Eidgenössische
Technische Hochschule
Zürich

Department Informatik
Institut für
Computersysteme

H. Mössenböck
J. Темпл
P. Гризмер

Объект "Оберон"

Объектно-ориентированное расширение "Оберона"

Июнь 1989 г.

Адрес авторов:

Institut für Computersysteme
ETH-Zentrum
CH-8092 Zürich, Switzerland

электронная почта: moessenboeck@inf.ethz.ch

Аннотация

Object Oberon расширяет Оберон [1] за счет концепции классов, делая объектно-ориентированное программирование в Oberon более естественным. Добавленные функции оставляют Oberon практически неизменным и вызывают лишь незначительные накладные расходы на вызовы методов в объектном коде и во время выполнения.

Содержание

1. Мотивация

Язык Oberon [1] предоставляет концепцию расширения типов, которая является необходимым условием для объектно- ориентированного программирования. Сам Oberon намеренно не является полностью объектно-ориентированным языком, т. Е. Он не динамический допускает объявление классов и методами, которые могут быть переопределены в подклассах (известных как привязка). В этом отчете описывается небольшое языковое расширение Oberon, которое добавляет концепцию классов.

Причиной расширения стала потребность в простом языке, который можно было бы использовать в курсе по объектно-ориентированному программированию в ETH. Нашей целью было сохранить Oberon в качестве базового языка и внести в него как можно меньше изменений. Другой целью было определить язык, который минимально поддерживает объектно- ориентированное программирование. Таким образом, были включены только основные функции.

Чтобы продемонстрировать необходимость более удобного языкового механизма, мы сначала покажем, как объектно-ориентированное программирование может выполняться в обычном Oberon. Затем мы определяем расширение языка. Наконец, мы набросаем реализацию классов и приведем некоторые измерения. В приложении 1 кратко излагается грамматика Object Oberon, а в Приложении 3 описывается базовая библиотека классов, написанная на Object Oberon.

2. Объектно-ориентированное программирование в Oberon

В Oberon существуют различные возможности моделирования и объектно-ориентированного стиля программирования, хотя пока ни одна из них не была описана явно. в отчете Oberon даже не упоминается термин "объектно-ориентированное программирование".

Концепции объектно-ориентированного программирования соответствуют следующим концепциям Oberon: записи и Объекты являются переменными типа записи; переменные экземпляра объявлены ли поля записи и переменные процедуры как поля записи; наследование расширение записей; суперклассы являются базовыми типами и расширениями базового типа; Сообщение просто означает вызов соответствующей переменной процедуры.

Классы являются методы являются между классами соответствует типу подклассы отправляют

В принципе, есть два способа использовать Oberon для объектно-ориентированного программирования: (1) Все методы объявляются в записях как переменные процедуры; (2) каждый класс имеет единственную переменную процедуры (обработчик), который получает и распространяет сообщения, отправляемые объекту.

2.1 Классы с переменными процедуры в качестве методов.

Использование обычных процедур для доступа к данным в записи иногда бывает неудовлетворительным, поскольку такие процедуры вызываются явно и не могут быть переопределены для производного типа. Следовательно, такие системы на самом деле не являются расширяемыми. Лучший способ - реализовать операции как переменные процедуры и включить их в запись, чтобы на их место можно было установить новые процедуры для каждого производного типа. Например, класс stack может быть реализован следующим образом:

```
Stack = УКАЗАТЕЛЬ НА StackRec;
StackRec = ЗАПИСЬ
    s: МАССИВ 128 ЦЕЛЫХ ЧИСЕЛ;
    sp: ЦЕЛОЕ ЧИСЛО;
Push: ПРОЦЕДУРА (стек, ЦЕЛОЕ
ЧИСЛО); Pop: ПРОЦЕДУРА
(стек): ЦЕЛОЕ ЧИСЛО; КОНЕЦ;
```

Это естественный способ реализации классов в Oberon, но у него есть несколько недостатков:

- Методы занимают много места в каждой вещи. В глубокой иерархии классов 20 и более методов не необычно.
- Процедура переменные должны быть инициализированы каждый раз, когда создается новый объект. Это нудно и ошибка.
- Приемник сообщения (объект, данные которого будут манипулировать с помощью вызова метода должен быть передается как параметр каждого метода, чтобы сделать доступными личные данные объекта.

Опыт показал, что такого рода объектно-ориентированное программирование, как правило, скорее усложняет задачу, чем упрощает.

