name_c, course.code).
END.
COMBO-BOX-1 = INTEGER(COMBO-BOX-1:ENTRY(1)).
```

Напишем для кнопки ОК триггер на событие CHOOSE:

```
CREATE marks.
marks.num_st = prm.
ASSIGN combo-box-1.
marks.code = COMBO-BOX-1.
ASSIGN marks.mark.
```

В Property Sheet для Dialog-Box уберем флажок Open the Query. Запишем Dialog Box в файл с именем **c:\examples\part8\dialog-1.w**.

Убедимся, что подпрограмма работает (Run).

Пункту меню "ОЦЕНКИ СТУДЕНТОВ" в основном окне сопоставим триггер на событие CHOOSE:

```
RUN c:\examples\part8\i08_0104.w.
```

Исполним приложение (выберем Run в то время, когда активным будет основное окно приложения).

5) Пункт меню "ОЦЕНКИ ПО ПРЕДМЕТАМ".

Самостоятельно обеспечьте пункт меню "ОЦЕНКИ ПО ПРЕДМЕТАМ".

6) Компиляция головной программы и всех подпрограмм.

Смотри приложение 10.

Запустим новый сеанс Progress с нашим приложением в качестве стартового (если работа идет в режиме single-user, предварительно отсоединимся от базы данных и используем при запуске стартовый параметр -1):

```
prowin32.exe c:\examples\db\univ -p c:\examples\part8\i08_0101.r -1
```

- Программа prowin32.exe запускает на выполнение среду разработчика, начиная с процедуры, указанной параметром -p. Найти prowin32.exe можно в директории PROGRESS\bin.

9. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ INTERFACE-TEMPLATE

Interface-template - это файл (обычно с расширением wx), содержащий 4GL-код, который может использоваться как шаблон при разработке однотипных фрагментов приложений. Interface-template может быть создан как средствами редактора, так и в АВ. Универсальность применения interface-template в различных приложениях достигается использованием препроцессорных имен.

9.1. Препроцессор

Препроцессор - это компонента Progress, которая обрабатывает исходный текст процедуры до компилятора. Препроцессор выполняет текстовые подстановки в соответствии с препроцессорными директивами, расположенными в исходном тексте. Использование таких директив способствует построению приложений, удобных для понимания, модификации и переноса на другие платформы.

Препроцессорные директивы - это операторы, которые начинаются с символа амперсанта (&) и анализируются только препроцессором.

Директивы &GLOBAL-DEFINE и &SCOPE-DEFINE.

Эти директивы позволяют создавать препроцессорные имена. Синтаксис для их использования:

```
&GLOBAL-DEFINE preprocessor-name definition
&SCOPE-DEFINE preprocessor-name definition
```

В приведенных выше определениях preprocessor-name - это строка литер. Если эта строка не помещается на одной линии - следует применять знак тильды (~) как символ продолжения. В дальнейшем препроцессорные имена используются заключенными в фигурные скобки, с символом амперсанта перед идентификатором:

```
{&preprocessor-name}
```

Директива &UNDEFINE

Для уничтожения препроцессорного определения используется директива &UNDEFINE:

```
&UNDEFINE preprocessor-name
```

Директива &MESSAGE.

Директива &MESSAGE позволяет вывести сообщение во время компиляции. Ее синтаксис:

```
&MESSAGE text-string
```

Директивы &IF, &THEN, &ELSEIF, &ELSE, &ENDIF.

Эти директивы являются управляющими при компиляции кода. При этом выражения после директив &IF и &ELSEIF могут состоять только из препроцессорных имен, констант, операторов и функций (см. Приложение 11).

Ниже приведен пример определения и использования препроцессорных имен.

Пример 9.1.1

```
/* PREPROCESSOR DEFINITIONS */
&GLOBAL-DEFINE table student
&GLOBAL-DEFINE fields student.num_st student.name_st student.bdate student.address ~
student.sex student.phone
{c:\examples\part9\include1.i}
```

Текст INCLUDE-файла:

```
DEFINE BUTTON btn-next LABEL "next {&table}".
DEFINE BUTTON btn-prev LABEL "prev {&table}".
DEFINE BUTTON btn-first LABEL "first {&table}".
DEFINE BUTTON btn-last LABEL "last {&table}".
DEFINE QUERY t-query FOR {&table}.
/* DEFINE FRAME */
DEFINE FRAME t-frame
{&fields} SKIP
btn-first btn-prev btn-next btn-last
WITH 1 COLUMN TITLE "{&table}".
```

```

/* DEFINE TRIGGERS */
ON CHOOSE OF btn-first
DO:
  GET FIRST t-query.
  DISPLAY {&fields} WITH FRAME t-frame.
END.
ON CHOOSE OF btn-prev
DO:
  GET PREV t-query.
  IF NOT AVAILABLE {&table}
  THEN GET FIRST t-query.
  DISPLAY {&fields} WITH FRAME t-frame.
END.
ON CHOOSE OF btn-next
DO:
  GET NEXT t-query.
  IF NOT AVAILABLE {&table}
  THEN GET LAST t-query.
  DISPLAY {&fields} WITH FRAME t-frame.
END.
ON CHOOSE OF btn-last
DO:
  GET LAST t-query.
  DISPLAY {&fields} WITH FRAME t-frame.
END.

/* MAIN LOGIC */
OPEN QUERY t-query FOR EACH {&table}.
ENABLE ALL WITH FRAME t-frame.
WAIT-FOR CLOSE OF CURRENT-WINDOW.

```

Препроцессорные имена различаются по области определения в зависимости от того, какой директивой они были определены. Директива &GLOBAL-DEFINE порождает глобальные определения, которые действительны в пределах compile-unit (группы файлов, компилируемых вместе). Директива &SCOPE-DEFINE порождает неглобальные определения, которые действительны до конца файла, содержащего это определение, и включенных в него файлов.

Progress определяет препроцессорное имя в соответствии с препроцессорным определением (definition). Это определение, в свою очередь, может использовать препроцессорные имена подобно тому, как указано в следующем примере:

Пример 9.1.2

```

/* PREPROCESSOR DEFINITIONS */
&GLOBAL-DEFINE table student
&GLOBAL-DEFINE fields name_st
&GLOBAL-DEFINE list student.num_st student.name_st student.sex student.bdate student.address
student.phone
&GLOBAL-DEFINE where-cond WHERE {&fields} MATCHES "*"A*"
{c:\examples\part9\include2.i}

```

Ниже приведен текст INCLUDE-файла:

```

/* DEFINE WIDGETS */
DEFINE QUERY comquery FOR {&table}.
DEFINE BROWSE combrow QUERY comquery
  DISPLAY {&fields} WITH 7 DOWN NO-LABELS.
/* DEFINE FRAMES */
DEFINE FRAME frame1
  combrow WITH TITLE "{&table}".
DEFINE FRAME frame2
  {&list} WITH 1 COLUMN.
/* DEFINE TRIGGERS */
ON VALUE-CHANGED OF combrow IN FRAME frame1

```

```

DO:
    DISPLAY {&list} WITH FRAME frame2.
END.
/* MAIN LOGIC */
OPEN QUERY comquery FOR EACH {&table} {&where-cond}.
ENABLE ALL WITH FRAME frame1.
APPLY "VALUE-CHANGED" TO combrow.
WAIT-FOR CLOSE OF CURRENT-WINDOW.

```

Стандартные препроцессорные имена.

Progress автоматически строит следующие препроцессорные имена:

{&opsys}	- строка литер, содержащая имя операционной системы;
{&file-name}	- строка литер, содержащая имя компилируемого файла;
{&line-number}	- строка литер, содержащая номер текущей линии в компилируемом файле;
{&sequence}	- целое, которое увеличивается на единицу всякий раз, когда происходит ссылка на это имя (начальное значение - 0);
{&window-system}	- строка литер, содержащая имя windows-системы, в которой компилируется файл.

Пример 9.1.3

В следующем примере обратите внимание на контекст использования препроцессорных имен.

```
DISPLAY "{&opsys}" "{&window-system}".
```

AB автоматически генерирует определения препроцессорных имен. Вот некоторые из них:

WINDOW-NAME	- указатель на текущее окно;
FRAME-NAME	- имя первого фрейма;
BROWSE-NAME	- имя первого Browse (и соответствующего Query);
TABLES-IN-QUERY-{&FRAME-NAME}	- список таблиц в Query для фрейма;
TABLES-IN-QUERY-{&BROWSE-NAME}	- список таблиц в Query для Browse;
FIRST-TABLE-IN-QUERY-{&FRAME-NAME}	- первая таблица в Query для фрейма;
FIRST-TABLE-IN-QUERY-{&BROWSE-NAME}	- первая таблица в Query для Browse;
DISPLAYED-FIELDS	- поля фрейма;
DISPLAYED-OBJECTS	- объекты фрейма;
SELF-NAME	- имя объекта, по которому работает триггер;
LIST-1, ..., LIST-6	- списки объектов, формируемые разработчиком.

Среди упомянутых выше препроцессорных имен особого внимания заслуживает препроцессорная переменная SELF-NAME. AB перед блоком триггеров для каждого создаваемого объекта интерфейса объявляет препроцессорную переменную SELF-NAME и инициализирует ее именем объекта интерфейса (в конце блока переменная уничтожается). Этот бесхитростный прием позволяет использовать препроцессорную переменную {&SELF-NAME} вместо имени объекта интерфейса в контексте соответствующих триггеров.

Пример 9.1.4 (AB)

1. Откроем новое окно.
2. Разместим в текущем фрейме кнопку.
3. Напишем для кнопки триггер на событие CHOOSE:

```
MESSAGE "HELLO".
```

4. Через меню Compile → Code Preview найдите операторы объявления и уничтожения препроцессорной переменной SELF-NAME в разделе Control Triggers.

5. Измените триггер для кнопки на событие CHOOSE:

```
MESSAGE "КООРДИНАТЫ КНОПКИ:" + STRING({&SELF-NAME}:X) +
    "," + STRING({&SELF-NAME}:Y).
```

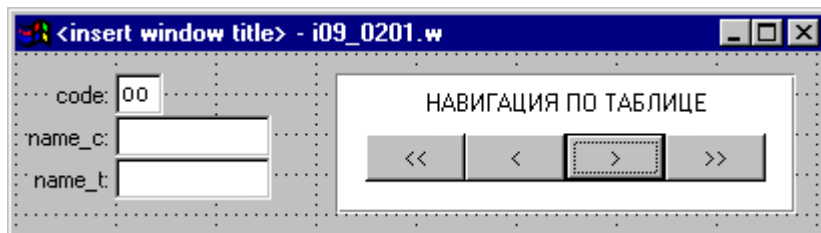
6. Через пункты меню Edit → Copy, а затем Edit → Paste продублируйте кнопку и разместите ее правее оригинала.

7. Запустите программу на исполнение.

9.2. Создание и использование interface-template

Средствами АВ создадим interface-template для навигации по таблице базы данных.

Пример 9.2.1 (АВ)



1. Откроем новое окно.
2. Разместим в текущем фрейме любые поля из любой таблицы.
3. Через меню Compile → Code Preview посмотрим на препроцессорные определения в разделе Preprocessor Definition.
4. Разместим во фрейме кнопку с меткой "<<" и напомним триггер на событие CHOOSE:


```
GET FIRST {&FRAME-NAME}.
DISPLAY {&DISPLAYED-FIELDS} WITH FRAME {&FRAME-NAME}.
```

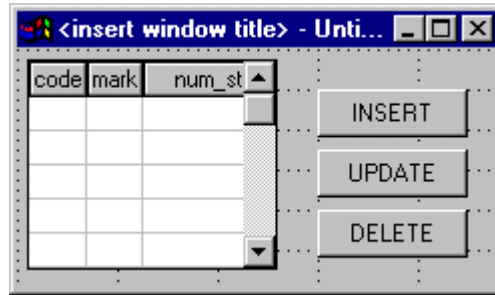
 - Препроцессорные имена следует выбрать из списка через Insert → Preprocessor Name...
5. Справа от предыдущей кнопки разместим новую кнопку с меткой "<" и напомним триггер на событие CHOOSE:


```
GET PREV {&FRAME-NAME}.
IF NOT AVAILABLE
  {&FIRST-TABLE-IN-QUERY-{&FRAME-NAME}}
THEN GET FIRST {&FRAME-NAME}.
DISPLAY {&DISPLAYED-FIELDS} WITH FRAME {&FRAME-NAME}.
```
6. Справа от предыдущих кнопок разместим новую кнопку с меткой ">" и напомним триггер на событие CHOOSE:


```
GET NEXT {&FRAME-NAME}.
IF NOT AVAILABLE
  {&FIRST-TABLE-IN-QUERY-{&FRAME-NAME}}
THEN GET LAST {&FRAME-NAME}.
DISPLAY {&DISPLAYED-FIELDS} WITH FRAME {&FRAME-NAME}.
```
7. Справа от предыдущей кнопки разместим новую кнопку с меткой ">>" и напомним триггер на событие CHOOSE:


```
GET LAST {&FRAME-NAME}.
DISPLAY {&DISPLAYED-FIELDS} WITH FRAME {&FRAME-NAME}.
```
8. Заключим созданные кнопки в Rectangle и "покрасим" его (для этого в Property Sheet укажем флажок Filled и выберем цвет фона). Вставим в качестве заголовка в сформированную навигационную панель текстовый объект "НАВИГАЦИЯ ПО ТАБЛИЦЕ" (через пиктограмму T в палитре).
9. Запустим программу на исполнение и убедимся, что она работает должным образом.
10. Выделим во фрейме навигационную панель (как группу объектов) и через пункт меню Edit → Copy to file... запишем полученный interface-template в файл с именем **panel.wx**. Текущее окно теперь можно закрыть без записи.
11. Откроем новое окно. Разместим во фрейме поля из какой-либо другой таблицы. Позаботимся о свободном месте в той части фрейма, где разместится панель (она "помнит" свое расположение в родительском фрейме). Через Edit → Insert from file... найдем и вставим во фрейм нашу панель panel.wx. Убедимся, что программа работает.

Пример 9.2.2 (AB)



Создадим interface-template для редактирования таблицы.

1. Откроем новое окно.
2. Разместим в текущем фрейме Browse для любой таблицы, указав для вывода в нем все поля таблицы.
3. Разместим в текущем фрейме кнопку "INSERT" и напомним для нее триггер на событие CHOOSE:


```
INSERT {&FIRST-TABLE-IN-QUERY-}&BROWSE-NAME}
  WITH VIEW-AS DIALOG-BOX.
OPEN QUERY {&BROWSE-NAME}
  FOR EACH {&FIRST-TABLE-IN-QUERY-}&BROWSE-NAME}.
DISPLAY {&BROWSE-NAME} WITH FRAME {&FRAME-NAME}.
```
4. Разместим кнопку "UPDATE" и напомним для нее триггер на событие CHOOSE:


```
GET CURRENT {&BROWSE-NAME} SHARE-LOCK.
UPDATE {&FIRST-TABLE-IN-QUERY-}&BROWSE-NAME}
  WITH VIEW-AS DIALOG-BOX 1 COLUMN.
DISPLAY {&FIRST-TABLE-IN-QUERY-}&BROWSE-NAME}
  WITH BROWSE {&BROWSE-NAME}.
```
5. Разместим кнопку "DELETE" и напомним для нее триггер на событие CHOOSE:


```
MESSAGE "ARE YOU SURE?" VIEW-AS ALERT-BOX QUESTION
  BUTTONS YES-NO UPDATE I AS LOGICAL.
IF I THEN DO:
  GET CURRENT {&BROWSE-NAME} SHARE-LOCK.
  DELETE {&FIRST-TABLE-IN-QUERY-}&BROWSE-NAME}.
IF NOT {&BROWSE-NAME}:DELETE-SELECTED-ROWS()
  THEN MESSAGE "It is not possible".
END.
```
6. Запустим программу на исполнение и убедимся, что она работает должным образом.
7. Выделим во фрейме группу кнопок корректировки и выхода, а затем через пункт меню Edit → Copy to file... запишем полученный interface-template в файл с именем **update.wx**.
8. Проверим работоспособность нового interface-template: создадим новое окно; разместим Browse по любой другой таблице и вставим наш новый interface-template update.wx. Запустим программу на выполнение.

IV. КОМПОНЕНТНАЯ МОДЕЛЬ РАЗРАБОТКИ ПРИЛОЖЕНИЙ

10. СТАНДАРТНЫЕ СРЕДСТВА ADM

10.1. Идеология создания приложений в ADM

Начиная с V.8, Progress предлагает такой подход к разработке приложений как **компонентное** программирование - **Application Development Model** (в версии 9 эта технология получила свое дальнейшее развитие и называется **ADM2**). Методология ADM базируется на построении 4GL-приложений из переиспользуемых компонент (процедур) - **Smart Objects**. Эти объекты с точки зрения языка представляют собой внешние процедуры.

ADM2 предлагает следующие базовые классы (или типы) объектов - **templates**:



SmartContainer (**SmartWindow**, **SmartDialog**, **SmartFrame**) – контейнер для размещения экземпляров smart-объектов.



SmartDataObject – объект для получения записей из базы данных и замещения измененных записей.



SmartDataBrowser - объект для показа и редактирования данных запроса, извлеченного из базы объектом **SmartDataObject**, в виде **Browse**.



SmartDataViewer – форма для демонстрации и редактирования отдельных записей, извлеченных из базы объектом **SmartDataObject** (может использоваться и как специальный вариант контейнера).



SmartPanel – управляющая панель из кнопок (для навигации, редактирования и т.д.).



SmartFolder – объект, управляющий навигацией по страницам smart-контейнера.



SmartDataField – объект для представления отдельного поля из базы данных в нестандартном виде внутри **SmartDataViewer**.



SmartFilter – объект, используемый для задания динамического фильтра.



SmartToolBar – объект, имеющий вид панели и/или меню с набором функций для управления навигацией, редактированием, транзакциями и фильтрами в **SmartWindow** (совмещает в себе функциональность нескольких других smart-объектов).



SmartObject – универсальный smart-объект с минимальным кодом (еще одно название – **SimpleSmartObject**), предназначенный для создания пользовательских visual-templates.

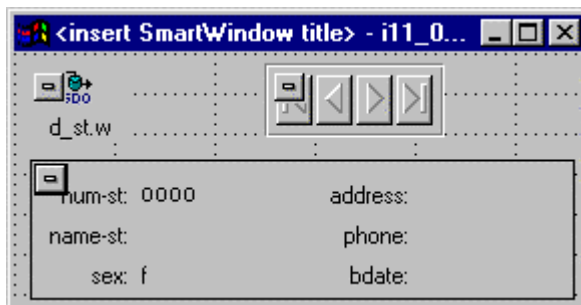
Разработчик может использовать templates для создания своих объектов (**masters**), изменяя свойства соответствующих базовых объектов, добавляя элементы интерфейса, программный код и записывая получившийся объект в виде файла под собственным именем (template → master). Такой пользовательский объект, master, может быть впоследствии многократно использован в приложении (каждое такое включение называется экземпляром объекта - **instance**), а также может, в свою очередь, послужить исходной точкой для создания нового объекта (masters → masters).

Можно создавать достаточно сложные объекты, встраивая в **SmartContainers** другие **SmartObjects**, обычные объекты интерфейса и программный код. Например, можно создать свой объект для просмотра записей таблицы, "собрать" его из **SmartWindow**, **SmartDataObject** (извлекающего данные из базы), **SmartPanel** навигационного типа (позволяющей передвигать указатель в query объекта **SmartDataObject**), и **SmartDataViewer** (экранной формы для работы с записью). Откомпилированная и сохраненная в файле, такая процедура будет являться пользовательским объектом (master), и ее экземпляры могут многократно использоваться в различных приложениях.

Если в дальнейшем какой-либо объект (master) будет изменен, то эти изменения автоматически отразятся во всех приложениях, где используются экземпляры этого объекта, поскольку в приложении присутствует только ссылка на объект (то есть, вызов соответствующей процедуры). Объект не зависит от других объектов, он переиспользуем. Каждый объект содержит в себе базовую информацию, скопированную из моделей базовых типов (они хранятся в ...\\progress\\src\\adm). Это информация о том, что объект "знает" и что он "делает" ("know & do"). Объект, как правило, хранит ("знает") свои атрибуты и переменные. "Действия" объекта представлены методами.

При построении нового объекта следует учитывать определенные правила "включения". Smart-объекты не могут располагаться внутри обычных контейнеров (WidgetContainers). SmartContainers (к ним относятся SmartWindow, SmartDialog и SmartFrame) могут включать в себя другие Smart-объекты. В некоторых случаях SmartDataViewer также может выступать в роли контейнера.

Основная среда для работы со smart-объектами – Application Builder. Экранная форма, собранная из объектов, во время разработки может иметь, например, такой вид:



Взаимодействие объектов друг с другом осуществляется главным образом через механизм связей (**SmartLinks**). При установлении связи один из объектов объявляется источником (SOURCE), другой – получателем (TARGET).

В ADM предусмотрен некоторый набор видов связей (разработчик может создать и свои виды). Каждый объект, в зависимости от своего типа, может быть источником некоторого набора видов связи, и получателем другого набора.

Для каждого объекта, исполняющего роль источника или получателя в связи определенного вида, характерны свои события, для обслуживания которых этот вид связи предназначен. Например, для Navigation-Source характерны такие события как "fetchFirst" и "fetchNext", а для Commit-Source – "commitTransaction" и "undoTransaction".

Ниже перечислены основные виды связей:

CONTAINER - связывает объект-контейнер с объектом включенным (например: SmartWindow -> SmartDataViewer);

NAVIGATION - связывает объект, имеющий интерфейс для навигации с объектом, способным отзываться на требования навигации (например: SmartPanel(navigation) -> SmartDataObject);

DATA - передает записи от источника к получателю (например: SmartDataObject -> SmartDataViewer);

TABLEIO - передает указания от источника к получателю о необходимости модификации данных (например: SmartPanel(update) -> SmartDataViewer);

UPDATE - передает результаты изменений (например: SmartDataViewer -> SmartDataObject);

GROUP ASSIGN - связывает объекты, подлежащие обработке в единой транзакции (например: SmartDataViewer -> SmartDataViewer).

COMMIT – источник дает указание получателю завершить или откатить транзакцию (например: SmartPanel -> SmartDataObject);

FILTER – связывает источник, имеющий интерфейс для определения критериев фильтрации данных, с получателем, формирующим запрос к данным (например: SmartFilter -> SmartDataObject);

PAGE - связывает источник, имеющий интерфейс для выбора страниц, с получателем, способным скрывать и показывать страницы (например: SmartFolder -> SmartWindow);

PAGE_n - связывает источник, управляющий сокрытием и визуализацией набора страниц, с получателем, расположенным на странице с номером n (например: SmartWindow -> SmartBrowser на Page n).

Один и тот же объект может быть участником нескольких связей, выступая в некоторых из них источником, а в других - получателем. Например, SmartDataObject может быть TARGET-NAVIGATION для навигационной панели и, одновременно, SOURCE-DATA для объекта SmartViewer.

SmartLinks формируются обычно во время построения интерфейса. Часть этой работы берет на себя Progress Advisor. Во время встраивания объекта в интерфейс Advisor предлагает подтвердить необходимость создания очередной связи, которая представляется ему уместной, а в неоднозначных ситуациях предлагает выбрать один из возможных вариантов или отказаться от всех.

Связи могут быть установлены и разработчиком на любом этапе конструирования через редактор связей (Link Editor), который открывается, например, через кнопку SmartLinks в окне Procedure Settings (исключением из общего правила являются связи типа CONTAINER - они всегда устанавливаются автоматически, поэтому AppBuilder их не показывает).

Динамическая установка связей (во время выполнения приложения), а также их уничтожение могут быть обеспечены вызовом соответствующих методов.

Подробное описание структуры Smart-объектов и механизмов реализации их взаимодействия приведено в 11-ой главе.

10.2. Технология сборки приложений

Построение интерфейса в стиле ADM можно разбить на три этапа:

1. Разработка функциональной схемы, включающая в себя определение необходимых SmartObjects и связей между ними.
2. Создание отдельных SmartObjects.
3. Сборка экрана из главного контейнера и встраиваемых в него SmartObjects:
 - подготовленных на втором этапе,
 - созданных при разработке предыдущих приложений,
 - предложенных разработчиками Progress (например, стандартная навигационная панель).

Пример 10.2.1 (AB)

Просмотр записей таблицы student через SmartDataViewer.

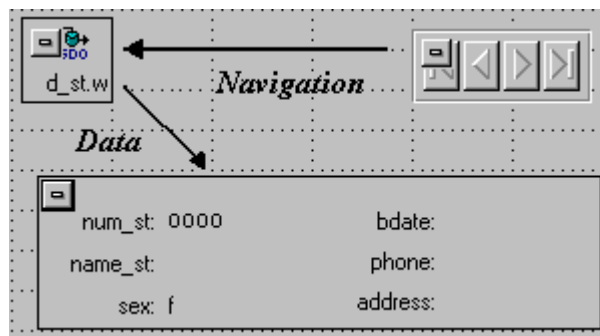
1 этап:

В данном приложении нам потребуются следующие объекты:

- SmartDataViewer - для отображения данных на экране;
- SmartDataObject - для формирования запроса к базе и извлечения данных;
- SmartPanel (навигационного типа) - для навигации по запросу.

Связи между объектами будут такими:


- Navigation - для передачи информации о сдвиге указателя в запросе;
- Data - для передачи информации о новой текущей записи.



2 этап:

На этом этапе нужно создать два объекта: SmartDataObject и SmartDataViewer. SmartPanel для навигации можно взять стандартную.


1. Создадим объект типа SmartDataObject, предназначенный для выборки данных из таблицы student:

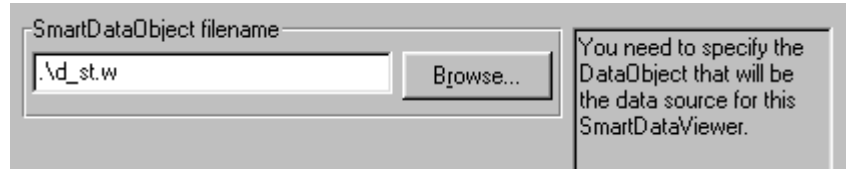
- через кнопку  откроем список объектов и выберем в нем SmartDataObject;
- следуя указаниям Wizard определим query (кнопка "Define query") по всем записям таблицы student;
- далее Wizard предложит выбрать поля таблицы (кнопка "Add Fields") – выберем все поля таблицы student, укажем, что их редактирование не потребуется (кнопка "None Updatable");



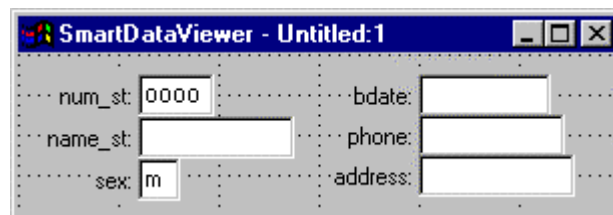
- сохраним новый объект в директории c:\examples\part10 под именем d_st.w.
- закроем SmartDataObject.

2. Создадим объект типа SmartDataViewer, предназначенный для вывода на экран данных из SmartDataObject:

- через  выберем SmartDataViewer;
- в качестве источника данных для него (SmartDataObject pathname) укажем созданный в предыдущем пункте d_st.w:






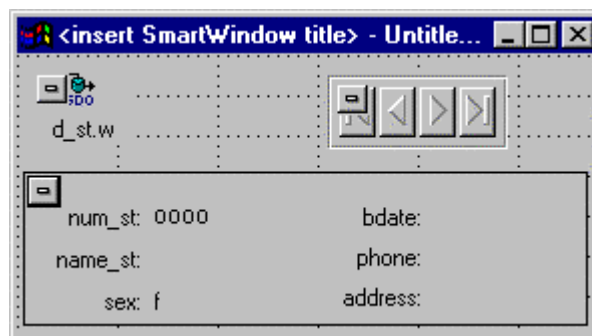
- далее выберем все поля таблицы student;
- сохраним новый объект в директории c:\examples\part10 под именем v_st.w.



3 этап:

1. Создадим объект типа SmartWindow () , разместим в нем:

- объект d_st.w (через кнопку  на панели объектов);
- объект v_st.w (через ) , согласимся на создание связи типа Data от d_st, которую предложит Advisor;
- объект SmartPanel () - навигационная панель rnavico.r, согласимся на создание связи типа Navigation к d_st.



2. Сохраним окно под именем i10_0201.w. Выполним приложение.

Объекты для представления данных - SmartDataViewer и SmartDataBrowser -находятся в сильной зависимости от источника данных – объекта SmartDataObject. При их создании всегда указывается источник - уже существующий SmartDataObject, а для визуализации предлагается выбирать только из тех полей, которые были указаны в источнике. При последующей модификации SmartDataViewer или SmartDataBrowser, когда программист захочет изменить список полей, ему будет предложен тот же выбор.

При сборке экрана можно связать объект для представления данных с "чужим" SmartDataObject, и эта связь будет допустима, если поля подойдут ("matching signature"), то есть для каждого поля найдется одноименное в источнике, в том числе по именам таблиц (student.num_st и marks.num_st – это неподходящие поля).

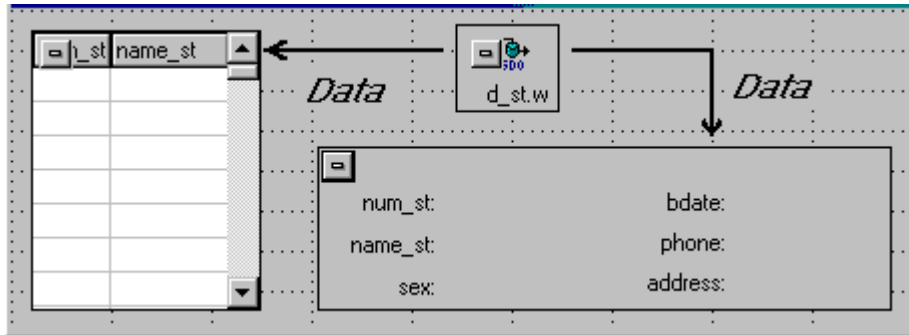
Следует быть очень внимательными при модификации SmartDataObject, по которому определены уже какие-либо другие объекты. Если перестанут совпадать сигнатуры, то в готовых приложениях поломаются связи и даже в случаях общей работоспособности программ соответствующие объекты при старте останутся “дохленькими”.

Пример 10.2.2 (AB)

Просмотр записей таблицы student через SmartDataBrowser и SmartDataViewer.

1 этап:

Функциональная схема объектов на экране:




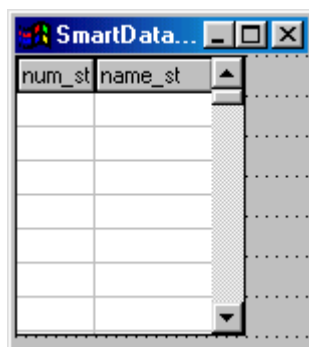
SmartDataObject будет извлекать данные из базы, SmartDataBrowser позволит пользователю выбрать запись по номеру и имени студента и, тем самым, переустановить указатель запроса в SmartDataObject. Все поля текущей записи будут выводиться на экран через SmartDataViewer.

Можно использовать уже созданные в предыдущем примере SmartDataObject и SmartDataViewer, а SmartDataBrowser создадим на втором этапе.

2 этап:


1. Создадим объект типа SmartDataBrowser:




- через  выберем SmartDataBrowser;
- с помощью Wizard определим для него в качестве источника данных объект d-st, созданный в предыдущем примере;
- на следующем экране нажмем кнопку Add Fields и выберем поля num_st и name_st;

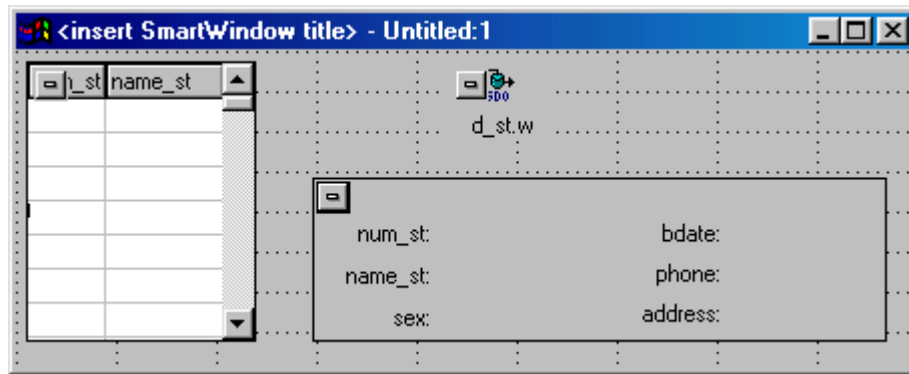


2. сохраним SmartDataBrowser в директории c:\examples\part10 под именем b_st.w.

3 этап:

1. Через  создадим объект типа SmartWindow, разместим в нем:

- объект d_st.w () , созданный ранее;
- b_st.w () , согласимся на создание связи типа Data от d_st;
- v_st.w () , созданный ранее, согласимся на создание связи типа Data от d_st;



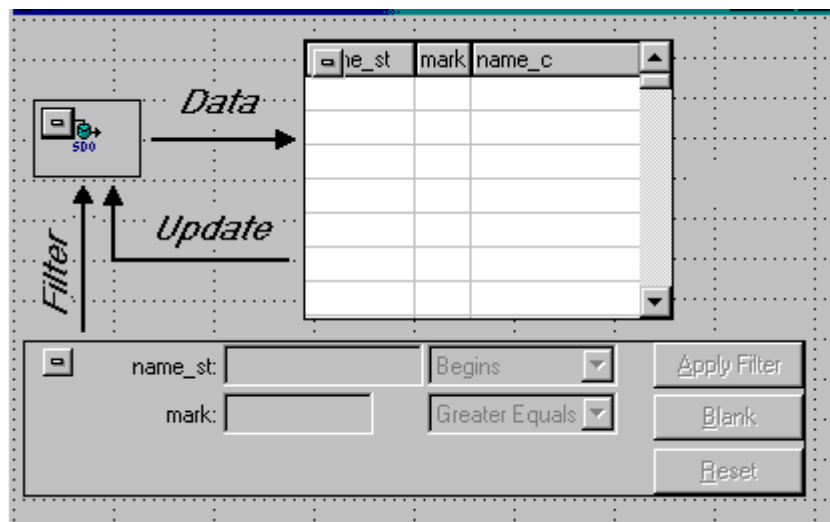
2. Сохраним окно под именем i10_0202.w. Выполним приложение.

При создании объекта SmartDataObject всегда указываются такие характеристики полей, как редактируемость. Если объект для представления данных создавался по источнику, в котором поля были на тот момент определены как не редактируемые, то даже изменение источника, не позволит редактировать данные через этот объект - связь вида Update от него создать не удастся. В следующем примере для источника данных очень важна редактируемость поля mark, значения которого мы собираемся изменять в SmartDataBrowser.

Пример 10.2.3 (AB)

Просмотр и редактирование оценок всех студентов с использованием динамического фильтра.


1 этап:




SmartDataObject будет извлекать данные из всех трех таблиц базы данных, а SmartDataBrowser позволит увидеть эти данные и отредактировать оценки. SmartFilter даст возможность пользователю наложить фильтр на запрос к базе. Этот объект можно использовать в стандартном виде, предусмотренном разработчиками Progress, настройка его на конкретный запрос производится во время сборки экрана через окно свойств экземпляра объекта.

2 этап:


1. Создадим объект типа SmartDataObject по всем полям всех трех таблиц:




- через кнопку  откроем список объектов и выберем в нем SmartDataObject;
- следуя указаниям Wizard определим query по таблицам student, marks, course;
- далее выберем все поля этих таблиц (кнопка "Add Fields") кроме "дублирующих" mark.num_st и course.code (в данном примере не все из них нужны, но мы хотим создать здесь относительно универсальный объект); по умолчанию все поля могут быть редактируемыми – оставим это без изменений;
- сохраним новый объект в директории c:\examples\part10 под именем d_st_mr_cr.w.

2. Создадим объект типа SmartDataBrowser:

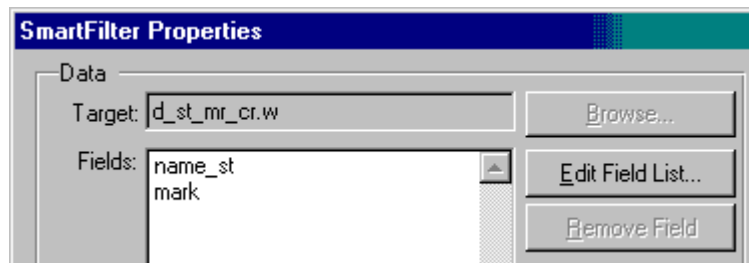
- через  выберем SmartDataBrowser;
- с помощью Wizard определим для него в качестве источника данных объект d_st_mr_cr, созданный в предыдущем примере;
- на следующем экране нажмем кнопку Add Fields и выберем поля name_st, mark и name_c; войдет в свойства Browse и поле mark отметим как редактируемое.
- сохраним SmartDataBrowser в директории c:\examples\part10 под именем b_st_mr_cr.w.

3 этап:

1. Через  создадим объект типа SmartWindow, разместим в нем:

- объект d_st_mr_cr.w ();
- объект b_st_mr_cr.w () , согласимся на создание связи типа Data от d_st_mr_cr и на создание связи типа Update к d_st_mr_cr;
- объект SmartFilter () , согласимся на создание связи типа Filter к d_st_mr_cr;

2. Войдем в Pop-up меню экземпляра (например, щелчком мыши по “вентилятору” объекта), выберем пункт Instance Properties и в окне свойств назначим поля, по которым будет производиться фильтрация – name_st и mark:

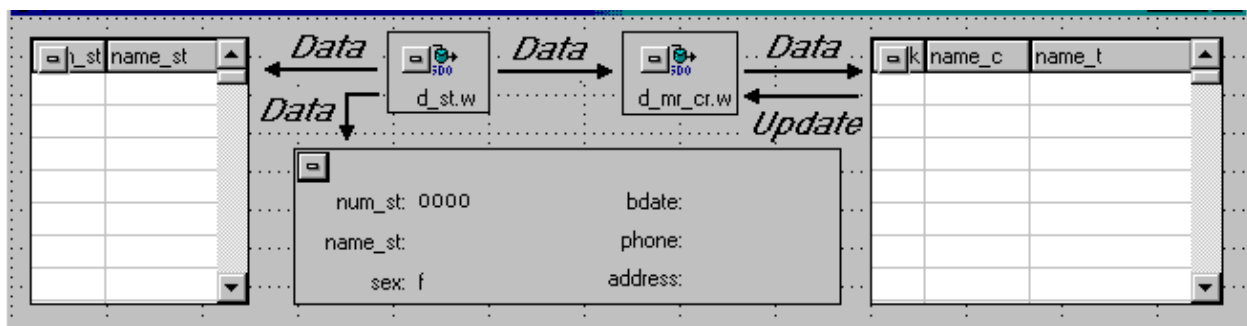


3. Сохраним окно под именем i10_0203.w. Выполним приложение.

Для реализации связанного запроса можно использовать либо один источник данных (SmartDataObject), как в предыдущем примере, либо несколько отдельных источников. Во втором случае связь между ними определяется только на этапе сборки приложения, вне его эти источники могут использоваться автономно.

Пример 10.2.4 (AB)


Просмотр информации о выбранном студенте и редактирование его оценок (этот пример можно рассматривать как порождение нового master из уже существующего - SmartWindow i10_0202.w).

1 этап:


Здесь используются два объекта типа SmartDataObject. Первый (d_st) достаёт записи из таблицы студент, второй (d_mr_cr) – из таблиц marks и course. Эти объекты нужно будет связать, поскольку второй должен будет извлекать из базы не все записи таблицы marks, как это будет определено при создании объекта, а только связанные с текущей записью из таблицы student.

2 этап:

1. Создадим объект типа SmartDataObject, предназначенный для выборки данных из таблиц marks и course:


- через кнопку  откроем список объектов и выберем в нем SmartDataObject;
- следуя указаниям Wizard определим query (кнопка "Define query") по всем записям таблиц marks и course;
- далее выберем поля (кнопка "Add Fields") – выберем все поля таблиц marks и course; уберем отметку о редактируемости (Updateable) у всех полей кроме поля mark;
- сохраним новый объект в директории c:\examples\part10 под именем d_mr_cr.w.

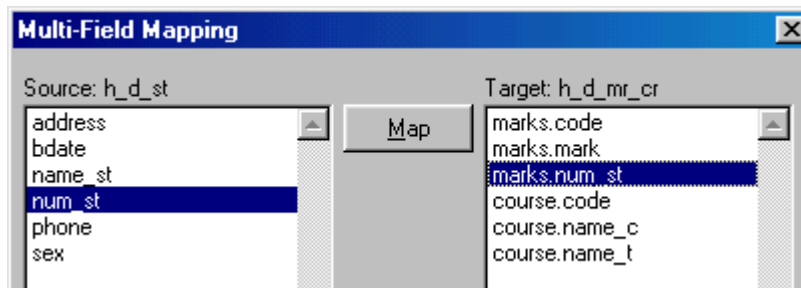
2. Создадим объект типа SmartDataBrowser:

- через  выберем SmartDataBrowser;
- с помощью Wizard определим для него в качестве источника данных объект d_mr_cr, созданный в предыдущем пункте;
- на следующем экране нажмем кнопку Add Fields и выберем поля mark, name_c и name_t;
- сохраним SmartDataBrowser в директории c:\examples\part10 под именем b_mr_cr.w.

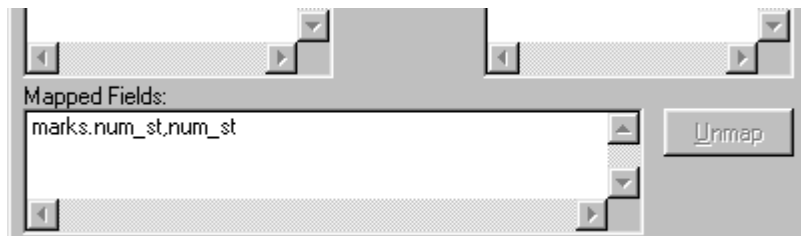
3 этап:

1. Откроем приложение i10_0202.w и добавим в него новые объекты:


- d_mr_cr.w () - согласимся на создание связи типа DATA от d_st; затем согласимся на предложение описать эту связь подробнее - "Choose. Specify the Foreign fields" - и в следующем окне укажем поля, по которым будут связываться таблицы student (в запросе из d_st) и marks (в запросе из d_mr_cr):



а затем нажатием кнопки "Map" перенесем их в область Mapped Fields внизу экрана:



2. Следующий вопрос будет о создании связи типа Update от старого объекта b_st ко вновь вставленному объекту. Откажемся от этой связи.

- вставим в окно объект b_mr_cr () , согласимся на создание связи типа Data от d_mr_cr и на создание связи типа Update к d_mr_cr.

3. Сохраним измененное окно под именем i10_0204.w – как новое приложение. Выполним приложение.

Среди стандартных панелей, подготовленных разработчиками Progress, имеется панель, предназначенная для создания, редактирования и удаления записей. Эта панель функционирует в тесном взаимодействии с объектом SmartDataViewer через связь вида TableIO.

Следует иметь в виду, что механизм редактирования записей с помощью стандартной панели редактирования требует особых приемов работы с полями, на создание, изменение или удаление которых есть триггеры. Например, если в базе данных есть триггер на создание записи, в котором уникальному полю присваивается значение из секвенции, этот триггер будет игнорирован. Если в базе данных Univ для поддержки автоматического создания номеров студентов и кодов предметов имеются соответствующие триггеры, следует закрывать для редактирования или совсем убирать

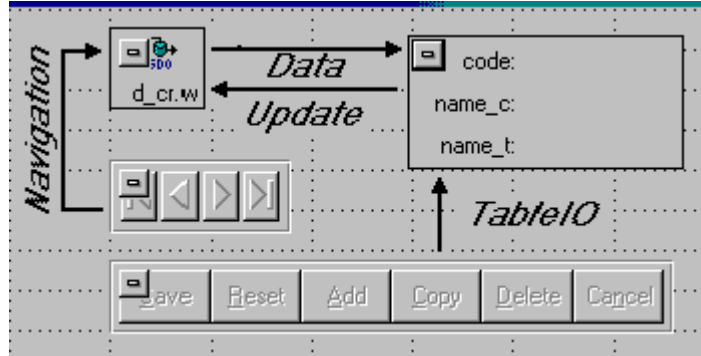
поля student.num-st и course.code из SmartDataViewers, с которыми работают стандартные панели редактирования.

В следующем примере поле code таблицы course будет совсем закрыто для редактирования.

Пример 10.2.5 (AB)

Редактирование информации о предметах с помощью стандартной панели редактирования.


1 этап:




Новый объект – панель для редактирования, должна быть связана с объектом, в котором непосредственно происходит редактирование данных, связью типа TableIO. Информация об измененных в SmartDataViewer данных будет возвращаться в базу через SmartDataObject по связи типа Update.

2 этап:

1. Создадим объект типа SmartDataObject для работы с таблицей course:





- через кнопку  откроем список объектов и выберем в нем SmartDataObject;
- следуя указаниям Wizard назначим в качестве query для него запрос по таблице course;
- выберем все поля таблицы course;
- сохраним под именем d_cr.w в директории c:\examples\part10.

2. Создадим объект типа SmartDataViewer, предназначенный для вывода на экран и редактирования данных из таблицы course:

- через  выберем SmartDataViewer;
- в качестве источника данных для него (SmartDataObject pathname) укажем созданный в предыдущем пункте d_cr.w и выберем все поля;
- войдем в окно свойств поля code и уберем флажок Enable.
- сохраним объект под именем v_cr.w.

3 этап:

1. Создадим объект типа SmartWindow, разместим в нем:

- d_cr.w ()
- SmartPanel () - навигационная панель nnavico.r (согласимся на создание связи типа Navigation к d_cr);
- v_cr.w () - согласимся на создание связи типа Data от d_cr;
- затем Advisor спросит о необходимости создания связи типа Update к d_cr – согласимся;
- SmartPanel () - панель редактирования rupdsav.r (согласимся на создание связи типа TableIO к v_cr).

2. Сохраним окно под именем i10_0205.w. Выполним приложение.

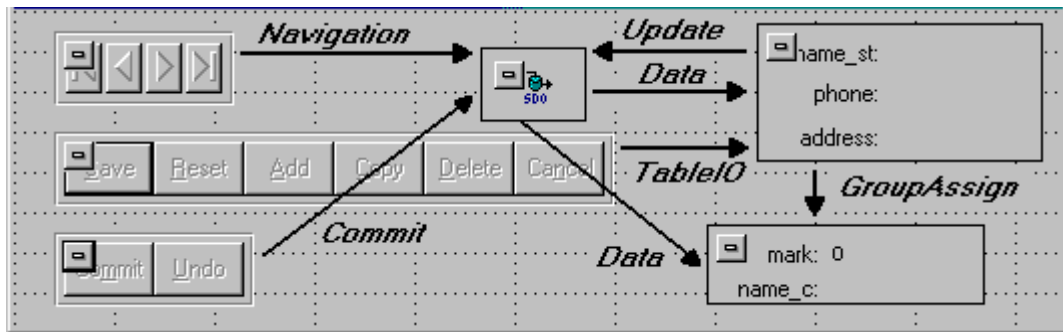
Самым простым способом редактирования является работа с каждой таблицей по отдельности. Возможно и одновременное редактирование данных более чем из одной таблицы (связных записей),

поставляемых одним источником данных. Такое редактирование обычно осуществляется через объекты SmartDataViewer (один обобщенный или несколько, соединенных связями GroupAssign). В этом случае можно пользоваться одной панелью редактирования и одной транзакционной панелью.

Пример 10.2.6 (AB)

Редактирование информации о некоторых полях из разных таблиц через одну панель редактирования; использование стандартной транзакционной панели.

1 этап:




В этом примере два объекта SmartDataViewer получают данные из одного источника. Соединенные связью GroupAssign, они редактируются через одну панель редактирования. Транзакционная панель даст возможность пользователю динамически переопределить размеры транзакции.


В качестве источника данных здесь используется SmartDataObject, созданный в одном из предыдущих примеров, а оба объекта SmartDataViewer будут созданы на втором этапе.

2 этап:


1. Создадим объект типа SmartDataViewer, предназначенный для вывода на экран полей name_st, phone и address из таблицы student, причем поле name-st не должно быть редактируемым:




- через  выберем SmartDataViewer;
- в качестве источника данных для него (SmartDataObject pathname) укажем созданный ранее d_st_mr_cr.w, выберем поля name_st, phone и address;
- войдем двойным щелчком в свойства поля name_st и в разделе OtherSettings уберем флажок Enable.
- сохраним под именем v_st1.w.


2. Создадим объект типа SmartDataViewer, предназначенный для вывода на экран полей marks.mark и course.name_c, причем поле name-c не должно быть редактируемым:


- через  выберем SmartDataViewer;
- в качестве источника данных для него (SmartDataObject pathname) укажем созданный ранее d_st_mr_cr.w, выберем поля marks.mark и course.name_c;
- войдем двойным щелчком в свойства поля name_c и в разделе OtherSettings уберем флажок Enable.
- сохраним под именем v_mr_cr.w.


3 этап:

1. Через  создадим объект типа SmartWindow, разместим в нем:

- объект d_st_mr_cr.w ();
- SmartPanel () – навигационная панель pnavico.r (согласимся на создание связи типа Navigation к d_st_mr_cr).
- v_st1.w () - согласимся на создание связи типа Data от d_st_mr_cr и связи типа Update к d_st_mr_cr;

-  `v_mr_cr.w` - согласимся на создание связи типа Data от `d_st_mr_cr` и связи типа GroupAssign от `v_st1`;

-  `SmartPanel` - панель редактирования `rupdsav.r` (или `mypanel`), согласимся на создание связи типа TableIO к `v_st1`.


-  `SmartPanel` - транзакционную панель `pcommit.r` (согласимся на создание связи типа Commit к `d_st_mr_cr`).

2. Сохраним окно под именем `i10_0206.w`. Выполним приложение.

- Заметим, что нежелательно использовать возможности создания и удаления записей через стандартную панель редактирования при работе более чем с одной записью одновременно.

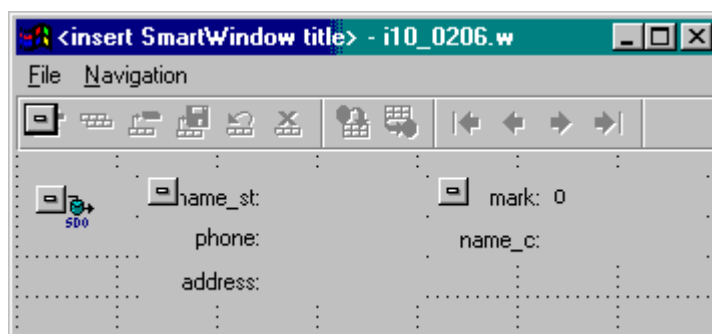
3. Изменим приложение. Вместо трех кнопочных панелей вставим объект `SmartToolBar`, который совмещает функциональность нескольких стандартных панелей:

- удалим панели, передвинем остальные объекты, чтобы освободить место в верхней части окна;

- вставим в окно `SmartToolBar` () , согласимся на создание связей Navigation - к `d_st_mr_cr`, и TableIO - к `v_st1`;

- добавим к стандартным возможностям `SmartToolBar` (навигация и редактирование) средства управления транзакцией; для этого в свойствах экземпляра объекта (popup-меню → `InstanceProperties`) поставим флажок `Commit`;

- установим связь `Commit` от нового объекта (`h_dyntoolbar`) к `d_st_mr_cr` (popup-меню → `SmartLinks` → `Add`).



4. Выполним приложение.

10.3. Использование виртуальных страниц

Каждый `SmartContainer` (`SmartWindow`, `SmartFrame`, `SmartDialog`) может содержать логические (виртуальные) страницы. Логические страницы помогают расширить рабочее пространство.

Основная (физическая) страница контейнера называется `Page0`, виртуальные страницы нумеруются начиная с единицы: `Page1`, `Page2`,... На `Page0` могут размещаться любые объекты и они всегда видны ("просвечивают" через виртуальные страницы). На логических страницах могут быть размещены лишь `SmartObjects` и видны они только тогда, когда активна соответствующая страница - как во время проектирования интерфейса, так и во время исполнения программы.

Существуют также понятия `Startup Page` и `Design Page`. `Startup Page` - это та страница, которая будет видна при активизации контейнера во время исполнения программы. По умолчанию - это физическая страница (`Page0`). Понятие `Design Page` имеет смысл только во время проектирования - это текущая страница. По умолчанию это также `Page0`. Информация о том, какая страница сейчас является текущей, находится в поле `Page Number` информационной строки управляющего окна `ApplicationBuilder`. Изменить текущую страницу можно в окне `Goto Dialog Box`, которое появляется после двойного щелчка мышью по полю `Page Number` или при выборе пункта меню `Edit` → `Goto Page`....

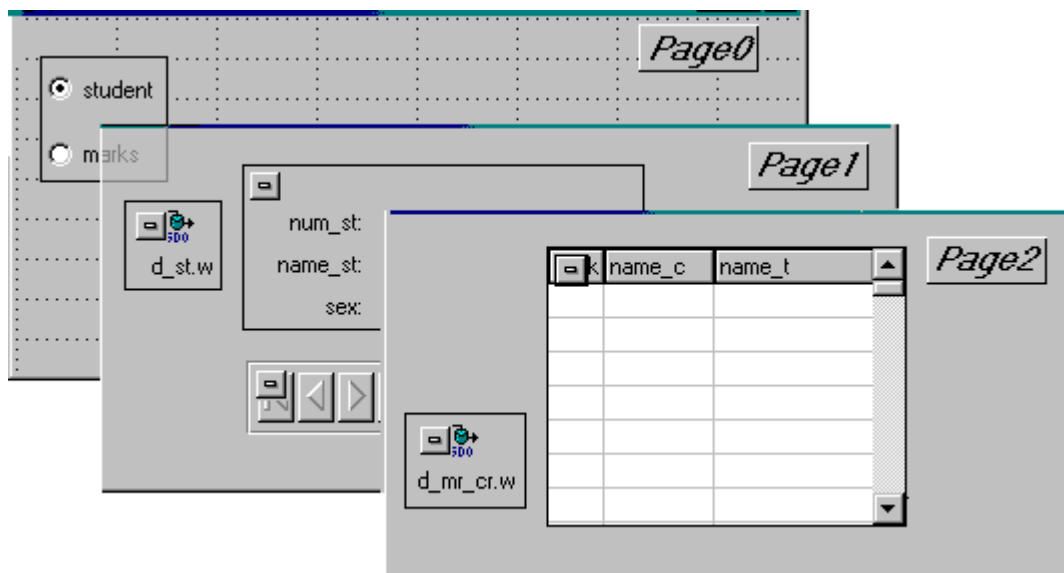
Полная информация о страницах активного контейнера содержится в окне `Pages Dialog Box`, которое можно вызвать, выбрав в управляющем окне пиктограмму `Procedure settings` (или `Tools` → `Procedure`

Settings), а затем щелкнув по кнопке Pages (пиктограмма). Здесь же можно переназначать Startup Page, удалять и перемещать содержимое страниц.

Управление страницами во время исполнения приложения может осуществляться различными программными способами. Следующие примеры демонстрируют некоторые из этих приемов.

Пример 10.3.1 (AB)

Управление страницами SmartWindow с помощью метода SelectPage.



1. Создадим объект типа SmartWindow, разместим в нем обычный RADIO-SET. В качестве списка значений (через Property) выберем 1 и 2, и назовем метки "student" и "marks". Триггер для radio-set будет создан в пункте 4.

2. Перейдем на страницу Page1 (двойной щелчок мышью по информационному полю Page в управляющем окне) и вставим в нее созданные ранее объекты d_st.w и v_st.w (согласимся на связь типа Data), а также стандартную навигационную панель:

3. Перейдем на страницу Page2 и вставим в нее созданные ранее объекты d_mr_cr.w и b_mr_cr.w (согласимся на создание всех предложенных связей – аналогично примеру 10.2.3):

4. Создадим для RADIO-SET триггер на событие VALUE-CHANGED:

```
ASSIGN {&SELF-NAME}.
RUN SelectPage({&SELF-NAME}).
```

5. Назначим в качестве Startup Page страницу Page1:

Procedure Settings → Pages (пиктограмма) → кнопка Start.

6. Выполним приложение. Сохраним окно под именем i10_0301.w.

Еще один способ управления страницами - использование SmartFolder. Этот объект предназначен для естественного и наглядного передвижения между страницами во время исполнения приложения. Каждой "закладке" SmartFolder соответствует одна виртуальная страница, начиная с Page1. При этом сам SmartFolder должен размещаться на физической странице. В отличие от других SmartObjects, при работе со SmartFolder не создается master (хотя, вообще говоря, это возможно). Smart Folder достаточно прост и гибок для того чтобы встраиваться в интерфейс во время исполнения прямо из главного шаблона. Поскольку вызов метода SelectPage "зашит" внутри этого объекта, при использовании SmartFolder нет надобности писать что-либо об управлении страницами, как в предыдущем примере. Привязка SmartFolder к страницам контейнера происходит в момент встраивания его в интерфейс - создаются связи типа PAGEn.

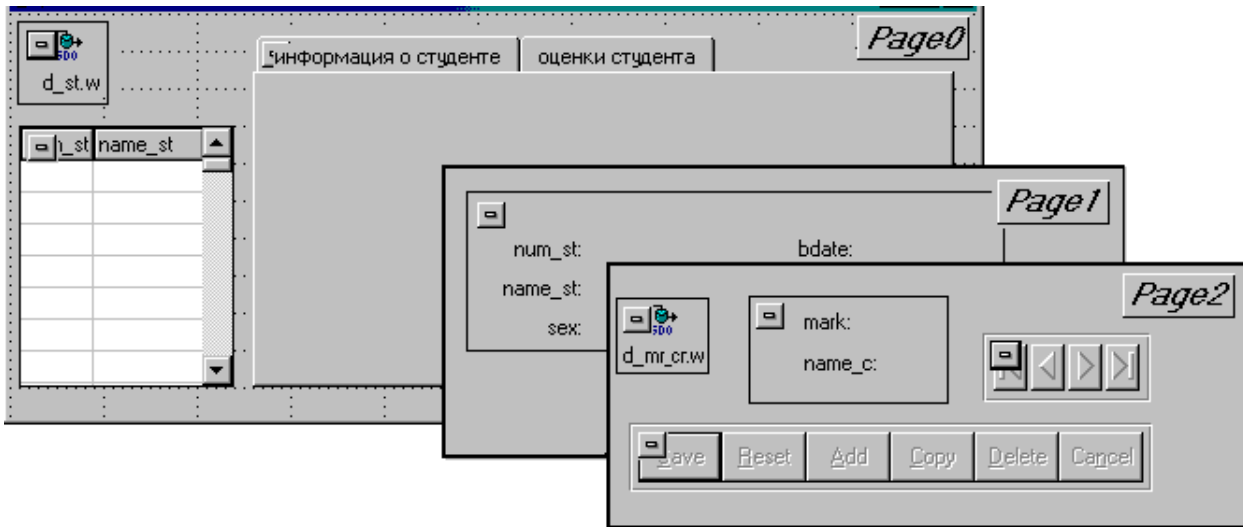
Пример 10.3.2 (AB)

Управление страницами с помощью объекта SmartFolder.

1. Создадим объект типа SmartWindow, разместим на его физической странице созданные ранее объекты d_st и b_st.w, согласимся на установление связи между ними.

2. Вставим на эту же страницу объект SmartFolder (согласимся на создание связи типа PAGE со SmartWindow). Щелчком правой клавиши мыши по SmartFolder войдем в Popup menu, выберем пункт "Instance Properties" и создадим две "закладки":

- 1 информация о студенте
- 2 оценки студента



3. Перейдем на Page1 (двойной щелчок мышью по информационному полю о страницах в управляющем окне). Все ранее вставленные в окно объекты при этом останутся на виду, так как они находятся на физической странице. Разместим на Page1 ранее созданный объект v_st.w (согласимся на создание связи типа Data от d_st).

4. Перейдем на Page2 и вставим в нее созданный ранее d_mr_cr.w (при запросе о связи согласимся на связь типа Data от d_st.w, назначим в качестве Foreign Field num-st и откажемся от связи Update от b_st, которую будет предлагать Advisor). Разместим рядом панель pnavico (связь типа Navigation к d_mr.w), объект v_mr_cr.w (связи типа Data от d_mr_cr.w и Update к d_mr_cr.w) и панель pupdsav (связь типа TableIO к v_mr_cr.w)

5. Сохраним окно под именем i10_0302.w. Выполним приложение.

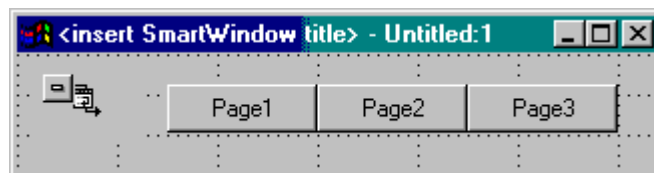
Следующий пример демонстрирует еще один способ управления страницами - с помощью метода ViewPage, примененного к BUTTONs. Этот метод отличается от метода SelectPage главным образом тем, что активизируя очередную страницу, он "не прячет" предыдущую. Этот метод удобен для вызова различных SmartContainers из главного окна.

Пример 10.3.3 (AB)

1. Создать объект типа SmartWindow, разместим в нем три кнопки для управления страницами следующим образом: на палитре объектов щелкнем правой клавишей мыши по BUTTON; из открывшегося меню выберем ViewPage.

2. Изменим метки кнопок (например, Page1, Page2 и Page3). Изменим для второй и третьей кнопок триггеры на событие CHOOSE - в качестве параметра метода ViewPage подставим вместо 1 соответственно, 2 и 3 (первая кнопка будет работать с Page1, вторая - с Page2, третья - с Page3).

3. Перейдем на Page 1. Вставим в нее созданное ранее окно i10_0201.w:



Page1

4. Перейдем на Page2. Вставим в нее созданное ранее окно i10_0202.w (откажемся от всех предложений Advisor по поводу связей).

5. Перейдем на Page3. Вставим в нее созданное ранее окно i10_0203.w (откажемся от всех предложений Advisor по поводу связей).

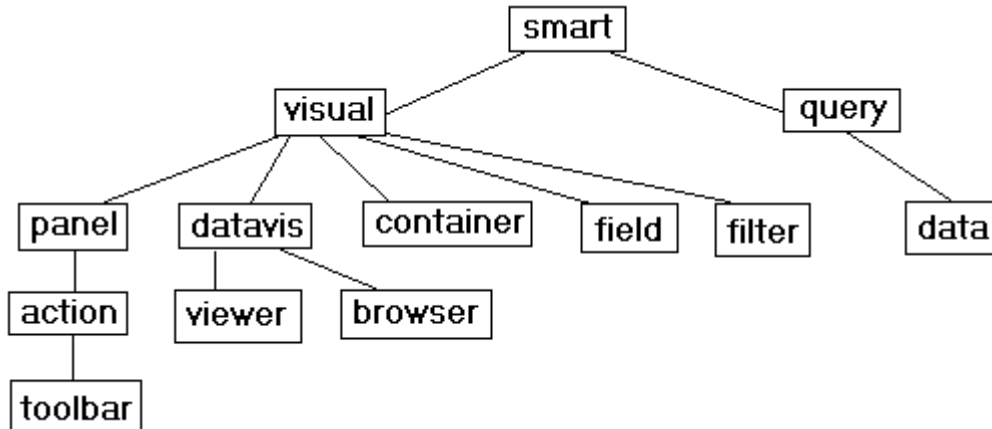
6. Сохраним окно под именем i10_0303.w. Выполним приложение.

11. АРХИТЕКТУРА ADM

11.1. Структура SmartObjects

Архитектура ADM базируется на нескольких основных понятиях: иерархия классов объектов, система связей между объектами, средства управления свойствами объектов и механизмы вызовов процедур.

Иерархия базовых классов Progress представляет следующей схемой:



Программисты в процессе разработки приложений могут создавать дополнительные базовые классы, но основная часть их работы приходится на порождение пользовательских объектов из базовых и сборка приложений из этих пользовательских объектов.

Smart-объекты в приложениях активно взаимодействуют, как через систему связей между ними, так и другими способами. Любое взаимодействие обеспечивается некоторыми протоколами, основанными на механизмах запуска чужих внутренних процедур и функций, публикации событий и подписки на них, установки и получения атрибутов ADM.

Любой SmartObject имеет некоторую стандартную структуру, позволяющую реализовать эти протоколы, и состоит из процедурных файлов и include-файлов, которые хранятся в системных директориях Progress.

Каждому базовому классу объектов соответствует основной файл (процедура) с расширением .w (например, viewer.w). Этот модуль содержит минимум выполнимого кода. Когда программист порождает новый пользовательский объект, происходит копирование именно этого файла.

Базовый модуль включает в себя одноименный include-файл поддержки (например, viewer.w содержит {src/adm2/viewer.i}). В include-файле нет кода, который бы напрямую поддерживал SmartObject. Здесь находятся только те элементы, которые требуются для компиляции порожденных объектов: переменные и другие определения, необходимые во всех объектах, ссылка на следующий include-файл в цепочке включений (например, SmartDataViewer, как видимый объект, включает файл datavis.i, а тот - smart.i), свои собственные атрибуты, код для инициализации в Main Block и т.д.

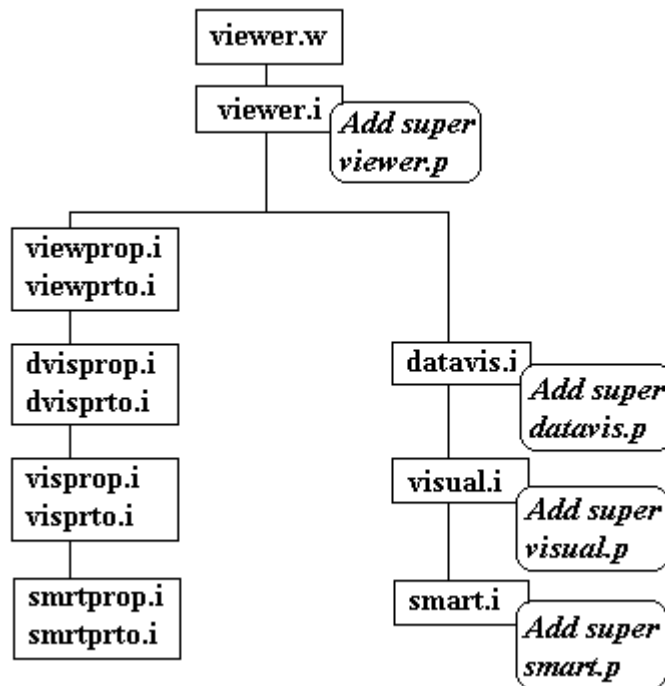
Большая часть кода для поддержки базового объекта хранится в отдельной процедуре, построенной как структурированный .p файл (например, src/adm2/viewer.p). Большинство внутренних процедур в этом файле не предназначены для прямого запуска. В начале сессии .p запускается как super-процедура для каждого из объектов, стартующих в этой сессии.

- Механизм super-процедур в Progress существует для того, чтобы динамически менять тела внутренних процедур и функций активных внешних процедур. С помощью метода ADD-SUPER-PROCEDURE процедура может объявить некоторую другую процедуру (лежащую сейчас в памяти как persistent) super-процедурой для себя или для текущей сессии Progress (и выполнить обратное действие, используя метод REMOVE-SUPER-PROCEDURE). Если текущая процедура имеет внутреннюю процедуру (или пользовательскую функцию) и настал момент ее вызова, Progress находит и выполняет одноименную процедуру (функцию) в super-процедуре, используя определенные правила поиска.

После старта super-процедур все объекты сессии пользуются кодом, хранящимся в их внутренних процедурах. Этот код включает типичные для объектов процедуры (такие как displayFields для SmartDataViewer), функции работы с атрибутами и прочее.

Последние два вида файлов, входящих в состав SmartObject - это include-файлы атрибутов и прототипов (например, viewprop.i и viewprto.i). В них описываются атрибуты объекта и точки входа во внутренние процедуры.

Ниже в качестве примера приведена схема структуры SmartDataViewer:



Файлы с исходным кодом, лежащим в основе базовых SmartObjects, размещены в поддиректориях Progress/src: adm2; adm2/template и adm2/support. Те из них, что представляют собой с точки зрения Progress 4GL процедуры, имеют откомпилированные варианты, хранящиеся в одноименных директориях под Progress/gui.

В директориях с именами custom под src и gui находятся файлы, играющие роль вспомогательных, и предназначенные для внесения разработчиком своих изменений и дополнений к базовым объектам.

11.2. Взаимодействие SmartObjects

Взаимодействие SmartObjects обеспечивается через:

- выполнение внутренних процедур и функций;
- публикацию именованных событий и подписку на них (publishing/subscribing);
- получение и установку свойств объектов (ADM properties).

Многие особенности поведения SmartObjects определяются **вызовом внутренних процедур и функций**. Обычно это происходит через использование super-процедур, которые поддерживают SmartObjects. Например, для установки связи (link) от одного SmartObject к другому в программе можно использовать процедуру addLink:

```
RUN addLink ( h-SourceProc, 'Navigation', h-TargetProc ).
```

где h-SourceProc - HANDLE источника,
h-TargetProc - HANDLE получателя.

Оператор RUN написан так, будто addLink является internal procedure текущей процедуры. Фактически же addLink содержится в super-процедуре smart.p. Механизм super-процедур делает содержимое smart.p доступным для каждого SmartObject. В этой же super-процедуре содержатся такие стандартные для всех SmartObjects процедуры как:

```

initializeObject  – запускается при старте объекта;
destroyObject    – запускается при разрушении объекта;
addMessage       – запускается для добавления error message в message log.
  
```

Подобным образом все видимые объекты используют super-процедуру visual.p, а все объекты-контейнеры - containr.p.

Помимо запуска процедур, хранящихся в “родных” super-процедурах, SmartObject может обращаться в другой SmartObject, зная его procedure handle, и запускать internal procedures или functions в этом SmartObject. Например, один SmartObject разрушает другой выполнением следующего оператора:

RUN destroyObject IN h-OtherObject.

Механизм вызова процедур позволяет также создавать пользовательские версии процедур и функций, подменяющие стандартные для конкретных SmartObject. В этом случае программист пишет в Section Editor для создаваемого объекта свою одноименную процедуру или функцию, которая и будет выполняться локально для данного объекта.

Механизм публикации именованных событий и подписки на них (publishing/subscribing) позволяет связанным объектам влиять друг на друга без знания handles.

Каждый вид SmartLink характеризуется набором именованных событий (named events). Во время создания SmartLink источник (Source) и получатель (Target) автоматически подписываются (subscribed) на именованные события в связанной procedure. Например, Navigation-Target (SmartDataObject) подписывается на события fetchFirst, fetchNext, fetchPrev и fetchLast в Navigation-Source (SmartPanel). Во время работы приложения в ответ на нажатие пользователем кнопки Next объект SmartPanel выполняет следующий оператор:

PUBLISH 'fetchNext'.

Как видно по синтаксису, не требуется, чтобы SmartPanel знала адрес Navigation-Targets. Этот адрес известен среде ADM после выполнения оператора SUBSCRIBE, который отработал в процедуре addLink от имени объекта-получателя. Процедура addLink была выполнена, когда объект инициализировался.

Стандартный код ADM содержит много таких операторов PUBLISH. Программист также может использовать этот оператор при создании нового типа связи. Когда встречается оператор RUN addLink(h1, 'newlink', h2), процедура addLink выполняет оператор SUBSCRIBE PROCEDURE h2 TO 'newlink' IN h1. И программист должен предусмотреть операторы PUBLISH в процедуре h1 во всех случаях, когда нужно вызвать именованное событие в h2: PUBLISH 'newlink'.

Каждый SmartObject знает, на какие named events для каждого вида link он автоматически должен подписаться в процедуре addLink. Эта информация хранится в таких атрибутах объекта как linktypeSourceEvents и linktypeTargetEvents. Следующая таблица показывает значения этих атрибутов для стандартных видов link. Для пользовательских типов связей программист сам задает значения атрибутов SourceEvents и TargetEvents.

Link Type	SourceEvents	TargetEvents
Commit	CommitTransaction UndoTransaction	RowObjectState
Container	InitializeObject HideObject ViewObject DestroyObject EnableObject ConfirmExit	ExitObject
Data	DataAvailable QueryPosition	UpdateState
GroupAssign	AddRecord CopyRecord UpdateRecord ResetRecord CancelRecord EnableFields DisableFields CollectChanges	UpdateState
Navigation	FetchFirst FetchNext FetchPrev FetchLast	queryPosition updateState linkState
Page		changeFolderPage deleteFolderPage

Link Type	SourceEvents	TargetEvents
TableIO	addRecord updateRecord copyRecord deleteRecord resetRecord cancelRecord updateMode	queryPosition updateState linkState

Одно из основных требований к SmartObjects – доступность атрибутов объекта. Внутри Progress-сессии не практикуется использование SHARED переменных или буферов для разделения данных между объектами. SmartObjects запускаются как persistent procedures, которые равноправны и не имеют фиксированной последовательности выполнения; поэтому SHARED механизм не очень подходит.

ADM поддерживает взаимодействие между объектами через **получение и установку атрибутов (ADM properties)** посредством двух механизмов, которые имеют общий синтаксис.

Первый механизм - это использование функции "getPropname" для получения значений атрибутов и функции "setPropname" для их назначения (например, getBgColor). Для "setPropname" входным параметром является значение атрибута любого Progress-типа, а возвращается логическое значение true или false, в зависимости от успеха выполнения. У "getPropname" нет входных параметров, а возвращается значение параметра какого-либо типа. Атрибут может иметь характеристику read-only, и тогда функция, "set" не применима.

Атрибуты могут храниться как локальные переменные, или быть чем-либо другим, что может управляться функциями set и get. Функции set/get могут, кроме того, выполнять и какие-либо вспомогательные действия, например, проверять допустимость значений атрибутов.

Таким образом, функции get и set подходят для работы из одного объекта с любыми атрибутами другого объекта. Однако существует более эффективный доступ к атрибутам (второй механизм). Он предлагает set и get, через которые можно получить быстрый доступ к базовым атрибутам объекта (basic properties). Эти атрибуты хранятся во временной таблице ADMProps, известной каждому объекту.

Super-процедуры объектов включают include-файлы, которые описывают их базовые атрибуты, и каждому объекту, использующему эти super-процедуры, также известен этот список атрибутов. Например, super-процедура smart.p содержит функции и процедуры, которые используются всеми SmartObjects. Это верхушка иерархии super-процедур. Процедура smart.p включает файл smrtprop.i, который описывает те базовые атрибуты, которые применимы для всех SmartObjects. Для каждого базового атрибута определена препроцессорная константа (xpPropname) и описано поле во временной таблице ADMProps. В качестве базовых здесь фигурируют такие атрибуты, как ObjectType, PropertyDialog, QueryObject, ContainerHandle, InstanceProperties, SupportedLinks, DataSource и т.д.

Каждый тип SmartObjects дополняет этот список базовых атрибутов. Например, SmartDataObject объявляет smart.p как первую свою super-процедуру, а затем добавляет data.p, которая описывает поведение, характерное для SmartDataObjects. Процедура data.p включает файл datarprop.i, который описывает дополнительные базовые атрибуты, добавляя препроцессорные имена и поля в таблицу ADMProps.

Если необходимо выполнять get или set через property function (чтобы помимо получения или установки атрибута совершать какие-либо дополнительные действия), препроцессорные константы не определяются, и в этом случае всегда выполняется соответствующая функция. Например, атрибут QueryPosition содержит информацию о позиции курсора в query ('FirstRecord', 'LastRecord' и т.д.). Дополнительно к простому хранению этого описательного значения, при переустановке атрибута должен быть выполнен оператор PUBLISH (чтобы другие объекты могли отреагировать). Поэтому препроцессорная константа xpQueryPosition не определена.

Чтобы сделать доступ к значениям атрибутов как можно более простым и не вспоминать о том, является ли атрибут базовым или нужен доступ через его get и set functions, возможно использование специального синтаксиса (pseudo-syntax) через включение в код include-файлов с именами get и set (хранятся в директориях gui и tty и не имеют стандартного расширения .i из соображений мнемоничности синтаксиса включения). Использование этих файлов имеет следующий вид:

```
{get propname target-variable [ object-handle ] }.
```

При проверке синтаксиса система смотрит, определено ли препроцессорное имя "xpPropname". Если да, то берется значение атрибута из таблицы ADMProps в target-procedure (или object-handle), и помещается в target-variable. Если xpPropname не определено, выполняется функция getPropname в указанном объекте. Аналогично для set.

Например:

```
[ Stat = ] {get DataColumns cColumns}.
```

Если DataColumns – один из базовых атрибутов, хранимых в ADMProps, генерируется следующий 4GL код:

```
ASSIGN
    ghProp = WIDGET-HANDLE(ENTRY(1, TARGET-PROCEDURE:ADM-DATA, CHR(1)))
    ghProp = ghProp:BUFFER-FIELD('DataColumns')
    cColumns = ghProp:BUFFER-VALUE.
```

Если DataColumns не является базовым атрибутом, тогда породится оператор:

```
cColumns = dynamic-function("getDataColumns":U IN TARGET-PROCEDURE).
```

Переменная ghProp описана в smart.i. Указатель на буфер записи таблицы ADMProps известен каждому объекту, поэтому может использоваться в этом include-файле. В любом случае include-файл порождает один выполнимый Progress-оператор, делая это одинаково быстро независимо от наличия данного атрибута в списке базовых.

При создании нового базового класса SmartObject следует продумать список базовых атрибутов, хранимых в ADMProps (сюда нужно отнести все атрибуты, к которым будут часто обращаться) и построить include-файл атрибутов для класса подобно тому, как был создан viewprgor.i для SmartDataViewer.

Каждый тип объекта имеет определенный набор атрибутов, которые уместно назначать во время инициализации объекта, при сборке объектов в Smart-контейнеры. Например, атрибуты HideOnInit, DisableOnInit и ObjectLayout для видимых объектов показывают, должен ли конкретный экземпляр (instance) этого объекта быть видимым и активным, когда он впервые реализуется, и какой из возможных визуальных шаблонов следует использовать в данном случае. Такие атрибуты называются атрибутами экземпляра объекта (Instance Properties). Кроме упомянутых атрибутов, назначаемых при инициализации объекта, могут быть и другие, которые устанавливаются в ходе выполнения приложения, так как не имеют определенного "initial" значения.

В каждом include-файле атрибутов объекта список Instance Properties может быть определен через препроцессорные значения xcInstanceProperties. Для каждого типа объекта Instance Properties инициализации поддерживаются процедурой Instance Property Dialog (известной системе через preprocessor-name ADM-PROPERTY-DIALOG), которая хранится в отдельном файле и может быть изменена разработчиком. Значения Instance Properties вручную устанавливаются программистом во время сборки приложения через окно Instance Properties и присваиваются в коде, генерируемом AppBuilder'ом в процедурах adm-create-objects и constructObject.

Далее приведены некоторые полезные функции работы с атрибутами:

Функция **assignLinkProperty** – устанавливает значение property в одном или более objects на другом конце указанного link.

Функция **linkProperty** – возвращает значение указанного property в одном объекте на другом конце указанного link.

Функция **propertyType** - возвращает datatype указанного property.

Особо следует выделить две функции - setUserProperty и getUserProperty, которые могут быть использованы для поддержки динамических атрибутов (dynamic properties) в SmartObjects.

Функция **setUserProperty** устанавливает именованные атрибуты:

```
setUserProperty RETURNS LOGICAL
    ( pcPropName AS CHARACTER, pcPropValue AS CHARACTER ) :
```

Если property name не известно в текущем объекте, создается поле в ADMProps и устанавливается значение атрибута. Если имя известно, просто устанавливается новое значение.

Функция **getUserProperty** возвращает значение именованного атрибута, предварительно назначенного с помощью setUserProperty:

```
getUserProperty RETURNS CHARACTER
    ( pcPropName AS CHARACTER ) :
```

- В случаях, когда программисту нужно отследить во время работы приложения внутреннюю кухню среды ADM, можно воспользоваться инструментами **PRO*Spy** и **Procedure Object Viewer**, которые вызываются через панель PRO*Tools (кнопки PRO*Spy и Procedures). PRO*Spy дает возможность стартовать ADM-приложение и трассировать его. Procedure Object Viewer позволяет отследить все persistent-процедуры, выполняющиеся в текущей сессии, внутренние процедуры и их параметры, super-процедуры и ссылки на базы данных.
- **Progress Debugger** также может быть использован для отладки ADM-приложений. Например, для трассирования super-процедуры можно в код SmartObject перед запуском процедуры вставить инициализацию отладки (DEBUGGER:INITIATE(). DEBUGGER:SET-BREAK().) и тогда во время выполнения стартует Debugger.

11.3. Управление записями в ADM

Большинство SmartObjects предназначено для обработки записей из базы данных. Такие объекты как SmartDataObject, SmartDataViewer и SmartDataBrowser работают непосредственно с данными, другие объекты играют вспомогательную роль:

SmartDataObject определяет запрос, извлекает из базы набор записей, сохраняет его во временном хранилище, замещает измененные данные в базе, поддерживает транзакции.

SmartDataViewer показывает данные из временного хранилища, дает возможность их редактировать.

SmartDataBrowser имеет свой механизм демонстрации и редактирования данных из временного хранилища, навигации по ним.

SmartPanel's и **SmartToolbar** помогают другим объектам перемещать и редактировать записи, управлять размерами транзакций.

SmartFilter дает возможность дополнительной фильтрации данных.

Каждый базовый класс объектов имеет набор "любимых" видов связей. Связи, обслуживающие обработку данных, для большей гибкости построения приложений специализированы, по каждому виду передается своя информация (о значениях очередной строки во временном хранилище данных - Data, о перемещении указателя в запросе - Navigation, о состоянии и режимах редактирования - TableIO, о необходимости проверки и замещения измененных данных - Update, о старте и завершении транзакций - Commit).

Функциональная дифференциация объектов и связей помогает:

- унифицировать операции обработки данных (разделить процесс управления данными на стандартные части);
 - отделить пользовательский интерфейс от управления данными (например, объекты визуализации могут выполняться в сессии без database connection);
 - повысить универсальность объектов (так, один и тот же SmartDataObject может быть использован в одном месте приложения для работы со всеми записями таблицы, а в другом – для выборки только тех записей, которые совпадают по ключевому значению с записью связной таблицы).

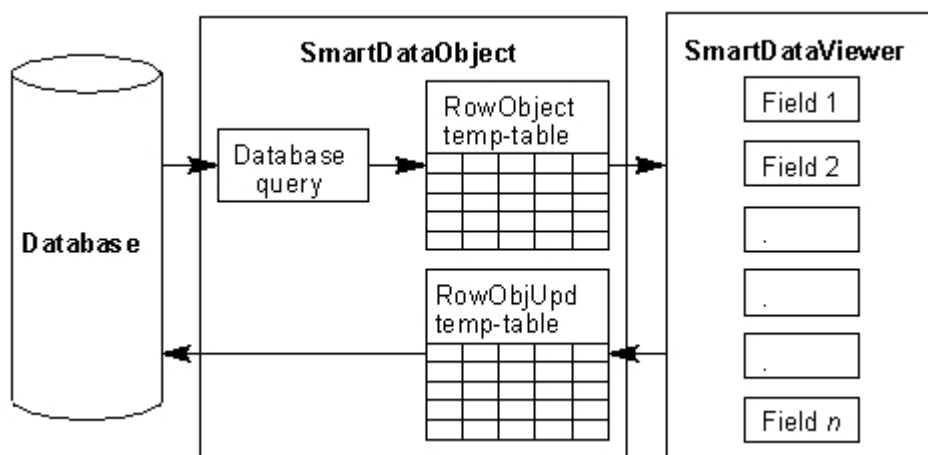
Ключевую роль в управлении данными играет объект SmartDataObject. Он всегда содержит внутренний запрос к базе данных и переводит результаты этого запроса во временную таблицу RowObject, что позволяет изолировать объекты визуализации данных от базы. При создании SmartDataObject можно выбирать поля из базы, менять их имена, формат и прочие атрибуты, назначать редактируемость полей. Можно использовать вычисляемые поля для отображения действий над другими полями в таблице RowObject, или в качестве места для хранения информации, которой SmartDataObject обменивается с другими объектами.

Объекты, которые работают с базой через SmartDataObject, видят только колонки во временной таблице, получают оттуда данные и возвращают измененные значения и новые строки также во временную таблицу. SmartDataObject поддерживает две временные таблицы - RowObject и RowObjUpd. Ниже приведена упрощенная схема перемещения данных через SmartDataObject.

Структура таблицы RowObject включает в себя поля с данными и три служебных поля: RowNum - порядковый номер строки в запросе, RowIdent - список rowid записей строки, RowMod - режим редактирования строки.

Таблица RowObjUpd по своей структуре является копией таблицы RowObject плюс дополнительное поле ChangedFields - список измененных в данной строке полей.

Помимо хранения и передачи новых данных, SmartDataObjects может также поддерживать проверку корректности измененных данных (validation logic) в дополнение к аналогичному контролю в Data Dictionary и триггерах.



Обычно запрос из SmartDataObject к базе открывается во время инициализации объекта, так как его атрибут OpenOnInit по умолчанию имеет значение true. Когда SmartDataObject перемещает данные из базы в RowObject, берется слепок состояния этих данных на некоторый момент времени. Таблица RowObject не обязана динамически отображать изменения, вносимые в базу другими пользователями. Обновлять содержимое RowObject можно в любом месте кода с помощью функций openQuery (весь запрос) или refreshRow (текущая строка).

Для динамического изменения запроса перед его обновлением полезными могут быть функции, модифицирующие условия выборки и сортировки запроса:

assignQuerySelection()	addQueryWhere(),
removeQuerySelection()	columnQuerySelection(),
setQueryWhere()	setQuerySort()

Если необходимо более серьезное изменение запроса (такое, как оптимизация построения фразы для связанного query), можно сделать это через атрибут QueryHandle и метод QUERY-PREPARE.

Строки начинают передаваться из database query во временную таблицу RowObject, когда впервые производится операция выборки. Дополнительные строки передаются, когда поступает запрос на следующую порцию информации. Передача строк происходит через процедуру sendRows. Количество строк, передаваемых одновременно, определяется атрибутом RowsToBatch объекта SmartDataObject (по умолчанию - 200). Еще один атрибут - RebuildOnReposition - влияет на алгоритм обновления запроса; при работе с большими объемами данных рекомендуется устанавливать для этого атрибута значение true.

Для позиционирования в таблице RowObject используются процедуры fetchFirst, fetchNext, fetchPrev и fetchLast. Функция fetchRow позволяет репозиционировать данные по определенному номеру строки в запросе, а функция fetchRowIdent - по физическому адресу записи в базе данных.

Сразу после изменения позиции в RowObject функция colValues() возвращает список форматированных значений текущей строки и генерируется событие dataAvailable, на которое подписаны объекты визуализации данных. Эти объекты не видят собственно таблицу RowObject, они получают только список значений полей строки и некоторую служебную информацию о положении этих данных в запросе.

Если объекты визуализации имеют право на редактирование данных (в случаях, когда соответствующие поля определены в SmartDataObject редактируемыми и установлена связь Update) и редактирование действительно происходит, его обработка развивается по определенному сценарию.

Процесс редактирования можно условно разделить на две части - инициализация и собственно редактирование. К первой части относятся событийные процедуры updateRecord, addRecord, copyRecord, deleteRecord, cancelRecord и resetRecord в объектах визуализации, обычно вызываемые из триггеров за кнопками на редактирующей панели. Действительное редактирование данных происходит в SmartDataObject через функции submitRow(), addRow(), copyRow(), deleteRow() и cancelRow(). Именно эти функции вызывают изменения во временных таблицах SmartDataObject:

- изменения данных в объекте визуализации сразу же отражаются в таблице RowObject, а в RowObjUpd фиксируются старые значения данных (они используются для проверки - не изменил ли кто-либо тем временем данные в базе);
- по завершению редактирования новые данные копируются из RowObject в RowObjUpd;
- вновь созданная строка записывается в RowObjUpd;
- при удалении данных строка запроса удаляется из RowObject и сохраняется в RowObjUpd для передачи этой информации на сервер.

Изменение каждой строки запроса автоматически записывается в базу данных через функцию Commit(), если значением атрибута AutoCommit объекта является true (таково его значение по умолчанию, когда к SmartDataObject не привязана транзакционная панель). Иначе эти изменения не будут переданы в базу, пока не придет явное требование Commit от транзакционной панели.

Логика транзакций строится в SmartDataObject с использованием технологии "оптимистичного блокирования". Транзакция не открывается, пока данные вводятся и проверяются. Когда редактирование завершено, соответствующая запись перечитывается с замком EXCLUSIVE-LOCK (при конфликте доступа редактирование будет отменено и пользователь извещен). Затем SmartDataObject проверит, не была ли запись изменена другим пользователем с момента первого прочтения (если разработчик хочет пропустить эту проверку, значение атрибута checkCurrentChanged в SmartDataObject нужно поменять на false). Наконец, те поля, которые были реально изменены, возвращаются в базу данных. Во временной таблице RowObjUpd поле ChangedFields хранит последовательность, в которой модифицировались поля записи. Эту информацию можно использовать, если программист изменяет обработку редактирования (например, если он хочет добавить свою проверку корректности данных). После того как возвращение данных в базу завершено, некоторые из них могут претерпеть дальнейшие изменения в триггерах базы данных. В этом случае новые изменения отразятся у клиента в объектах визуализации.

Для изменения размеров транзакций обычно используется транзакционная панель (pcommit.w). Триггер за кнопкой Commit публикует событие commitTransaction, а триггер за кнопкой Undo - undoTransaction. Панель подписывается на событие RowObjectState в Target-объекте. Процедура RowObjectState принимает в качестве параметра "NoUpdates" или "RowUpdated". Эти состояния управляют активностью кнопок панели.

На этапе сборки приложения разработчик может сформировать достаточно сложную связку объектов для организации большой транзакции. Можно собрать объекты так, что Commit Panel и несколько SmartDataObjects будут связаны в цепь, чтобы вместе завершать транзакцию. В этой цепи каждый объект может иметь только один Commit-Source и один Commit-Target. Изменения базы данных завершаются от головы (Commit Panel) к концу цепи.

Более подробное описание архитектуры ADM и некоторые примеры написания программ можно найти в документации "Progress Development Tools. ADM2 Guide and Reference".

V. ЭЛЕМЕНТЫ АДМИНИСТРИРОВАНИЯ

12. МНОГОТОМНЫЕ БАЗЫ ДАННЫХ

Progress по умолчанию определяет базу данных как одноотомную и генерирует 5 файлов с расширениями db, d1, lg, st и .b1. В следующей таблице указано назначение каждого из этих файлов:

Расширение	Описание
Db	Управляющий файл
d1	Область данных
Lg	Журнальный файл
St	Структурное описание базы
b1	Транзакционный файл (before-image)

Одноотомная база данных по определению ограничена одной файловой системой, в которой она расположена и, следовательно, на нее распространяются ограничения этой файловой системы.

Progress позволяет снять эти ограничения и расширить пространство, занимаемое базой. С помощью набора специальных утилит можно преобразовать одноотомную базу в многоотомную, или создать новую многоотомную базу данных. В отличие от одноотомных баз, многоотомные состоят из одного структурного файла (.st), одного управляющего файла (db), одного или более файлов данных (.dn), одного или более before-image файлов (.bn), одного или более after-image файлов (.an) и т.д.

Управляющий файл (.db) не содержит данных. Это всего лишь таблица, в которой указано местоположение сегментов базы данных. Созданием этого файла занимается специальная утилита Progress после того, как администратором подготовлено описание структуры многоотомной базы в виде текстового файла с расширением .st. Сегмент - это группа физических записей, размещение которых контролирует Progress. Данные записываются в сегменты в том порядке, в котором они определены в описании структурного файла. Progress начинает запись во второй сегмент после того, как будет исчерпано все пространство первого сегмента. Отметим, что объекты базы (такие как student, marks или course) можно располагать в предусмотренных сегментах базы.

Файл структурного описания создается в одном из доступных текстовых редакторов (например, в редакторе Progress). Этот файл не может содержать пустых строк, но может содержать комментарии - строки, которые начинаются символом #. В описании сегмента указываются: тип сегмента, имя, местоположение и, возможно, длина. Тип - это литера d для сегментов данных, литера b для before-image сегментов и литера a для after-image сегментов. Местоположение - стандартное имя файла в соответствующей операционной системе. Длина сегмента может быть фиксированной или переменной длины. Длина сегмента фиксированной длины описывается в блоках (1 блок = 1 кбайт). Число блоков должно быть кратно тридцати двум. Минимальная возможная длина - 32 блока, максимальная зависит от возможностей файловой системы.

Области размещения базы данных уточняются во время создания базы. Например, если в системе определены следующие 5 папок /usr1, /usr2, /usr3, /usr4 и /usr5, администратор базы данных может сконструировать следующий план размещения базы данных:

Наименование области	Физическая точка
Область первичного отката (Roll-Back Log)	/usr1
Схема базы	/usr2
Пользовательские данные	/usr3
Пользовательские индексы	/usr4
Область отката Roll-Forward	/usr5

Пример 12.1

Продemonстрируем основные этапы создания базы на примере создания многоотомной базы UNIV в директории c:\examples\db1 (предполагается, что одноотомная база расположена в директории c:\examples\db). Для создания такой многоотомной базы данных нужно выполнить следующие действия:

1. Создать в любом текстовом редакторе файла структурного описания. Ниже приведен пример файла структурного описания univ.st (этот файл находится в директории c:\examples\db1):