2.2 Классы с обработчиком сообщений

Ниже показано, как классы реализованы в системе Oberon [2], поэтому мы рассматриваем это как рекомендуемый способ написания объектно-ориентированных программ в Oberon: у каждого класса есть одна процедура переменная называется its обработчик, который получает сообщения, состоящие из идентификационного номера и некоторых параметров сообщения. Используя идентификацию сообщения, обработчик выполняет соответствующее действие (обычно это вызывает некоторую процедуру). Пример класса stack превращается в:

```
Message = ИДЕНТИФИКАТОР ЗАПИСИ: ЦЕЛОЕ
ЧИСЛО; val: КОНЕЦ ЦЕЛОГО ЧИСЛА; Stack =
УКАЗАТЕЛЬ НА StackDesc; StackDesc = ЗАПИСЬ
        s: МАССИВ 128 ЦЕЛЫХ ЧИСЕЛ;
        sp: ЦЕЛОЕ ЧИСЛО;
        дескриптор: ПРОЦЕДУРА (стек,
сообщение переменной) КОНЕЦ;
```

Здесь обработчик будет набросан следующим образом:

```
ПРОЦЕДУРА * Обработайте (stack:
стек; VAR msg: сообщение); НАЧНИТЕ
        РЕГИСТР msg.id ИЗ
0: INC(stack^.sp); stack^.s[stack^.sp] := сообщение.val      (* сообщение "Push" *)
1: сообщение.val := стек ^.s[stack^.sp]; DEC(stack^.sp) I    (* сообщение "Pop" *)
КОНЕЧНЫЙ
дескриптор КОНЦА;
```

Эта схема является гибкой, поскольку новые сообщения могут быть добавлены в класс без изменения интерфейса для клиентов. (Новое сообщение - это просто новый идентификационный номер.) Но у него также есть некоторые недостатки.:

- Это происходит медленно, потому что обработчик должен различать сообщения во время выполнения. Каждый вызов метода включает поиск сообщения обработчиком.
- Это трудно читать, поскольку объявление класса не дает никаких подсказок, какие сообщения могут быть отправлены объекту и какие параметры разрешены.
- Обработчик должен быть установлен явно для каждого созданного объекта.
- Принимающий объект все еще должен быть передан в качестве параметра сообщения.
- Отправка сообщения очень неуклюжа: она включает в себя настройку записи сообщения (идентификацию и другие параметры сообщения) и вызов обработчика.

Схема хорошо работает, когда объектов немного, а сообщения отправляются довольно редко. Но это неуместно, когда программы написаны в объектно-ориентированном стиле, где объекты и методы являются основными строительными блоками программы.

3. Языковой объект Oberon

Object Oberon основан на том, что Oberon добавил к нему концепцию классов. Oberon был разработан для написания расширяемых и эффективных систем. Это также является целью Object Oberon: классы помогают сделать системы еще более расширяемыми, и, как мы покажем в главе 5, они могут быть реализованы более эффективно, чем схемы, описанные в предыдущей главе.

В этой главе описывается, как классы встраиваются в Oberon. Это не полный языковой отчет. Все, что здесь опущено, определено так же, как и в отчете Oberon.

3.1 Лексические дополнения

Зарезервированные слова Oberon дополняются ключевым словом CLASS . Комментарии могут быть вложенными.

3.2 Классы

Класс - это абстрактный тип данных, состоящий из личных данных и операций, определенных над этими данными.

$ ClassDeclaration = ИДЕНТИФИКАТОР КЛАССА$_1$ [СуперКласс] ";" Идентификатор КОНЦА тела$_2$

$ Тело . = Методы полей [BEGIN StatementSequence].

является именем класса и должно соответствовать .$_2$ Класс представляет собой область видимости для всех имен, объявленных в Тело. Правила видимости для имен, объявленных в области видимости класса, такие же, как и для имен, объявленных в а идентификатор ident$_1$ область действия записи. Последовательность операторов в Тело выполняется всякий раз, когда выделяется объект этого класса (см. 3.3). Его можно использовать для инициализации переменных экземпляра объекта.

$ СуперКласс = "(" qualident ")".

Класс C может быть производным от другого класса C0, указанного через qualident . В объявлении

CLASS C (C0); ... END C;

C0 называется прямым суперклассом C, а C - прямым подклассом C0. Термин суперкласс (подкласс) означает прямой суперкласс (подкласс) или его суперкласс (подкласс). Подкласс наследует все переменные и методы экземпляра от своего суперкласса (ов).

$ Поля = Последовательность списков полей.

$ Методы = {ProcedureDeclaration ";"}.

Поля (также называемые переменными экземпляра) эквивалентны полям записи, а методы эквивалентны объявлениям процедур в Oberon.

Пример:

```
Средство просмотра КЛАССОВ (фрейм);
    режим: ЦЕЛОЕ ЧИСЛО;
    видимый: ЛОГИЧЕСКИЙ;

    ПРОЦЕДУРА открыта (x, y: ЦЕЛОЕ ЧИСЛО; режим: ЦЕЛОЕ ЧИСЛО); НАЧАТЬ ...
    ЗАВЕРШЕНИЕ открытия; Закрытие ПРОЦЕДУРЫ; НАЧАТЬ ... ЗАВЕРШЕНИЕ закрытия;
    Перемещение ПРОЦЕДУРЫ (x, y: ЦЕЛОЕ ЧИСЛО); НАЧАТЬ ... ЗАВЕРШЕНИЕ перемещения;

ВИДИМОЕ НАЧАЛО : = FALSE
КОНЕЧНЫЙ просмотрщик;
```

3.3 Объекты

к его полям могут обращаться объекты, являющиеся переменными, тип

просмотра, v.mode и v.visible . Его методы могут вызываться с помощью v.Open(100, 100, 0) и т.д. Как поля, так и методы могут использоваться без каких-либо ограничений в классе, который их объявляет или наследует.

Объекты должны быть выделены стандартной процедурой NEW, прежде чем их можно будет использовать. NEW выделяет память для полей объекта и вызывает выполнение последовательности инструкций в теле класса.

Как и в случае с указателями, не существует явного способа освободить объект.

Внутри методов заранее объявленная псевдопеременная SELF обозначает объект, для которого был вызван метод (получатель сообщения). SELF также может использоваться в телах классов для ссылки на вновь выделенный объект. Его нельзя использовать в качестве цели присваивания.

Хотя нет необходимости уточнять частные поля и методы с помощью SELF (т. Е. mode и SELF.mode эквивалентны в методе просмотра), иногда это может улучшить читаемость, особенно для унаследованных полей и методов, объявлений которых нет под рукой. Ссылка на SELF необходима только когда сам объект назначен какому-либо другому объекту.

Пример использования объектов:

```
VAR v1,
v2: Viewer; . . .
СОЗДАТЬ (v1); v1.Открыть (x0,
y0, режим); СОЗДАТЬ(v2);
v2.Открыть (x1, y1, режим); . . .
v1.Закрыть; v2.Закрыть;
```

3.4 Порядок объявления

Константы, типы, переменные, процедуры и классы могут быть объявлены в любой последовательности внутри блока. Классы могут быть объявлены только в блоке модуля.

```
$       Последовательность деклараций      = { CONST {ConstantDeclaration ";" }
$                                              | TYPE {TypeDeclaration ";"}
$                                              | VAR {VariableDeclaration ";"}
$                                              | ProcedureDeclaration ";" | ForwardDeclaration ";"
$                                              | ClassDeclaration ";" | ClassForwardDeclaration ";"}
```

Когда два класса ссылаются друг на друга рекурсивно, однопроходная компиляция требует прямого объявления (частей) класса.

```
$       ClassForwardDeclaration = CLASS " " идентификатор [СуперКласс] ";" КОНЕЧНЫЙ
$       идентификатор DefinitionBody. DefinitionBody = Заголовки методов полей.
$       MethodHeadings = {ProcedureHeading ";"}.
```

Поля или методы из фактического объявления класса могут быть опущены в предварительном объявлении. Однако поля и методы, объявленные в DefinitionBody должно встречаться как первое объявление в Тело из фактического объявления класса и должны отображаться в том же порядке. Типы полей и параметры списков методов должны быть идентичны в объявлении класса и в его прямом объявлении. Если класс имеет суперкласс, его имя должно быть указано в объявлении класса, а также в предварительном объявлении.

3.5 Суперклассы и подклассы

Подкласс наследует все поля и методы от своего суперкласса. Унаследованный метод можно переопределить, повторно объявив его в подклассе. Повторно объявленный метод должен иметь тот же список параметров, что и переопределенный метод. Переопределенные методы по-прежнему могут быть доступны псевдопеременной SUPER (см. Пример в конце этого раздела). Супер ссылкой в классе С обозначает самостоятельного объекта с бегом времени типа вынуждены суперкласс С. супер может быть использован только для квалификации методы. Поля не могут быть переопределены, поэтому их имена должны отличаться от имен полей во всех суперклассах.

Перед выполнением тела класса выполняется тело его суперкласса.