```
b c:\examples\db1\bi
#
```

```

a c:\examples\db1\ai
#
d "Schema Area" c:\examples\db1\schema
#
d "student",32 c:\examples\db1\student
#
d "course",32 c:\examples\db1\course
#
d "marks",32 c:\examples\db1\marks
#
d "Index",32 c:\examples\db1\index

```

2. Создать вспомогательные директории, в которых будут размещены экстенды области отката и пользовательские объекты базы:

```

c:\examples\db1\bi
c:\examples\db1\ai
c:\examples\db1\schema
c:\examples\db1\student
c:\examples\db1\course
c:\examples\db1\marks
c:\examples\db1\index

```

3. Запустить утилиту администратора prostrct со следующими параметрами:

```
d:\progress\bin\prostrct create c:\examples\db1\univ
```

4. Инициировать начальное состояние базы утилитой proddb:

```
d:\progress\bin\proddb c:\examples\db1\univ d:\progress\empty
```

Теперь с базой UNIV можно работать через DATA DICTIONARY, однако если вы хотите загрузить в многоотомную базу содержимое одноотомной базы UNIV, вам необходимо дополнительно выполнить следующие шаги:

5. Открыть одноотомную базу через Data Dictionary (C:\EXAMPLES\DB\UNIV). Дампировать схему (в среде Data Administration: Admin → Dump Data and Definition) и содержимое одноотомной базы данных UNIV в директорию C:\EXAMPLES\DB. Закрыть одноотомную базу.

7. Отредактировать файл (C:\EXAMPLES\DB\UNIV.DF) схемы базы (в любом текстовом редакторе, например, в процедурном редакторе Progress), заменив "Shema Area" для таблиц и индексов на "student", "course", "marks" и "Index".

8. Открыть многоотомную базу через Data Dictionary (C:\EXAMPLES\DB1\UNIV). Загрузить схему и содержимое базы (в среде Data Administration:Admin → Load Data and Definition) из файлов

```

C:\EXAMPLES\DB\UNIV.DF,
C:\EXAMPLES\DB\STUDENT.D,
C:\EXAMPLES\DB\MARKS.D,
C:\EXAMPLES\DB\COURSE.D.

```

9. Убедитесь, что база загружена должным образом т.е., что каждая таблица и индексы загружены в свою область.

13. ЗАЩИТА ОТ НЕСАНКЦИОНИРОВАННОГО ДОСТУПА

Защита данных от несанкционированного доступа в Progress осуществляется на четырех уровнях:

connect-security
compile-security
runtime-security
schema-security

13.1. Connect-security

Каждому пользователю в Progress могут быть приписаны:

USERID - идентификатор пользователя, строка из букв, цифр, символов \$, %, ^, -, _; прописные и строчные буквы не различаются;

PASSWORD - пароль, строка из 16 символов, прописные и строчные буквы различаются.

Создадим список идентификаторов пользователей (он хранится в таблице _USER):

DATA ADMINISTRATION → ADMIN → Security → Edit User List...

Сделаем disconnect, а затем снова connect базы данных (укажем при этом идентификатор пользователя и пароль - через Options>>).

Запретим доступ к базе незарегистрированных пользователей:

DATA ADMINISTRATION → ADMIN → Security → Disallow Blank Userid Access...

Теперь назначим администраторов базы данных (по умолчанию, администраторами являются все пользователи):

DATA ADMINISTRATION → ADMIN → Security → Security Administrators...

Доступ к редактированию списка пользователей после назначения администраторов будет разрешен только им.

Пароль может быть назначен и изменен самим пользователем:

DATA ADMINISTRATION -> ADMIN -> Security -> Change Your Password

Просмотреть список зарегистрированных пользователей (но не их паролей) и распечатать его можно следующим образом:

DATA ADMINISTRATION -> ADMIN -> Security -> User Report

13.2. Compile-security

Теперь рассмотрим, каким образом можно защитить таблицы и поля базы данных от программиста, компилирующего свои программы:

DATA ADMINISTRATION → ADMIN → Security → Edit Data Security → список таблиц

Выбираем из списка таблицу и определяем доступ к ней.

Действие:

Can-Read	- можно читать;
Can-Write	- можно модифицировать;
Can-Create	- можно добавлять записи;
Can-Delete	- можно удалять записи;
Can-Dump	- можно делать dump;
Can-Load	- можно делать load.

Право:

*	- разрешено всем;
id1,id2,...	- разрешено указанным;
!id5,!id6,*	- разрешено всем, за исключением указанных;
строка*	- разрешено всем, чей идентификатор начинается с этого префикса.

Аналогичным образом назначаем доступ к полям таблиц.

13.3. Runtime-security

Progress позволяет защитить от несанкционированного доступа не только базу данных, но и программы:

Первый способ - список доступа задан в процедуре.

Пример 13.3.1

```

DEFINE BUTTON add-b LABEL " insert ".
DEFINE BUTTON upd-b LABEL "update".
DEFINE BUTTON quit-b LABEL " quit ".
DEFINE FRAME a add-b SKIP upd-b SKIP quit-b WITH CENTERED.
ON CHOOSE OF add-b DO:
  IF USERID <> "id1"
  THEN DO:
    MESSAGE "not permission to you"
    VIEW-AS ALERT-BOX INFORMATION BUTTON OK.
    RETURN.
  END.
  RUN c:\examples\part13\add.p.
END.
ON CHOOSE OF upd-b DO:
  IF NOT CAN-DO("id1,id2")
  THEN DO:
    MESSAGE "not permission to you"
    VIEW-AS ALERT-BOX INFORMATION BUTTON OK.
    RETURN.
  END.
  RUN c:\examples\part13\upd.p.
END.
ENABLE ALL WITH FRAME a.
WAIT-FOR CHOOSE OF quit-b.

```

Процедура add.p:

```

REPEAT:
  INSERT student.
END.

```

Процедура upd.p:

```

REPEAT:
  PROMPT-FOR student.name_st.
  FIND student USING name_st.
  UPDATE student.
END.

```

Неудобство такого способа - при изменении списка доступа нужна перекомпиляция процедуры.

Второй способ - список доступа хранится в таблице базы данных:

Пример 13.3.2

Средствами DATA DICTIONARY опишем таблицу permission с полями:

```

activity   - x(10), CHARACTER
can-run    - x(60), CHARACTER

```

и индексом:

Index name	Primary	Unique	Components	Ascending	Word Index
i6	Yes	Yes	activity	Yes	No

Назначим администраторов программы и пользователей отдельных подпрограмм:

activity	can-run
security	adm1,adm2
add	id1
upd	id1,id2

используя процедуру: REPEAT: INSERT permission. END.

Доступ к чтению этой таблицы (Can-Read) разрешим всем, а доступ к изменению (Can-Write, Can-Create, Can-Delete) - только администраторам программы.

Изменим главную процедуру предыдущего примера, добавив еще один пункт - "admin", доступный только администраторам, и напомним процедуру sec.p для корректировки таблицы permission. Изменим также проверку доступности первого и второго пунктов.

```

DEFINE BUTTON add-b LABEL " insert ".
DEFINE BUTTON upd-b LABEL "update".
DEFINE BUTTON sec-b LABEL "admin ".
DEFINE BUTTON quit-b LABEL " quit ".
DEFINE FRAME a
    add-b SKIP upd-b SKIP sec-b SKIP quit-b
    WITH CENTERED NO-BOX.
ON CHOOSE OF add-b DO:
    FIND permission WHERE activity = "add".
    IF NOT CAN-DO(can-run)
    THEN DO:
        MESSAGE "not permission to you"
        VIEW-AS ALERT-BOX INFORMATION BUTTON OK.
    RETURN.
    END.
    RUN c:\examples\part13\add.p.
END.
ON CHOOSE OF upd-b DO:
    FIND permission WHERE activity = "upd".
    IF NOT CAN-DO(can-run)
    THEN DO:
        MESSAGE "not permission to you"
        VIEW-AS ALERT-BOX INFORMATION BUTTON OK.
    RETURN.
    END.
    RUN c:\examples\part13\upd.p.
END.
ON CHOOSE OF sec-b DO:
    FIND permission WHERE activity = "security".
    IF NOT CAN-DO(can-run)
    THEN DO:
        MESSAGE "not permission to you"
        VIEW-AS ALERT-BOX INFORMATION BUTTON OK.
    RETURN.
    END.
    RUN c:\examples\part13\sec.p.
END.

ENABLE ALL WITH FRAME a.
WAIT-FOR CHOOSE OF quit-b.

```

Процедура sec.p:

```

REPEAT:
    PROMPT-FOR permission.activity.
    FIND permission USING activity.
    UPDATE can-run.
END.

```

Теперь администратор программы может редактировать таблицу доступа во время выполнения программы.

13.4. Schema-security

Progress позволяет временно защитить структуру таблицы базы данных от изменений, независимо от доступности ее пользователям. "Замораживают" и "размораживают" таблицы следующим образом:

DATA ADMINISTRATION → UTILITIES → Freeze/Unfreeze

14. РЕГЛАМЕНТНЫЕ РАБОТЫ

14.1 Копирование и восстановление баз данных (Backup/Restore)

Progress предлагает утилиту PROBKUP для создания копий базы данных. Копии after-image файла следует создавать системными средствами. PROBKUP может работать как в **off-line**, так и в **on-line** режимах. On-line режим можно использовать только для multi-user баз данных, в системах с shared-memory. Если активен after-image процесс, то on-line копирование возможно только в случае использования after-image-extents.

Существуют два варианта выполнения копирования:

- full** - копирование всего файла;
- incremental** - копирование изменившихся блоков.

При разработке алгоритма регламентных работ по копированию базы данных, следует учитывать несколько факторов:

- время (зависит от интенсивности работы с базой);
- целостность (копирование .db, .ai, dump схемы базы...)
- размер базы данных (выполнять ли всегда полное копирование ...)

В особых случаях требуется нерегламентированное копирование (например, при переполнении диска) и его характеристики зависят от ситуации.

Полезно сохранять не только текущие копии файлов, но и поддерживать архивы копий - от 10 версий и больше. При архивировании следует как можно более полно маркировать версии:

- имя базы;
- дата, время;
- имя утилиты;
- full/incremental;
- имя устройства (порядковый номер);
- имя исполнителя;

Ниже приведены алгоритмы выполнения различных видов копирования. Сокращенный синтаксис вызова утилит, реализующих отдельные пункты алгоритмов, будет приведен чуть позже.

Off-line Backup.

1. Проверить, используется ли сейчас база данных (PROUTIL).
2. Завершить работу сервера (SHUTDOWN).
3. Выполнить усечение before-image файла (PROUTIL).
4. Выполнить копирование (full или incremental) (PROBKUP).

On-line Backup.

1. Выполнить копирование (full или incremental) (PROBKUP).

System Backup.

1. Проверить, используется ли сейчас база данных (PROUTIL).
2. Завершить работу сервера (SHUTDOWN).
3. Выполнить копирование системными средствами.
4. Пометить файл как имеющий копию - если копируется сама база (RFUTIL).

Восстановление базы данных.

1. Восстановить базу из полной (full) копии (PROREST).
2. Если после полного копирования были выполнены частичные копии (incremental), последовательно обновить базу (PROREST).

Краткое описание утилит.

Используется ли сейчас база данных (PROUTIL):

Первый вариант:

```
proutil db-name -C busy
```

Второй вариант:

```
proutil db-name -C holder
```


Усечение before-image файла (PROUTIL):
 proutil db-name -C truncate bi

Копирование базы данных утилитой PROBKUP:
 probkup [online] db-name [incremental] bk-name [options]

Восстановление базы данных утилитой PROREST:
 prorest db-name bk-name [options]

Пометка базы данных, как имеющей копию, если ее копирование производилось системными средствами (RFUTIL):
 rfutil db-name -C mark backedup

14.2. Поддержка after-imaging

After-imaging позволяет отслеживать изменения базы данных с момента последнего копирования базы для восстановления актуального состояния базы (roll forward recovery) в случае ее разрушения.

After-image файл (он имеет расширение .ai) должен храниться отдельно от файлов базы данных (на другом диске).

After-imaging не поддерживается автоматически. Администратор базы принимает решение о необходимости этого механизма, запускает его и поддерживает. Для запуска after-imaging (enable after-imaging) используется утилита RFUTIL:

```
rfutil db-name -C aimage begin [buffered/unbuffered]
```

Если эта утилита запущена на базу данных, которая была изменена с момента последней backup копии, последует сообщение об ошибке.

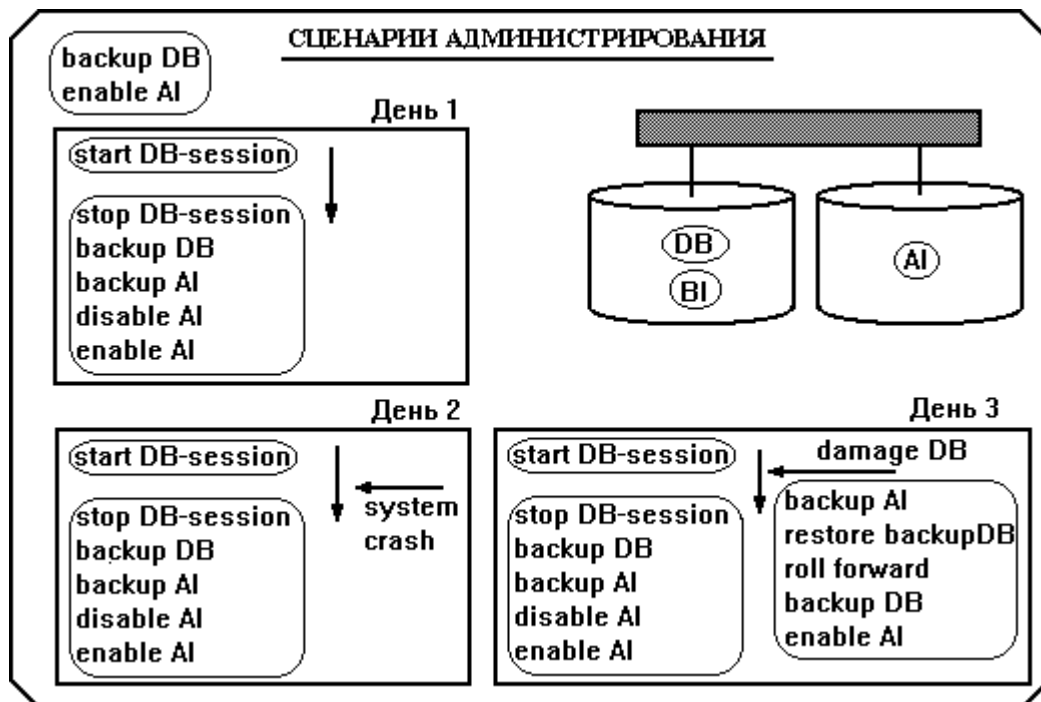
Перед переключением на новый after-image файл следует сделать копию базы.

Для снятия after-imaging используется утилита RFUTIL:

```
rfutil db-name -C aimage end
```

Полезно наряду с архивированием копий базы данных архивировать и after-image файлы.

Roll Forward Recovery.



Если разрушена база данных, следует:

1. Сделать копию after-image файла (системными средствами).
2. Восстановить последнюю копию базы данных (утилитой prorest).

3. Обновить восстановленную базу слиянием с after-image файлом:

```
rfutil db-name -C roll forward [verbose] -a ai-name [options]
```

verbose - указание этой опции приведет к появлению на экране отчета об обновляемых записях.

4. Сделать копию обновленной базы данных.

5. Запустить after-imaging (утилитой rfutil).

На рисунке приведены возможные сценарии администрирования базы данных при использовании механизма after-imaging. Рассматриваются три рабочих дня:

- в первый день работа идет без сбоев;
- во второй день происходит аварийное завершение сеанса из-за системной или машинной ошибки;
- на третий день разрушается база данных.

Вмешательство администратора требуется только в последнем случае.

Пример 14.2.1

Работа в режиме after-imaging с базой данных UNIV, восстановление базы после разрушения.

Утилиты запускаются вне сеанса Progress. Если сеанс не завершен, следует отсоединиться от базы данных.

1. Truncate BI:

```
_proutil c:\examples\db1\univ -C truncate bi
```

2. Backup DB:

```
probkup c:\examples\db1\univ c:\examples\db1\univ.bk
```

3. Enable AI:

```
rfutil c:\examples\db1\univ -C aimage begin
```

4. - Connect UNIV

- В Procedure Editor выполним следующую процедуру (реально изменим несколько записей):

```
FOR EACH student:
    UPDATE student.
END.
```

- Disconnect UNIV.

5. Damage DB (физически удалим все файлы UNIV, кроме UNIV.BK, UNIV.ST и UNIV.A1).

6. Файл UNIV.A1 передвинем из директории c:\examples\db1\ai в c:\examples\db1.

7. Restore Backup:

```
prorest c:\examples\db1\univ c:\examples\db1\univ.bk
```

8. Roll Forward Recovery:

```
rfutil c:\examples\db1\univ -C roll forward -a c:\examples\db1\univ.a1
```

9. - Connect восстановленной базы UNIV.

- Проверим, что восстановлены последние изменения базы:

```
FOR EACH student:
    DISPLAY student.
END.
```

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ 1. Запуск и снятие сеансов Progress, стартовые параметры

Вход в Progress в однопользовательском режиме:

Unix: pro [db-name] [parameters]
Windows: prowin32 [db-name] -1 [parameters]

Вход в Progress в многопользовательском режиме:

- Запуск клиента Progress:
 Unix: mpro [db-name] [parameters]
 Windows: prowin32 [db-name] [parameters]
- Запуск сервера Progress:
 Unix: proserve db-name [parameters]
 Windows: mprosrv db-name [parameters]

Завершение работы сервера: proshut db-name

Некоторые стартовые параметры клиентской сессии

Описание	Параметр
Язык	-lng
Формат дат	-d
Физическое имя базы данных	-db
Файл сортировок	-convmap
Файл параметров	-pf
Только чтение	-RO
Тип сетевого протокола	-N
Стартовая процедура	-p
Однопользовательский режим работы с базой	-1
Максимальное количество вложенных блоков	-nb
Логическое имя базы данных	-ld
Кодовая страница терминала	-cpterm
Кодовая страница принтера	-cpprint
Кодовая страница Internal	-cpinternal
Имя серверного процесса	-S
Имя host-машины	-H
Европейский числовой формат	-E
Директория для размещения триггеров	-trig
Век	-yy
Без замков	-NL
User	-U
Password	-P
ANSI SQL Client	-Q2
ANSI SQL	-Q

Некоторые стартовые параметры сервера

Описание	Параметр
Файл параметров	-pf
Тип сетевого протокола	-N
Количество пользователей	-n
Количество замков (по умолчанию - 500)	-L
Имя серверного процесса	-S
Имя host-машины	-H

ПРИЛОЖЕНИЕ 2. Способы открытия (connect) и закрытия (disconnect) баз данных

Connect:

- через окно Database Connections, вызываемое из соответствующих пунктов меню различных компонент, или через иконку DB List в панели PRO*Tools;
- при запуске сеанса Progress с указанием имени базы данных и прочих параметров (имя базы может быть единственным позиционным параметром или ключевым параметром с ключом -db);
- из программы посредством оператора Progress 4GL
CONNECT db-name [parameters] [NO-ERROR].

Disconnect:

- автоматически по концу сеанса Progress;
- из программы посредством оператора Progress 4GL
DISCONNECT db-name [NO-ERROR].

ПРИЛОЖЕНИЕ 3. Форматы данных

Форматирования значений типа DATE

Формат	Значение	Результат
99/99/99	3/10/1990	03/10/90
99/99/9999	3/10/2090	03/10/2090
99-99-99	3/10/1990	03-10-90
99-99-99	3/10/2090	????????
999999	3/10/1990	031090
999999	03/10/90	031090
99999999	03/10/1990	03101990

Форматирования строковых значений

Формат	Значение	Результат
xxxxxxx	These are characters	These a
x(9)	These are characters	These are
x(20)	These are characters	These are characters
xxx	These are characters	The
AAA-9999	abc1234	abc-1234
!!!-9999	abc1234	ABC-1234
(999) 999-9999	6176635000	(617) 663-5000

Форматирование целочисленных и вещественных значений

Формат	Значение	Результат
9999	123	0123
9,999	1234	1,234
\$zzz9	123	\$ 123
\$ > > > 9	123	\$123 (1)
\$->, > > 9.99	1234	\$1,234.00
\$->, > > 9.99	1234	\$1,234.00
#-zzz9.999	-12.34	#- 12.340
Tot= > > > 9Units	12	Tot=12Units
\$>, >>9.99	-12.34	????????
\$>, >>9.99	1234567	????????
>>, >99.99<<<	12,345.6789	12,345.68
>>, >99.99<<<	1,234.5678	1,234.568
>>, >99.99<<<	123.45	123.45
>>, >99.99<<<	12.45678	12.45678

Форматирование логических значений

Формат	TRUE	FALSE
yes/no	yes	no
Yes/No	Yes	No
true/false	true	false
да/нет	да	нет

ПРИЛОЖЕНИЕ 4. Цветовая настройка графической среды

Код цвета	RGB	Цвет	Код цвета	RGB	Цвет
0	0, 0, 0	Черный	8	192, 192, 192	Светло-серый
1	0, 0, 128	Синий	9	0, 0, 255	Голубой
2	0, 128, 0	Темно-зеленый	10	0, 255, 0	Зеленый
3	0, 128, 128	Зелено-голубой	11	0, 255, 255	Бирюзовый
4	128, 0, 0	Красный	12	255, 0, 0	Красный
5	128, 0, 128	Фиолетовый	13	255, 0, 255	Розовый
6	128, 128, 0	Желтовато-зеленый	14	255, 255, 0	Желтый
7	128, 128, 128	Серый	15	255, 255, 255	Белый

В файле progress.ini есть соответствующая секция Colors, в которой описаны эти настройки и менять их не рекомендуется, т.к. они активно используются в Progress ADE. При необходимости использования других цветов расширьте таблицу через системную переменную COLOR-TABLE и диалоговое окно SYSTEM-DIALOG COLOR.

ПРИЛОЖЕНИЕ 5. Файлы базы данных PROGRESS

Расширения файлов базы данных PROGRESS

Файл	Описание	Размещение
.an	After-image экстенды	Указывается в файлах структурного описания
.bn	Before-image экстенды	Указывается в файлах структурного описания
.dn	Экстенды данных	Указывается в файлах структурного описания
.db	Экстент управляющей области	Генерируется файлом структурного описания и размещается на любом доступном диске
.lg	Журнальный файл	В той же директории, что и управляющий файл
.lk	Файл блокировки (указывает на использование базы данных)	В той же директории, что и управляющий файл
.tn	Транзакционный журнальный файл	Указывается в файлах структурного описания и обычно располагается в той же директории, что и управляющий файл
.lic	Лицензионный файл	В той же директории, что и управляющий файл
.st	Файл структурного описания	Обычно располагается в той же директории, что и управляющий файл

Временные файлы клиентской сессии

Файл	Описание
.lbi	Локальный before-image файл (сабтранзакционные откаты)
.dbi	Хранит временные таблицы
.pge	Хранит содержимое буферов
.srt	Временное пространство для сортировок.
.trp	Хранит изменения Data Dictionary до тех пор, пока они не будут зафиксированы

ПРИЛОЖЕНИЕ 6. Разделители слов в WORD-индексах

Progress использует следующие таблицы символов для разделения слов в WORD-индексах:

LETTER	буквы(всегда являются частью слова);
DIGIT	цифры(всегда являются частью слова);
USE_IT	символы, являющиеся частью слова;
BEFORE_LETTER	символы, которые трактуются как часть слова, если следующий символ из таблицы LETTER;
BEFORE_DIGIT	символы, которые трактуются как часть слова, если следующий символ из таблицы DIGIT;
BEFORE_LET_DIG	символы, которые трактуются как часть слова, если следующий символ из таблицы DIGIT или LETTER;
IGNORE	символы, которые игнорируются при построении WORD-индексов;
TERMINATOR	символы, которые ограничивают слова и никогда не являются частью слова.

Таблицы могут быть заданы явно с помощью специальных утилит (см. Progress Programming Handbook).

По умолчанию Progress использует следующие таблицы:

LETTER	A-Z a-z
DIGIT	0-9
USE_IT	\$ % # @ _
BEFORE_DIGIT	· , -
IGNORE	'
TERMINATOR	все остальные символы.

ПРИЛОЖЕНИЕ 7. Мета-схема базы данных

В базе данных помимо таблиц, индексов и секвенций хранятся служебные таблицы, в которых описана структура данных - **мета-схема**. Имена служебных таблиц начинаются с символа "подчеркивание", увидеть их структуру можно в Data Dictionary, включив опцию просмотра "скрытых" таблиц через пункт меню View. Работать с содержимым этих таблиц можно прямо из программы, точно так же, как и с пользовательскими таблицами (заметим лишь, что редактировать их опасно). Например, список имен таблиц базы данных и их полей можно получить следующим образом:

Пример 7.1

```
FOR EACH _file:
  DISPLAY _file_ file-name WITH TITLE "ТАБЛИЦА" NO-LABEL.
  FOR EACH _field OF _file:
    DISPLAY _field_ field-name WITH TITLE "ПОЛЯ" NO-LABEL.
  END.
END.
```

ПРИЛОЖЕНИЕ 8. Структура генерируемых в АВ программ

Название раздела	Содержание раздела	Примечания
Definitions	Описания переменных и параметров	редактируется пользователем
Preprocessor Definitions	Описания препроцессорных имен	
Control Definitions	Описания data widgets, buttons, graphic widgets	
Frame Definitions	Описания фреймов	
Procedure Settings	Описания процедурных установок	
Create Window	Оператор создания окна	
Runtime Attributes and UIB Settings	run-time атрибуты и некоторые установки UIB	
Included-Libraries	Подключение библиотек, методов	
Control Trigger	Триггеры, созданные АВ и пользователем	редактируется пользователем
Main Block	Описание основной логики процедуры	редактируется пользователем
Internal Procedures and Functions	Стандартные процедуры (в т.ч. ENABLE и DISABLE) и внутренние процедуры и функции пользователя	редактируется пользователем

Разделы, разрешенные к редактированию, можно редактировать любыми средствами. Изменение прочих разделов приведет к тому, что с процедурой невозможно будет работать в АВ.

ПРИЛОЖЕНИЕ 9. Работа со шрифтами

В файле progress.ini есть следующая секция, в которой описаны шрифты, используемые в Progress по умолчанию:

```
[fonts]
*****
;
; THE DEFINITION OF FONT 0 THROUGH 7 IS PRIVATE TO THE PROGRESS ADE.
; MODIFYING FONTS 0 THROUGH 7 MAY PREVENT THE PROGRESS ADE FROM RUNNING.
; The following fonts definitions correspond to the ADE standards.
;
;  ? - DefaultFont from Startup Section
;
;  0 - DefaultFixedFont from Startup Section (1 char per PPU)
;
;  1 - Proportional System Font
;
;  2 - Editor Font for 4GL program entry
;
;  3 - TTY Simulator (should be fixed)
;
;  4 - Dynamically-sized widgets, eg status-line, selection-list
;
;  5 - Static widgets, eg. combo-boxes
;
;  6 - Dynamic, bold (TranMan2)
;
;  7 - Reserved
font0=Courier New, size=8
font1=MS Sans Serif, size=8
font2=Courier New, size=8
font3=Courier New, size=8
font4=MS Sans Serif, size=8
font5=MS Sans Serif, size=10
font6=MS Sans Serif, size=8, bold
font7=MS Sans Serif, size=8
```

Менять эти настройки не рекомендуется. Если во время исполнения приложения потребуются какие-то другие шрифты, можно их установить через системную переменную FONT-TABLE и диалоговое окно SYSTEM-DIALOG FONT. Следующие примеры демонстрируют приемы работы с ними.

Пример 9.1

В приведенной ниже программе через системную переменную FONT-TABLE и соответствующие атрибуты и методы уточняется количество доступных шрифтов и их размеры в пикселях.

```
DEFINE VARIABLE font-number AS INTEGER.
DEFINE VARIABLE max-count AS INTEGER.
max-count = FONT-TABLE:NUM-ENTRIES + 1.
DO font-number = 1 TO max-count:
    DISPLAY "FONT" + STRING(font-number)
    FONT-TABLE:GET-TEXT-HEIGHT-PIXELS(font-number)
    WITH DOWN.
    DOWN.
END.
```

Пример 9.2

В программе увеличиваются размеры таблицы шрифтов и шрифт с кодом 11 выбирается с помощью диалогового окна SYSTEM-DIALOG FONT.

```

FONT-TABLE:NUM-ENTRIES = 13.
DEFINE VARIABLE Font1 AS INTEGER INITIAL 11.
DEFINE VARIABLE st AS CHARACTER FORMAT "X(30)" FONT Font1.
DEFINE BUTTON BUTTON_1 LABEL "CHOOSE FONT".
DEFINE BUTTON BTN_EXIT LABEL "EXIT".
DEFINE FRAME fFont
  st SKIP
  BUTTON_1 BTN_EXIT WITH CENTERED NO-LABELS.

ON CHOOSE OF BUTTON_1 IN FRAME fFont
DO:
  SYSTEM-DIALOG FONT Font1.
END.
ENABLE ALL WITH FRAME fFont.
WAIT-FOR CHOOSE OF BTN_EXIT.

```

Обратите внимание, что выбранный шрифт с кодом 11 сохраняется до конца клиентской сессии. Для более длительного сохранения нужно предпринять дополнительные усилия и, например, нажать на кнопку Save Font Settings в любом подходящем диалоговом окне AppBuilder.

ПРИЛОЖЕНИЕ 10. Компиляция приложений

Компиляция процедур PROGRESS может быть осуществлена двумя способами:

- оператором COMPILE;
- утилитой APPLICATION COMPILER (Tools -> Application Compiler).

В результате компиляции создается файл с расширением **.r**, который представляет собой исполнимый и переносимый (в Progress среде) код.

Пример использования оператора COMPILE:

```
COMPILE C:\SOURCE\TEST1.P SAVE.
```

Заметим, что файл с расширением **.r** создается в той же директории, что и файл **test1.p**, если директория не задается явно опцией INTO. Например:

```
COMPILE C:\SOURCE\TEST1.P SAVE INTO C:\RCODE.
```

Вызов процедуры осуществляется оператором RUN, при этом Progress сначала пытается найти одноименный файл с расширением **.r** и только в случае, если такой файл не найден - с тем расширением, которое было указано в операторе RUN. Так, например, оператор

```
RUN TEST1.P.
```

попытается сначала найти и исполнить **test1.r** и лишь при его отсутствии - **test1.p**.

ПРИЛОЖЕНИЕ 11. Операции и функции допустимые в препроцессорных выражениях

Операции
+
-
*
/
=
<>
>
<
>=
<=
and
OR
NOT
Begins
Matches

Функции		
ABSOLUTE	LC	RIGHT-TRIM
ASC	LENGTH	RANDOM
DATE	LIBRARY	REPLACE
DAY	LOG	ROUND
DECIMAL	LOOKUP	SQRT
ENCODE	MAXIMUM	STRING
ENTRY	MEMBER	SUBSTITUTE
ETIME	MINIMUM	SUBSTRING
EXP	MODULO	TIME
FILL	MONTH	TODAY
INDEX	NUM-ENTRIES	TRIM
INTEGER	OPSYS	TRUNCATE
KEYWORD	PROPATH	WEEKDAY
KEYWORDALL	PROVERSION	YEAR
LEFT-TRIM	R-INDEX	

ПРИЛОЖЕНИЕ 12. Организация библиотек

Библиотека - это специальным образом организованный файл, который может использоваться для размещения откомпилированных процедур (с расширением *.r*).

Для работы с библиотекой используется утилита PROLIB. Имя библиотеки должно иметь расширение *.pl*. Progress открывает библиотеку во время первого к ней обращения. Открыв библиотеку, Progress размещает ее в оперативной памяти и сохраняет до конца сеанса.

Вызов утилиты PROLIB выглядит так:

```
prolib library-name -option [files] [-option [files]]
```

где files - это список имен файлов или шаблонов (с символами ? и *).

option - это указание на характер работы с библиотекой.

Перечислим некоторые опции:

```
create  - создание новой библиотеки;
add     - добавление процедур к библиотеке;
replace - замещение процедур в библиотеке;
delete  - удаление процедур из библиотеки;
list    - просмотр содержимого библиотеки;
extract - передвижение процедур из библиотеки в те каталоги, из которых они были взяты.
```

Вызов процедуры из библиотеки осуществляется одним из следующих двух способов:

1. Явное указание имени библиотеки и имени процедуры. Например:

```
RUN c:\lib\math.pl<<test1.r>>.
```

2. Включение имени библиотеки в PROPATH:

```
PROPATH = ... ; c:\lib\math.pl
```

и вызов процедуры без указания ее местоположения:

```
RUN test1.r.
```


ПРИЛОЖЕНИЕ А. Progress/SQL

PROGRESS/SQL состоит из двух компонент: Data Definition Language (DDL) и Data Manipulation Language (DML).

А.1. Язык обработки данных (DML)

Оператор SELECT

Оператор SELECT - наиболее используемый SQL-оператор, имеет следующий синтаксис:

```
SELECT [ ALL | DISTINCT ] { * | field-list } [ INTO var-list ]
FROM table-name [alt-name ][,table-name [alt-name]]*
[WHERE search-condition ]
[GROUP BY field[ ,field] *]
[HAVING search-condition ]
[ORDER BY sort-criteria ]
```

Замечание:

ALL - результирующее множество строк не факторизуется (эта опция используется по умолчанию);

DISTINCT - результирующее множество строк факторизуется.

Пример А.1.1

Фамилии всех студентов.

```
SELECT name-st FROM student
```

Пример А.1.2

Все сведения о студентах (все поля из таблицы STUDENT).

```
SELECT * FROM student
```

Пример А.1.3

Сравните результат работы двух следующих запросов:

```
SELECT ALL sex FROM student.
```

и

```
SELECT DISTINCT sex FROM student.
```

Ниже приведены агрегатные функции, которые можно использовать в операторе SELECT:

AVG([DISTINCT] expression)	Среднее значение
COUNT(*)	Количество записей
COUNT(DISTINCT expression)	Количество различных записей
MAX([DISTINCT] expression)	Максимальное значение
MIN([DISTINCT] expression)	Минимальное значение
SUM([DISTINCT] expression)	Сумма значений

Пример А.1.4

Количество записей в таблице STUDENT.

```
SELECT COUNT(*) FROM student.
```

Пример А.1.5

Средний возраст студентов.

```
SELECT AVG(YEAR(TODAY) - YEAR(bdate)) FROM student.
```

Замечание: здесь используются функции YEAR и TODAY, которые являются функциями не SQL, а окружения.

Пример А.1.6

Дни рождений самого старшего и самого юного студентов.

```
DEFINE VARIABLE minbdate AS DATE.
DEFINE VARIABLE maxbdate AS DATE.
SELECT MIN(bdate),MAX(bdate) INTO minbdate,maxbdate FROM student.
DISPLAY minbdate maxbdate.
```

Ниже представлены возможные варианты синтаксиса конструкта search-condition:

1. expression
2. expression [NOT] BETWEEN expression AND expression
3. field-name IS [NOT] NULL
4. field-name [NOT] LIKE "string" [ESCAPE "character"]
5. expression [NOT] IN ({ value-list!SELECT-statement})
6. expression relation-operator (SELECT-statement)
7. [NOT] EXISTS (SELECT-statement)
8. expression relation-operator { ANY | ALL | SOME } (SELECT-statement)

Замечание: здесь ключевые слова ANY и SOME - синонимы.

Пример А.1.7

Вся информация о студентах, родившихся после 01.01.75.

```
SELECT * FROM student WHERE bdate > 01/01/75.
```

Пример А.1.8

Фамилии студентов, родившихся в интервале от 01.01.75 до 01.01.77.

```
SELECT name_st FROM student WHERE bdate BETWEEN 01/01/70 AND 01/01/77.
```

Пример А.1.9

Студенты с неопределенным днем рождения (забыли ввести значение этого поля).

```
SELECT name_st FROM student WHERE bdate IS NULL.
```

Пример А.1.10

Отличники (т.е. студенты, у которых любая отметка равна 5).

```
SELECT name_st
FROM student
WHERE 5 = ALL (SELECT mark
                FROM marks
                WHERE marks.num_st = student.num_st )
```

Группировка записей осуществляется через конструкт GROUP BY с возможной опцией HAVING.

Пример А.1.11

Количество студентов и студенток (или количество записей в каждой группе поля sex).

```
SELECT sex, COUNT(*) FROM student GROUP BY sex.
```

Пример А.1.12

Номера зачеток студентов со средним баллом большим, чем 4.5.

```
SELECT num_st, AVG(mark) FROM marks
GROUP BY num_st
HAVING AVG(mark) > 4.5.
```

Соединение данных из нескольких таблиц в операторе SELECT осуществляется с помощью конструкта FROM.

Пример А.1.13

Фамилии студентов и оценки.

1-й вариант:

```
SELECT name_st, mark FROM student, marks
WHERE student.num_st = marks.num_st.
```

2-й вариант:

```
SELECT name_st, mark FROM student INNER JOIN
marks ON student.num_st = marks.num_st.
```

3-й вариант:

```
SELECT name_st, mark FROM student LEFT OUTER JOIN
marks ON student.num_st = marks.num_st.
```

Пример А.1.14

Фамилии студентов, оценки и названия курсов.

```
SELECT name_st, mark, name_c
FROM student, marks, course
WHERE student.num_st = marks.num_st
AND marks.code = course.code.
```

Пример А.1.15

Возможные варианты супружеских пар среди студентов (т.е. все пары студент и студентка).

```
SELECT student_m.name_st, student_m.sex, student_f.name_st, student_f.sex
FROM student student_m, student student_f
WHERE student_m.sex AND NOT student_f.sex.
```

Замечание: здесь впервые пришлось использовать альтернативные имена для таблиц - часть таблицы STUDENT (студенты) была переименована в STUDENT_M, а другая часть в STUDENT_F (студентки).

Конструкт ORDER BY используется для сортировки отобранных записей и имеет следующий синтаксис:

```
ORDER BY { n | expression } { ASC | DESC }
[, { n | expression } { ASC | DESC }]*
```

где n - порядковый номер поля в операторе SELECT.

Пример А.1.16

Фамилии студентов и оценки, упорядоченные по убыванию оценок.

```
SELECT name_st, mark FROM student, marks
WHERE student.num_st = marks.num_st
ORDER BY mark DESC.
```

Оператор UNION

Оператор UNION используется для объединения записей, полученных в результате исполнения нескольких операторов SELECT. Его синтаксис:

```
SELECT-statement
UNION
[ ALL ] { SELECT-statement | UNION-statement }
[ ORDER BY sort-criteria ]
```

Замечание: интересно отметить, что в результате исполнения оператора UNION множество строк по умолчанию факторизуется (если не используется опция ALL).

Пример А.1.17

Номера зачеток студентов, у которых есть двойки и пятерки.

```
SELECT num_st FROM marks WHERE mark = 5
UNION
SELECT num_st FROM marks WHERE mark = 2.
```

Оператор INSERT

Оператор INSERT предназначен для добавления новых записей. Его синтаксис:

```
INSERT INTO table-name [(field-list)]
{ VALUES (value-list) | SELECT-statement }
```

Пример А.1.18

Добавление нового студента.