Правила совместимости между суперклассами и подклассами такие же, как между records и расширенными записями в Oberon. Учитывая два объявления класса

```
CLASS C0; . . . END CO;
КЛАСС C1(C0); . . . END C1;
```

объекты подкласса С1 могут быть присвоены или переданы в качестве параметра значения объектам суперкласса С0, но не наоборот. В случае переменных параметров фактический и формальный типы параметров должны быть идентичными. Учитывая следующие объявления

```
VAR c0: C0; c1: C1;
```

назначение

```
c0 := c1
```

приводит к тому, что c0 ссылается на тот же объект, что и c1. Статический тип c0 - C0, но его динамический тип - C1. Оба объекта имеют общие значения своих полей. Чтобы получить физическую копию объекта, необходимо написать

присвоение

c0^ := c1^

Здесь копируются только общие поля c0 и c1. должно быть, c0 уже выделен.
Он сохраняет динамический тип, который был у него до назначения.

Подводя итог, правила совместимости присвоения между объектами следующие:

(1) Объект класса C может быть присвоен объекту того же класса или суперклассу C
(2) Значение NIL может быть присвоено любому объекту
(подсказка: это можно использовать, чтобы пометить пространство объекта как свободное для сборщика мусора.) (3) Псевдопеременные SELF и SUPER не должны быть целью назначения.

Type test, type guard с помощью операторы определяются для объектов и классов аналогично и записей в Oberon.

Пример для подклассов и СУПЕРКЛАССОВ:

```
Стек КЛАССОВ;
    s: МАССИВ 128 ЦЕЛЫХ чисел;
    sp: ЦЕЛОЕ ЧИСЛО;
ПРОЦЕДУРА Push (x: ЦЕЛОЕ ЧИСЛО); BEGIN . . . END Push;
ПРОЦЕДУРА Pop (): ЦЕЛОЕ ЧИСЛО; BEGIN . . . END Pop;
BEGIN sp := 0
КОНЕЧНЫЙ стек;

КОЛИЧЕСТВО КЛАССОВ (Stack);
    max: ЦЕЛОЕ ЧИСЛО;

    ПРОЦЕДУРА Push (x:                          (* переопределяет унаследованный Push *)
    ЦЕЛОЕ ЧИСЛО); НАЧАТЬ
        SUPER.Push(x); ЕСЛИ sp > max, ТО
    max := sp END КОНЕЧНОЕ нажатие;

НАЧАЛО max := sp                    (* тело стека выполняется перед этим телом *)
КОНЕЧНЫЙ счетчик;
```

Пример совместимости классов:

```
VAR s0: Stack; s1: CountStack;

НОВЫЙ(s0); NEW(s1);
s0 := s1; . . .
ЕСЛИ s0 - ЭТО счетчик, ТО s1 := s0 (счетчик) END
```

3.6 Экспорт классов

Экспортируются классы, объявленные в части определения модуля.

```
$   ClassDefinition                    = ИДЕНТИФИКАТОР КЛАССА [СуперКласс] ";" Идентификатор КОНЦА DefinitionBody.
```

Поля или методы, объявленные в части реализации, могут быть опущены в части определения.
Клиентам модуля видны только имена, объявленные в части определения.

Типы полей и списки параметров методов должны быть идентичны в обоих объявлениях. Если класс
имеет суперкласс, его имя должно быть указано в обоих объявлениях классов.

Примечание по реализации.: Поля и методы, объявленные в DefinitionBody (см. Главу 3.4), должны встречаться в качестве первых объявлений в соответствующей реализации Тело и должны отображаться в том же порядке, в противном случае клиенты модуля должны быть перекомпилированы.

Последовательность определений в части определения модуля следующая:

```
$   Последовательность деклараций   = { CONST {ConstantDeclaration ";" }
$                                       I TYPE {TypeDeclaration ";"}
$                                       I VAR {VariableDeclaration ";"}
$                                       I ProcedureDeclaration ";"          [так в оригинале: ProcedureHeading]
$                                       I ClassDefinition ";" I ClassForwardDeclaration ";"}
```

В частях определения нет прямых объявлений процедур, но прямые объявления классов все еще могут быть необходимы, поскольку каждый из двух классов может иметь поля, являющиеся объектами другого класса.

3.7 Переменные сообщения.

В некоторых ситуациях необходимо отправить сообщение не только одному объекту, но и всем объектам в структуре данных. Обычным решением было бы обойти структуру данных и вызвать соответствующий метод для каждого встреченного объекта. Однако это неудобно, поскольку алгоритм обхода приходится повторять снова и снова. Кроме того, структура данных может быть скрыта и, следовательно, недоступна для обхода. Более элегантным решением является передача сообщения в качестве параметра общей процедуре обхода, которая отправляет его каждому объекту в структуре данных.