```
INSERT INTO student (num_st, name_st)
VALUES (016, "Семенов").
```

Оператор UPDATE

Оператор UPDATE редактирует поля в таблицах. Его синтаксис:

```
UPDATE table-name SET field-name = { NULL | expression }
[ ,field-name = { NULL | expression }]*
[WHERE search-condition ]
```

Используется, как правило, для работы с множеством записей.

Пример А.1.19

Изменение номеров зачетов у всех студентов.

```
UPDATE student SET num_st = num_st + 100.
```

Пример А.1.20

Изменение адреса у всех студентов с фамилией 'Иванов'.

```
UPDATE student SET address = 'Красная пл., дом 2'
WHERE name_st = 'Иванов'.
```

Оператор DELETE

Оператор DELETE удаляет записи из таблицы в соответствии с указанным критерием. Его синтаксис:

```
DELETE FROM table-name [ WHERE search-condition ]
```

Пример А.1.21

Удаление записей с отметками единицами.

```
DELETE FROM marks WHERE mark = 1.
```

Пример А.1.22

Удаление всех записей из таблицы STUDENT.

```
DELETE FROM student.
```

Операторы работы с курсором

Операторы работы с курсором предназначены для позиционной работы с записями. В таблице приведен их общий список:

DECLARE CURSOR	Сопоставляет курсор оператору SELECT или UNION
OPEN	Выбирает записи, соответствующие оператору SELECT, и позиционирует курсор перед первой записью
FETCH	Позиционирует курсор на очередную запись и выбирает значения полей
Позиционный UPDATE	модифицирует запись, на которую указывает курсор
Позиционный DELETE	удаляет запись, на которую указывает курсор
CLOSE	закрывает курсор

Оператор DECLARE имеет следующий синтаксис:

```
DECLARE cursor-name CURSOR FOR
{SELECT-statement|UNION-statement}
```

Пример А.1.23

Объявление курсора.

```
DECLARE var-cursor CURSOR FOR
SELECT name-st,address
FROM student
WHERE num-st > 3
```

Пример А.1.24

Открытие курсора (выборка соответствующих записей).

```
OPEN var-cursor.
```

Пример А.1.25

Позиционирование курсора и выборка полей в "простые" переменные.

```
FETCH var-cursor INTO var-name, var-address
```

Пример А.1.26

Позиционное редактирование (редактируется именно та запись, на которую сейчас указывает курсор).

```
UPDATE student SET address = var-address
WHERE CURRENT OF var-cursor
```

Пример А.1.27

Позиционное удаление (удаляется именно та запись, на которую сейчас указывает курсор).

```
DELETE FROM student WHERE CURRENT OF var-cursor
```

Пример А.1.28

Закрытие курсора (освобождение памяти).

```
CLOSE var-cursor
```

Пример А.1.29

Упомянутые выше операторы работы с курсором можно использовать, например, в следующем контексте:

```
.....
объявление переменных var-name, var-address
.....
DECLARE var-cursor CURSOR FOR
    SELECT name-st,address
    FROM student
    WHERE num-st > 3.
OPEN var-cursor.
LOOP:
FETCH var-cursor INTO var-name, var-address.
.....
... редактирование var-name, var-address .....
.....
UPDATE student SET name-st = var-name, address = var-address
WHERE CURRENT OF var-cursor.
.....
END LOOP.
.....
CLOSE var-cursor.
```

Транзакции

PROGRESS/SQL трактует операторы INSERT, DELETE и UPDATE как единую транзакцию (сабтранзакцию) и, следовательно, если во время исполнения транзакции (сабтранзакции) произойдут какие-то сбои, будут отменены все изменения базы данных, которые успели произойти с момента начала транзакции (сабтранзакции).

Пример А.1.30

Попробуйте выполнить следующую процедуру (в примере подразумевается, что в таблице STUDENT уже есть студент с num_st равным 1):

```
UPDATE student SET num_st = 1 WHERE num_st > 3.
```

Поскольку PROGRESS имеет собственные средства управления транзакциями, нет необходимости завершать транзакцию явно.

Если вы используете конструкции SQL, чтобы скорректировать базу данных (INSERT, UPDATE, или DELETE), то PROGRESS/SQL выполняет целую операцию или ничего совсем. Это отличается от поведения PROGRESS. Например, если Вы корректируете 50 строк, используя собственные конструкции PROGRESS, и ошибка происходит в строке 50, то происходит откат только этой строки, а остальные 49 строк остаются скорректированными. Если та же ошибка происходит когда, вы используете конструкции SQL, то чтобы PROGRESS/SQL для восстановления базы данных, отменяет все 50 строк.

Поскольку коррекции SQL рассматриваются как единственная транзакция, PROGRESS сохраняет запись заблокированными для всех используемых строк до конца транзакции, чтобы обеспечить rollback. Для того, чтобы включить утверждение UNDO в ваши SQL INSERT, UPDATE или DELETE, вы должны включить объемлющую транзакцию, чтобы аннулировать сделку, что утверждение SQL

начиналось. Например, в следующей процедуре UNDO не может отменять всю работу; утверждение UPDATE рассматривается как транзакция.

```
DO ON ERROR UNDO, LEAVE:
  UPDATE Customer SET Credit-Limit = 0 WHERE State = 'NH'
UNDO, LEAVE.
END.
```

Триггеры базы данных

Операторы SQL так же, как и операторы PROGRESS 4GL вызывают исполнение schema-триггеров базы данных. В следующей таблице приведены операторы SQL и соответствующие им триггеры.

Оператор SQL	Триггер
DELETE FROM	FIND, DELETE
FETCH	FIND
INSERT INTO	CREATE, ASSIGN, WRITE
SELECT	FIND
UPDATE	FIND, ASSIGN, WRITE
UNION	FIND

А.2. Язык описания данных (DDL)

Таблицы в Progress могут создаваться как через Data Dictionary (Progress-таблицы), так и с помощью DDL операторов (Progress/SQL-таблицы). Однако следует иметь в виду, что их дальнейшая модификация возможна лишь теми же средствами, какими они были созданы. Работа с данными в том и другом случае обеспечивается как PROGRESS/4GL операторами, так и PROGRESS/SQL операторами.

Оператор CREATE

Оператор CREATE используется для создания таблиц. Его синтаксис:

```
CREATE TABLE table-name ({field-name type [ options ]} | {UNIQUE (field-list)}
  [{ , field-name type [ options ]} | { ,UNIQUE (field-list)}]*)
```

Ниже приведены типы данных, используемые в PROGRESS/SQL.

```
CHARACTER(n)
INTEGER
SMALLINT
DECIMAL[(m[,n])]
FLOAT[(m)]
DOUBLE PRECISION
DATE
LOGICAL
REAL
NUMERIC[(m[,n])]
```

Оператор ALTER TABLE

Этот оператор позволяет редактировать SQL-таблицы.

Добавление поля:

```
ALTER TABLE table-name ADD COLUMN field-name type [ options ]
```

Удаление поля:

```
ALTER TABLE table-name DROP COLUMN field-name
```

Изменение поля:

```
ALTER TABLE table-name ALTER COLUMN field-name [ options ]
```

Оператор DROP TABLE

Используется для удаления таблиц. Его синтаксис:

```
DROP TABLE table-name
```

Оператор CREATE INDEX

Используется для создания индексов. Синтаксис:

```
CREATE [ UNIQUE ] INDEX index-name ON table-name (field-list)
```

Оператор DROP TABLE

Удаление индекса:

```
DROP INDEX index-name
```

- Выполнение операторов DDL разрешено только зарегистрированным пользователям (см. главу 13. “ЗАЩИТА ОТ НЕСАНКЦИОНИРОВАННОГО ДОСТУПА”).

Пример А.2.1

1. Создание таблицы POSTGRAD:

```
CREATE TABLE postgrad
( num_p INTEGER NOT NULL,
  name CHARACTER(15),
  address CHARACTER(15),
  bdate DATE ).

MESSAGE "TABLE postgrad IS ADDED"
VIEW-AS ALERT-BOX INFORMATION BUTTONS OK.
```

2. Создание индекса по таблице POSTGRAD:

```
CREATE INDEX p1 ON postgrad(name).
```

3. Добавление в таблицу postgrad студентов отличников:

```
INSERT INTO postgrad (num_p,name,bdate,address)
SELECT num_st,name_st,bdate,address FROM student WHERE
NOT EXISTS(SELECT * FROM marks WHERE
marks.num_st = student.num_st AND
marks.mark < 5) .
```

4. Добавление к таблице POSTGRAD дополнительного поля sa:

```
ALTER TABLE postgrad ADD COLUMN sa CHARACTER(15).
```

VIEW-таблицы

Создание VIEW-таблиц:

```
CREATE VIEW view-name [(field-list)]
AS SELECT-statement [ WITH CHECK OPTION ]
```

Удаление VIEW-таблиц:

```
DROP VIEW view-name
```

Работать с VIEW-таблицами можно так же, как и с обычными, однако следует иметь в виду ограничения, налагаемые на редактируемые VIEW-таблицы:

- нельзя использовать конструкции DISTINCT, GROUP BY, HAVING или агрегатные функции в соответствующем операторе SELECT;
- конструкт FROM в операторе SELECT может специфицировать только одну таблицу;
- нельзя использовать вложенные SELECT;
- если конструкт FROM специфицирует VIEW-таблицу, эта таблица должна быть, в свою очередь, редактируемой.

Пример A.2.2

1. Создание VIEW-таблицы:

```
CREATE VIEW exstud AS SELECT * FROM student WHERE num_st > 5
WITH CHECK OPTION.
```

2. Просмотр ее содержимого:

```
SELECT * FROM exstud.
```

3. Добавление новой записи:

```
INSERT INTO exstud (num_st,name_st) VALUES (25,"Robin").
```

4. Неудачная попытка добавления новой записи:

```
INSERT INTO exstud(num_st,name_st) VALUES (0,"Jons").
```

5. Редактирование записи:

```
UPDATE exstud SET num_st = 24 WHERE num_st = 25.
```

6. Удаление записи из таблицы exstud:

```
DELETE FROM exstud WHERE num_st = 24.
```

Операторы GRANT и REVOKE

Назначение и ограничение прав доступа к SQL-таблицам осуществляется операторами GRANT и REVOKE. Они имеют следующий синтаксис:

```
GRANT {ALL [ PRIVILEGES ]} |
      {[ SELECT ] [ INSERT ] [ DELETE ] [ UPDATE [( field-list )] ]}
      ON table-name TO {grantee-list | PUBLIC} [ WITH GRANT OPTION ]

REVOKE {ALL [ PRIVILEGES ]} |
       {[ SELECT ] [ INSERT ] [ DELETE ] [ UPDATE [( field-list )] ]}
       ON table-name TO {grantee-list | PUBLIC}
```

Пример A.2.3

1. Назначение прав доступа к таблице postgrad пользователю ivan:

```
GRANT ALL PRIVILEGES ON postgrad
TO ivan WITH GRANT OPTION.
```

2. Убедимся, что пользователь ivan может выполнить следующую процедуру, а пользователь egor - не может:

```
FOR EACH postgrad:
  UPDATE postgrad.
END.
```

3. Ограничим права пользователя ivan:

```
REVOKE UPDATE ON postgrad FROM ivan.
```

4. Убедимся, что теперь он не может выполнить редактирование таблицы postgrad.


ПРИЛОЖЕНИЕ В. Report Builder

Для создания отчетов в интерактивном режиме существует специальная компонента Progress - Report Builder. Заметим, что Report Builder требует отдельного connect с базой данных, поэтому в однопользовательском сеансе перед входом в среду Report Builder следует отсоединиться от базы данных.

Пример В.1.

Создание простого отчета по таблице student.

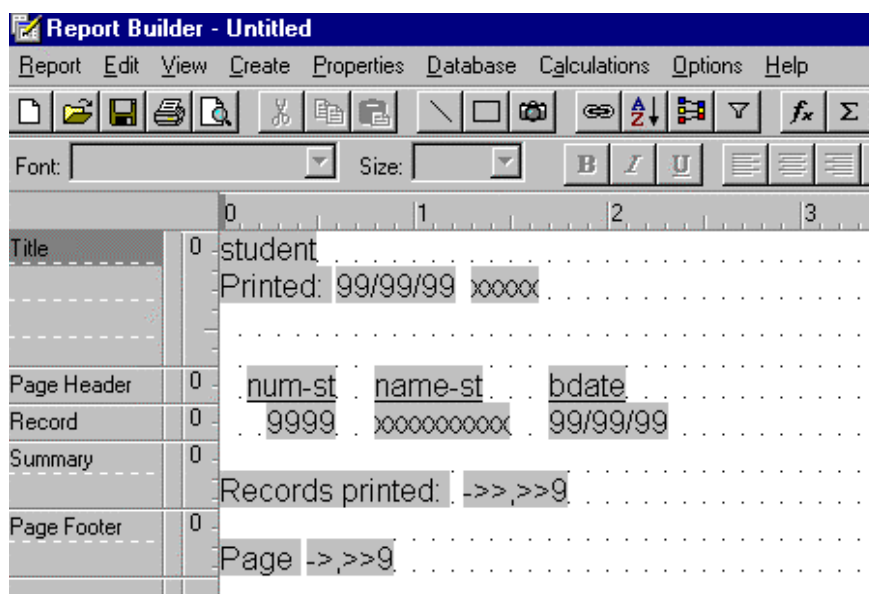



1. Находясь в среде Report Builder, выберем  (новый отчет). Произведем connect с базой данных Univ. В окне Database Tables выберем таблицу student в качестве основной таблицы создаваемого отчета.

2. Согласимся на создание Instant layout (стандартный шаблон отчета); в окне Instant Layout Fields выберем для отчета поля num_st, name_st, bdate таблицы student.

3. Созданный шаблон состоит из следующих разделов:

- Title - заголовок отчета;
- Page Head - заголовок страницы;
- Record - область размещения записей;
- Summary - область суммарных значений;
- Page Footer - подножие страницы.



4. Просмотрим отчет: Report → Print Preview (или ). Если русские буквы не удались, вернемся к шаблону отчета и сменим шрифт соответствующих полей или всего отчета сразу (выделив мышью соответствующие объекты).

student
Printed: 24/08/99 13:59

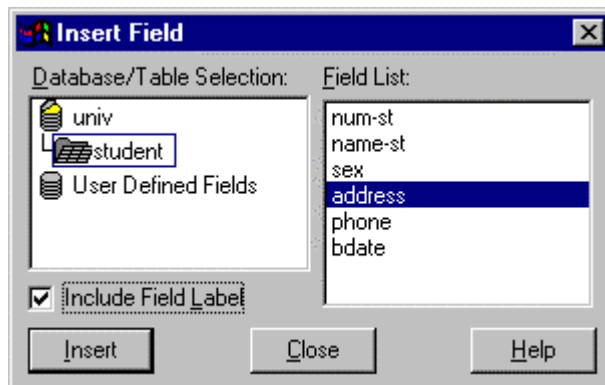
num-st	name-st	bdate
0001	Иванов	01/12/73
0002	Петров	27/02/74
0003	Сидорова	30/03/75
0004	Зотова	03/05/80
0005	Смирнов	05/11/77
0006	Кротов	12/11/70
0007	Косова	05/12/78
0008	Кузьмин	01/12/78
0009	Степанов	20/01/72
0010	Иванова	01/01/70
0011	Кошкин	30/12/81
0012	Глебова	04/05/76

Records printed: 12

5. Удалим поле bdate, выделив его вместе с меткой и нажав клавишу Delete или кнопку Trashcan



6. Добавим поля address и phone: установим курсор справа от поля name_st в разделе Record и нажмем клавишу Insert (или щелкнем по этому месту правой кнопкой мыши и выберем из popup меню пункт Insert Field). В открывшемся окне установим *Include Field Label*, выберем поле address, нажмем кнопку Insert. Не выходя из окна, сделаем то же самое с полем phone.



7. Изменим формат вывода поля phone ("9999999"): Properties → Format (или двойной щелчок мышью по полю, или через popup menu).

8. Изменим заголовок отчета (раздел Title):

- удалим текущий заголовок;
- напомним заголовок "Список студентов" (последующее редактирование текста можно будет осуществлять через его Properties (Edit In-Place);
- вставим пустые строки до и после заголовка (Edit → Insert Band Line или щелчок правой клавишей мыши по служебному серому полю слева);



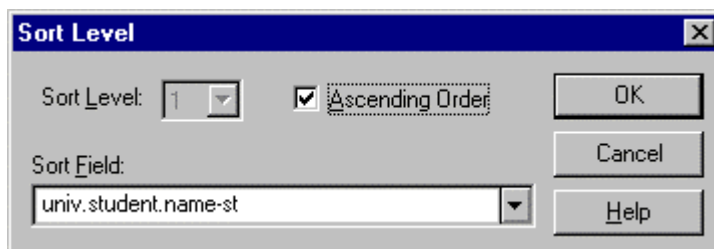
- заключим его в прямоугольную область (



- вставим картинку (



9. Отсортируем список студентов по фамилиям (- Sort Order).



10. Установим фильтр на выборку записей – num_st от 5 до 10 (- Filter).



11. Расположим записи не в одну, а в несколько строк:

- вставим в конец раздела Record пустую строку (Create → Band Line);
- передвинем поле phone в новую строку, перенесем его метку в ту же строку слева от поля;
- уменьшим ширину поля address (Width - в Properties → Format) до 7 символов и установим перенос слов (Word Wrap) в Properties → Alignment для этого поля:


num-st	name-st	address	
9999	xxxxxxxxxx	xxxxxxxxxx	
			phone 9999999

12. Разделим столбцы отчета вертикальными линиями (). Просмотрим отчет:

Список Студентов



<u>num-st</u>	<u>name-st</u>	<u>address</u>
0007	Косова	Дачный, 56 <u>phone</u> 3425476
0006	Кротов	Ботанич ,6 <u>phone</u> 4126577
0008	Кузьмин	Садовая ,2 <u>phone</u> 4565463
0009	Степанов	Фуршт., 17 <u>phone</u> 2878976

13. Запишем созданный шаблон в библиотеку отчетов (Record → Save или ). В библиотеке каждый шаблон идентифицируется своим именем, допускающим русские буквы, пробелы и т.д.)

Пример В.2.


Создание отчета с вложенными группами.

	<u>num-st</u>	<u>name-st</u>	<u>age</u>
пол: f			
несовершеннолетние			
	0004	Зотова	19
	0010	Иванова	17
совершеннолетние			
	0012	Глебова	23
	0003	Сидорова	24
	0007	Косова	21
всего в группе	5	студентов; средн.возраст -	21
пол: m			
несовершеннолетние			
	0002	Петров	18
	0009	Степанов	18
	0011	Кошкин	18
совершеннолетние			
	0006	Кротов	29
	0005	Смирнов	22
	0008	Кузьмин	21
	0001	Иванов	26
всего в группе	7	студентов; средн.возраст -	22

1. Начнем создание нового шаблона отчета - .

2. Создадим Instant-отчет по таблице student (с полями num_st, name_st).


3. Организуем группирование списка студентов по полю sex:


- назначим условие группировки ();

- отсортируем записи по полю sex ();


- создадим заголовок группы: Create → Band Line, щелчком мышкой по Different Type и выберем из списка заголовков группы первого уровня (1GH-sex);

- в шаблоне отчета вставим поле sex в раздел 1GH; посмотрим отчет;





- определим новое поле count-sex () - количество записей в группе (укажем, что количество - count - считается по полю num_st и это значение обнуляется перед началом каждой новой группы: в Reset укажем sex);
- добавим в шаблон раздел 1GF-sex – подножие группы (Create → Band Line) и вставим в него поле count-sex (в окне *Insert Fields* найдем это поле в *User Defined Fields – Aggregate Fields*).

4. С помощью редактора, открытого через , добавим вычисляемое поле age (возраст студента) и вставим его в раздел Record:

$\text{YEAR}(\text{TODAY}()) - \text{YEAR}(\text{bdate})$

5. Вычислим и выведем средний возраст студентов по группам: создадим новое поле avr-age через , укажем здесь же обновление накапливаемого значения (Reset) по группе sex, вставим новое поле в 1GF-sex.

6. Создадим группы второго уровня - по совершеннолетию студентов:

- создадим вычисляемое поле log-age: $\text{age} \geq 21$ ();
- назначим условие группировки второго уровня - по log-age () и сортировку по нему ();
- создадим вычисляемое поле head-age ():
 $\text{IIF}(\text{log-age}, \text{"совершеннолетние"}, \text{"несовершеннолетние"})$
- добавим раздел 2GH-log-age (Create → Band Line), и вставим в него новое поле head-age;
- добавим в шаблон комментарии: "пол:", "всего в группе:", "студентов; средний возраст -" и разделительные линии; удалим раздел Summary;

Title	0	student	Printed: 99/99/99	xxxxx
Page Header	0			
1GH-sex	0	пол: m		
2GH-log-age	0	xxxxxxxxxxxxxxxxxxxx		
Record	0	9999	xxxxxxxxxx	->,>>9
1GF-sex	0	всего в группе ->>9	студентов; средн.возраст ->>,>>9	
Page Footer	0	Page ->,>>9		


- посмотрим отчет; запишем созданный шаблон отчета в библиотеку отчетов (Report → Save).

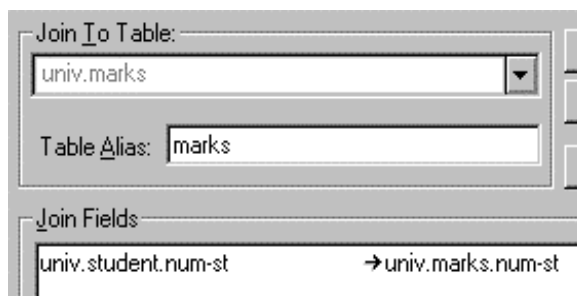
Пример В.3.


Создание отчета по связным таблицам.

Косова	5	Английский
	5	Алгебра
	5.00	
Кузьмин	4	Геометрия
	5	Физкульт.
	4.50	
Степанов	3	Английский
	4	Алгебра
	3	Геометрия
	3.33	

1. Начнем создание нового шаблона отчета.

2. Создадим Instant report с основной таблицей student (выберем одно поле - name_st).
3. Для таблицы student определим ее связь через поле num_st с таблицей marks ():



4. Укажем группирование по num_st и добавим в шаблон отчета раздел 1GH-num_st.
5. Поле name_st перенесем из раздела Record в раздел 1GH-num_st, его метку удалим.
6. В раздел Record вставим поле marks.mark (без метки).
7. Установим новую связь - marks через поле code с таблицей course.
8. В раздел Record вставим поле course.name_c.
9. Вычислим средние баллы студентов:
 - добавим раздел 1GF-num_st (Create → Band Line);
 - определим новое поле avg-mark (), указав поле mark и установив Reset для группы;
 - вставим это поле в раздел 1GF-num_st и изменим его формат на "9.99".
10. Запишем созданный шаблон отчета в библиотеку.

Пример В.4.

Самостоятельно создадим аналогичным способом новый шаблон с основной таблицей course и связанными с ней таблицами marks и student, группируя оценки по предметам и выводя в отчет средний балл по предмету.

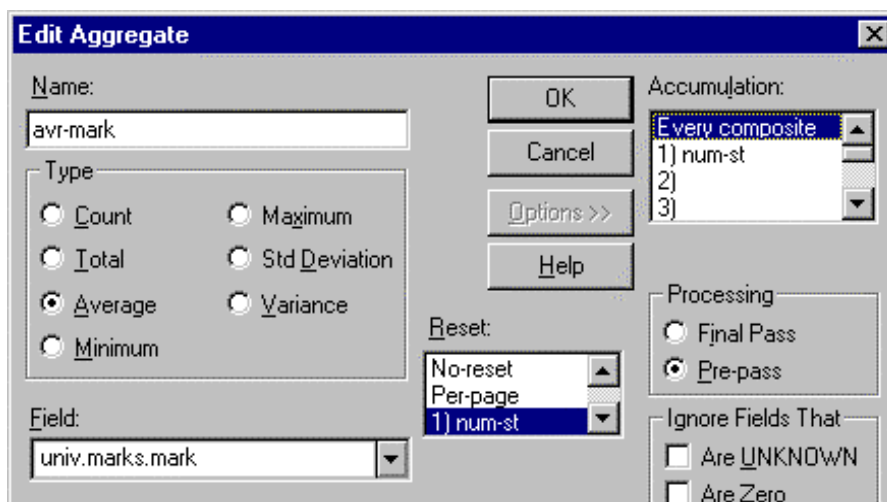
Пример В.5.

Создание двухпроходного отчета.

Создадим новый отчет, в котором будут фигурировать имена студентов и их средний балл, причем сортировка произойдет по среднему баллу (что потребует двухфазного прохода по отчету):

<u>name-st</u>	
Иванов	5.00
Косова	5.00
Иванова	4.67
Кузьмин	4.50
Глебова	4.33
Петров	4.00
Зотова	3.50

- создадим Instant report по таблице student (поле name_st);
- определим связь с таблицей marks;
- укажем группирование по num_st и создадим раздел 1GF-num_st;
- перенесем name_st в 1GF-num_st и удалим раздел Record;
- создадим агрегатное поле avg-mark, укажем для него в Options (в окне New Aggregate) в Processing pre-pass - "подсчитывать при предварительном проходе":



- вставим новое поле справа от name_st, изменим его формат на 9.99;
- укажем сортировку по убыванию этого поля;
- запишем созданный шаблон отчета в библиотеку.

Пример В.6.

Создание шаблона суммарного отчета по таблице student:

Всего студентов -	12	
средний возраст -		21
возраст самого младшего студента -		17
возраст самого старшего студента -		29

- создадим новый отчет по таблице student; откажемся от Instant report;
- удалим раздел Record и добавим раздел Summary;
- создадим вычисляемое поле avr – возраст студента (как в примере 2 – п.4);
- создадим и разместим в Summary агрегатные поля
 - count-st (“всего студентов”),
 - avr-age (“средний возраст”),
 - min-age (“возраст самого младшего студента”),
 - max-age (“возраст самого старшего студента”);
- добавим картинки, нестандартный заголовок, обрамление и т.д.;
- запишем созданный шаблон отчета в библиотеку.

Пример В.7.

Создание отчета с мемо-полями.

В случае, когда отчет должен содержать большие текстовые поля, удобно подготавливать и хранить их в отдельных файлах (мемо-файлах). Такие поля называются мемо-полями и могут включать в себя ссылки на поля базы данных, вычисляемые поля и т.д. Мемо-файлы могут быть созданы в текстовых редакторах под DOS или Windows (например, в Procedure Editor) и состоят из описаний одного или более мемо-полей, представляющих собой неформатированные тексты произвольной длины, имеющие заголовки с уникальными именами и заключенные в фигурные скобки. Такой файл должен иметь следующую структуру:

```

NEWMEMO memo-field1:
{
    текст
}
NEWMEMO memo-field2:
{
    текст
}

```

Мемо-поля могут включать в себя специальные управляющие символы:

- знак перевода каретки: ~n
- установка шрифта: <Fназвание> (например: <FNTTimes/Cyrillic>)
- установка размера шрифта: <Pчисло> (например: <P18>)
- установка стиля:
 - - Bold
 - <I> - Italic
 - <U> - Underscore
 - <N> - Normal
- возможны сочетания типа <BI> - Bold & Italic
- возврат к установкам из шаблона отчета: <D>

Указатели на поля базы данных и переменные выглядят следующим образом:

{@имя}

Пустые строки (не путать с пробельными строками) останутся пустыми и в отчете, поэтому знак ~n перед ними ставить необязательно.

Подготовленное мемо-поле может быть вставлено в любой отчет.

Существует возможность условной подстановки полей, указанных в мемо - через вычисляемые поля, описанные в Report Builder с использованием функций IIF или CASE:

В качестве примера создадим серию извещений о сроке защиты диплома для студентов-пятикурсников.

1. Создадим в Procedure Editor файл, включающий в себя одно мемо-поле dip:

```

NEWMEMO dip:
{
<FNTTimes/Cirillic><P16>
Кому: <U>{@name_st} <N>~n
Адрес: {@address}

```

Извещаем Вас, что защита вашего диплома состоится {@date-d}. Информация об аудиторией будет находиться на доске объявлений. За полчаса до назначенного времени Вам следует подойти к секретарю комиссии.

```

Деканат.~n
{@date-rb}
}

```

Сохраним этот файл на диске.

2. Создадим в Report Builder шаблон отчета (не Instant) с базовой таблицей student. Определим связную таблицу marks.

3. Укажем группирование по student.num_st и создадим раздел 1GH-num_st; удалим раздел Record.

4. Укажем сортировку по name_st.

5. Опишем агрегатное поле avr-mark.

6. Опишем вычисляемую переменную date-d:

```

CASE(round(avr-mark,0), 3, "8 июня в 10.00", 4, "6 июня в 14.00",
5, "4 июня в 10.00", "в будущем году")

```

7. Опишем вычисляемую переменную date-rb: TODAY()

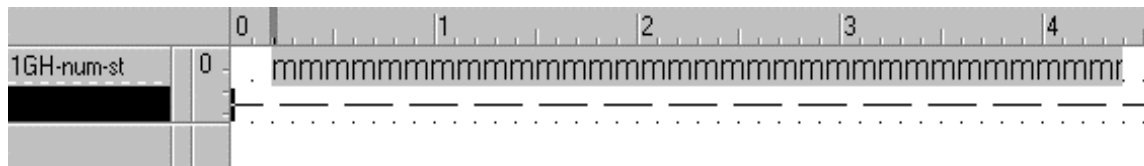
8. Для того чтобы вставить мемо-поле в отчет, следует выбрать:

Report → Attach Memo File

и указать в открывшемся окне файл, в котором описано это поле. Затем мемо-поле вставим в раздел 1GH-num_st шаблона отчета точно так же, как и любое другое поле. По умолчанию оно имеет свойства переноса слов и выравнивания по левому краю. Выравнивание, также как и ширину поля (Width), можно изменить через Property. Попытка отключения переноса слов приведет к тому, что в готовом отчете появится только начало мемо-поля.

9. В раздел 1GH добавим строку - разделитель страниц:

Create → Band Line (New Page Line)



10. Просмотрим созданный отчет и запишем его в библиотеку.

Пример В.8.

Вызов готового отчета из приложения.

Существует возможность вызывать готовые отчеты из приложения также, как и любые другие программные модули (без запуска Report Builder). При этом исходной информацией для создания отчета является, во-первых, шаблон отчета хранящийся в библиотеке, а во-вторых, набор параметров, определяющих характеристики печати и некоторые характеристики отчета. Параметры могут быть указаны либо непосредственно в приложении при вызове отчета, либо могут храниться в Progress-таблице (первый способ называется "PRINTRB interface", второй - "Table interface").

Можно передавать целый набор параметров, часть из которых может перекрывать свойства шаблона (например, фильтр записей). Помимо 19 обязательных позиционных параметров можно использовать дополнительные, определенные пользователем.

Стандартные параметры:

Название	Назначение	Тип
RB-REPORT-LIBRARY	имя библиотеки отчетов	C
RB-REPORT-NAME	имя отчета	C
RB-DB-CONNECTION	строка открытия базы данных	C
RB-INCLUDE-RECORDS	флаг фильтра: "" - без фильтра, "O" - с фильтром	C
RB-FILTER	выражение-фильтр	C
RB-MEMO-FILE	имя мемо-файла	C
RB-PRINT-DESTINATION	направление печати: "D" - с предварительным просмотром на экране, "" - без просмотра	C
RB-PRINTER-NAME	имя принтера	C
RB-PRINTER-PORT	порт для принтера	C
RB-OUTPUT-FILE	имя выходного файла	C
RB-NUMBER-COPIES	количество копий	N
RB-BEGIN-PAGE	начальный номер страницы	N
RB-END-PAGES	конечный номер страницы	N
RB-TEST-PATTERN	флаг тестирования шаблона	L
RB-WINDOW-TITLE	заголовок окна	C
RB-DISPLAY-ERRORS	флаг высвечивания ошибок	L
RB-DISPLAY-STATUS	флаг создания файла-статуса (создается файл rbrun.out в текущей директории)	L
RB-NO-WAIT	синхронный/асинхронный процесс	L
RB-OTHER-PARAMETER	дополнительные параметры	C
RB-TAG	какие отчеты генерировать	C
RB-STATUS	статус отчета	C

где C - указывает на символьный тип параметра (стандартное значение ""),
N - целый тип (умалчиваемое значение 0),
L - логический тип (умалчиваемое значение no).

Заметим, что RB-NO-WAIT - используется только для PRINTRB interface, а RB-TAG и RB-STATUS - только для Table interface.

Параметр RB-OTHER-PARAMETER указывает либо на отсутствие дополнительных параметров (""), либо на таблицу для замены таблицы такой же структуры из отчета, либо на определенные программистом параметры.

Выполним следующий оператор из любого приложения:

```
RUN aderb\_printrb.p ("reports.prl", "report3", "univ.db", "O", "num_st > 10", "", "D",
    "", "", "rep3.txt", 0, 1, 99, no, "", yes, yes, no, "").
```

Определить собственные параметры можно через вычисляемое поле в Report Builder, используя функцию RUNTIME-PARAMETER. В девятнадцатом параметре в этом случае указывается следующая конструкция:

имя_параметра = значение

Например, определив в Report Builder (в шаблоне отчета из примера 7.3.6.) вычисляемое поле user-id:

```
RUNTIME-PARAMETER("user")
```

и вставив его в раздел Summary шаблона отчета, можно из приложения передать в 19-ом параметре фамилию пользователя:

```
"user = Иванова"
```

- Для того, чтобы указать несколько таблиц и/или собственных параметров в девятнадцатом параметре, следует разделить их знаком ~п.

Если при вызове отчета вместо значения собственного параметра указать знак вопроса ("user = ?"), пользователь получит во время выполнения программы приглашение для ввода значения, оформленное как Dialog-box.

Пример вызова шаблона отчета с передачей пользовательского параметра:

```
RUN aderb\_printrb.p ("reports.prl", "report6", "univ.db", "", "", "", "D",
    "", "", "", 0, 1, 99, no, "", yes, yes, no, "user = ?").
```

ПРИЛОЖЕНИЕ С. Вопрос - ответ

1. Как установить настройки сеанса ProVISION?
2. С какими файлами работает Progress?
3. Какие функции для работы с операционной системой есть в Progress?
4. Как можно работать с системными диалоговыми окнами?
5. С какими указателями (Handles) работает Progress?
6. Как передается управление между объектами интерфейса?
7. Как можно работать с указателем мыши в Progress?
8. Каковы лимиты базы данных Progress?
9. Как можно организовать фоновый вычислительный процесс в приложении?

С.1. Как установить настройки сеанса ProVISION?

При старте сеанса Progress находит некоторые умалчиваемые характеристики в Windows Registry. Если есть необходимость изменить эти настройки, следует изменить содержимое файла Progress\bin\PROGRESS.INI и транслировать его программой Progress\bin\INI2REG.exe. В результате такой трансляции изменится соответствующая информация в Windows Registry.

Файл PROGRESS.INI разделен на секции. Ниже приведено краткое описание некоторых из них:

{Startup}	Здесь определяются переменные окружения, с которыми работает Progress. Вот некоторые из них: DLC - системная директория Progress. PROPATH - директории для программных файлов. PROMSG - файл с сообщениями об ошибках.
{Colors}	Здесь задается цветовая палитра.
{Default Window}	Здесь определяется размер и местоположение умалчиваемых окон сеанса.
{Fonts}	Здесь задается таблица используемых в сеансе шрифтов.
{ProADE}	Здесь указываются параметры для ADE.
{Proedit}	Установки для Procedure Editor.
{ProUIB}	Установки для UIB

Существует множество стартовых параметров Progress (их список приведен в документации), которые позволяют сконфигурировать сеанс. Они могут быть указаны, в частности, в файле Progress\startup.pf

Progress использует таблицы сортировок, хранящиеся в самой базе данных (используются при построении индексов) и в файле convmap.cp (используются для неиндексных сортировок). Если сортировки русских букв нехороши, следует:

- в первом случае выгрузить таблицы из базы (Data Administration → Admin → Dump Data and Definitions → Collation Tables), изменить их и загрузить обратно в базу (Data Administration → Admin → Load Data and Definitions → Data Definitions);

- подправить исходные таблицы в файле convmap.dat и транслировать его (в результате образуется правильный convmap.cp):

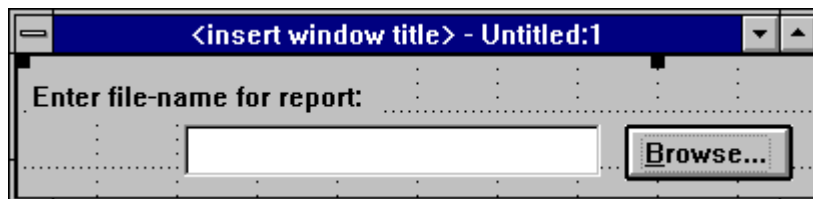
```
proutil db-name -C collation-compiler convmap.dat
```

С.2. Какие функции для работы с операционной системой есть в Progress?

OS-COMMAND	Выход в операционную систему и выполнение указанной в качестве параметра команды.
OS-COPY	Копирование файла.
OS-APPEND	Слияние двух файлов.
OS-RENAME	Переименование файла.
OS-DELETE	Удаление файла.
OS-CREATE-DIR	Создание директории.
OS-GETENV	Получение информации о переменной окружения.
OS-DRIVES	Получение списка системных устройств.
OS-ERROR	Получение информации об ошибке выполнения системной команды.

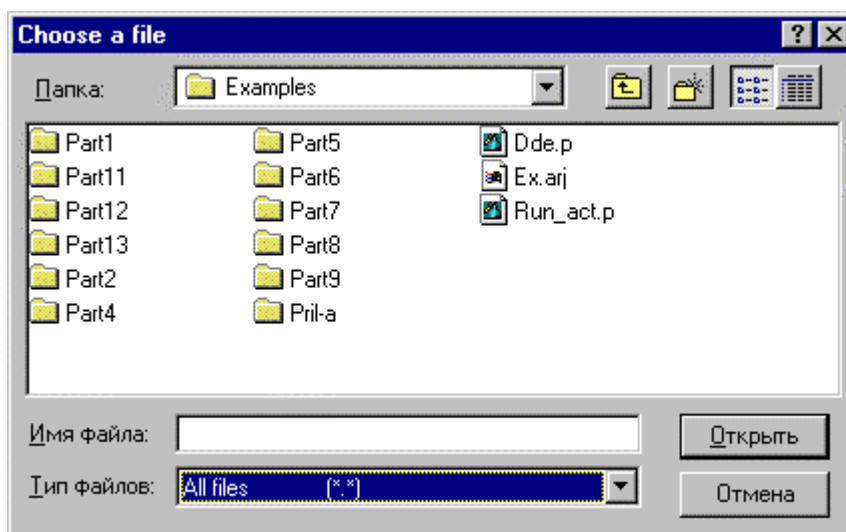
С.3. Как можно работать с системными диалоговыми окнами?

Операторы SYSTEM-DIALOG ... позволяют использовать в программах стандартные диалоговые окна для печати, выборки файла и т.д. Например, в следующем окне пользователю предлагается ввести имя некоторого файла:



Триггер для кнопки "Browse..." позволяет призвать к жизни системное окно выборки файла.

```
ON CHOOSE OF button-1 DO:
  DEFINE VARIABLE f-name AS CHARACTER.
  DEFINE VARIABLE ok-but AS LOGICAL.
  SYSTEM-DIALOG GET-FILE f-name
    TITLE "Choose a file"
    FILTERS "Files of students (*.st)" "*.st",
           "Files of courses (*.cr)" "*.cr",
           "All files (*.*)" "*. *"
    MUST-EXIST
    USE-FILENAME
    UPDATE ok-but.
  IF ok-but = YES THEN
    FILL-IN-1:SCREEN-VALUE IN FRAME DEFAULT-FRAME = f-name.
END.
```



С.4. С какими указателями (Handles) работает Progress?

При работе с объектами Progress удобно использовать указатели. Существует несколько категорий указателей:

Field-level handles	- для объектов интерфейса;
Event handles	- для событий;
Window handles	- для окон;
Procedure handles	- для процедур;
Session handles	- для текущего сеанса Progress;
Clipboard handles	- для системного CLIPBOARD;
File-info handles	- для файлов операционной системы;
Development Tool handles	- для инструментов среды разработчика (напр., Debugger);
Display format table handles	- для Progress-таблиц цветов и шрифтов.

Ниже перечислены некоторые из наиболее часто используемых системных переменных - указателей:

FOCUS (Field-level handle) - указывает на объект интерфейса, находящийся в состоянии "input focus";

SELF (Field-level handle) - указывает на объект интерфейса, по которому в данный момент выполняется триггер;

LAST-EVENT (Event handle) - указывает на последнее событие, случившееся в сеансе;

CURRENT-WINDOW (Window handle) - указывает на текущее окно;

DEFAULT-WINDOW (Window handle) - указывает на умалчиваемое окно;

ACTIV-WINDOW (Window handle) - указывает на активное окно;

SESSION (session handle) - указывает на текущий сеанс (например, можно использовать эту переменную для переопределения клавиши завершения ввода в fill-in поле с <tab> на <enter>:
SESSION:DATA-ENTRY-RETURN = TRUE.);

CLIBOARD (clipboard handle) - указывает на системный CLIBOARD;

FILE-INFO (file-info handle) - указывает на файл операционной системы.

С.5. Как передается управление между объектами интерфейса?

Умалчиваемая последовательность перехода от объекта к объекту в экранной форме определяется порядком перечисления их в операторе описания фрейма. UIB формирует этот перечень по местоположению объектов в контейнере: слева направо, сверху вниз. Изменить такой порядок можно через Edit → Tab Order.

Передать управление объекту интерфейса "вне очереди" можно с помощью программной генерации события ENTRY:

APPLY "ENTRY" TO <widget>

В операторе WAIT-FOR после ключевого слова FOCUS можно указать идентификатор объекта, который будет активным при актуализации окна.

С.6. Как можно работать с указателем мыши в Progress?

Метод LOAD-MOUSE-POINTER, применимый к большинству объектов интерфейса (в том числе контейнеров), позволяет изменять внешний вид указателя мыши при работе с этими объектами. В качестве параметра указывается строка. Некоторые из возможных этой строки:

ARROW - стандартный вид;
CROSS - крест;
WAIT - часы;
GLOVE - указательный палец.

С.7. Каковы лимиты базы данных Progress?

Областей размещения для базы данных	1,000
Экстентов для области размещения	255
Таблиц в базе	32,000
Таблиц в области размещения	32,000
Строк в таблице	2 миллиарда
Максимальный размер журнала Roll-Back	32,000 гигабайт
Максимальный размер журнала Roll-Forward	32,000 гигабайт
Индексов в базе	32,000
Индексов у таблицы	32,000
Столбцов в таблице	32,000
Столбцов (компонент) в индексе	16
Секвенций в базе	2,000
Пользователей	10,000
Параллельных транзакций	10,000
Минимальный размер блока	1024 байта
Максимальный размер блока	8192 байта
Максимальный размер записи	32,000 байт
Максимальный размер ключа	1,000 байт
Максимальная ширина столбца	32,000

Максимальное значение секвенции	2 ³¹ – 1
Минимальное значение секвенции	-2 ³¹
Максимальный размер области	16,000 гигабайт
Максимальный размер таблицы	16,000 гигабайт
Максимальный размер базы данных	16,000,000 гигабайт
Максимальное число аргументов в операторе SQL CALL	50
Максимальная длина оператора SQL	10,000 байт
Максимальная длина столбца фиксированной длины в таблице	2,000 байт
Максимальная длина спецификации значения по умолчанию	250 байт
Максимальная длина строки соединения (connect)	100 байт
Максимальная длина имени таблицы	32 символа
Максимальная длина имени пользователя в строке соединения	32 символа
Максимальная длина сообщения об ошибке	256 символов
Максимальное число столбцов в таблице	500
Максимальная длина правила целостности CHECK	240 байт
Максимальная вложенность SQL операторов	25
Максимальное число ссылок на таблицы в операторе SQL	250
Максимальный размер входных параметров для оператора SQL	512 байт
Максимальное число внешних ссылок в операторе SQL	25
Максимальное число вложенных уровней в представлениях	25
Максимальное число правил целостности CHECK в таблице	1,000
Максимальное число правил целостности FOREIGN в таблице	1,000

С.8. Как можно организовать фоновый вычислительный процесс в приложении?

Если появляется необходимость вести некоторые вычисления так, чтобы при этом пользователь продолжал работать с объектами интерфейса (например, вводить данные в поле), можно использовать оператор PROCESS EVENTS.