Мы определяем новый стандартный тип MESSAGE . Переменные этого типа могут предположить константы сообщения в качестве их значений. Постоянное сообщение-это имя, сообщение, квалифицированное имя класса, из которых получение объекты должны быть. У него также есть актуальный список параметров. Он называется а постоянная поскольку сообщение не отправляется немедленно, а сохраняется целиком в переменной message для отправки позже. Синтаксически константа сообщения является фактором:

```
$   коэффициент          = . . . IMessageConst.
$   MessageConst         = имя_класса "."
$   имя_класса           Имяметода [Фактические параметры]. =
$   имя_ метода          qualident. = ident.
```

Имяметода не должно обозначать функцию. обозначает класс, из которого должен быть получен объект. Имя_класса для получения этого сообщения (оно явно не выбирает метод из этого конкретного класса.) Константы сообщения могут использоваться только как параметры значений или присваиваться локальным переменным. Если сообщение

затем переменной присваивается другая переменная message, и она должна быть объявлена в той же области видимости.abab

Для отправки сообщения объекту существуют две стандартные процедуры:

SEND(x,m)	должен обозначать объект и быть переменной типа MESSAGE. x Объекту отправляется сообщение с фактическими параметрами, x указанными в соответствующей константе message . Класс должен быть равен,,,(или x быть подклассом) класса, указанного параметром Имя_класса в сообщении константа. Сообщения не должны отправляться в SUPER.
ПРИНИМАЕТ (x,m)	должен обозначать объект и должен быть переменной типа MESSAGE. x Функция ПРИНИМАЕТ возвращает TRUE , если "понимает" сообщение , т. m е. если класс of равен (или является подклассом) классу, указанному через className в константе сообщения , соответствующей . m

Пример

В системе Oberon сообщения могут использоваться для реализации трансляции всем зрителям на экране. Чтобы показать удаление фрагмента текста во всех телезрителях, содержащих этот текст, можно написать:

```
Число просмотров.Трансляция (TFrames.Удалить(начать, завершить), текст)
```

Затем трансляция процедуры будет реализована следующим образом:

```
ТРАНСЛЯЦИЯ ПРОЦЕДУРЫ (СООБЩЕНИЕ msg: MESSAGE; текст: Txt.Text);
    ПАРАМЕТР v: Viewer;
frame: Кадры.Frame; НАЧАТЬ
    v := firstViewer;
    ПОКА v # НИЧЕГО НЕ ДЕЛАЕТ (* для всех зрителей на экране *)
        frame := v.вниз;
        ПОКА кадр # НЕ ВЫПОЛНЯЕТСЯ (* для всех подкадров *)
            ЕСЛИ ПРИНИМАЕТ (кадр, сообщение) & (кадр.текст = текст) ТОГДА
                ОТПРАВИТЬ (кадр,
            сообщение) КОНЕЦ; кадр
            := кадр.следующий
КОНЕЦ; v := v.
следующий;
КОНЕЦ; ЗАВЕРШИТЬ
трансляцию;
```

3.8 Функции, доступные в других объектно-ориентированных языках

Object Oberon содержит лишь минимальный набор функций для объектно-ориентированного программирования. Другие языки, такие как Smalltalk [3] или C ++ [4], вводят больше концепций, хотя некоторые из них также могут быть реализованы в Object Oberon.

Статическое связывание по сравнению с динамическим.

C ++ различает статическую и динамическую привязку сообщений и методов. В Object Oberon все

методы динамически привязаны, т. е. Они берутся из класса, к которому принадлежит объект во время выполнения (который может быть подклассом класса, с помощью которого был объявлен объект). Это отличается от статической привязки, где методы всегда берутся из класса, с помощью которого был объявлен объект. Статически привязанные методы нельзя переопределять в подклассах. В общем случае вызов статически связанного метода более эффективен, чем вызов динамически связанного метода.

В Object Oberon статическая привязка может быть достигнута путем использования обычных процедур, объявленных вне класса вместо методов. Эти процедуры должны получать объект, на который они ссылаются, в качестве параметра, чтобы иметь возможность доступа к полям объекта (явное "self").

Переменные класса и методы класса

Smalltalk вводит концепцию переменных класса и методов класса. Такие элементы принадлежат классу , а не объекту этого класса. Переменные класса могут использоваться для хранения значений, необходимых всем объектам класса. Методы класса обычно используются для создания и инициализации объектов. По этой причине методы класса не могут принадлежать объекту, поскольку они должны быть вызваны до того, как объект будет существовать.

В Object Oberon переменная класса может быть реализована как глобальная переменная модуля, содержащего класс, а метод класса - как глобальная процедура. Если рассматривать не класс, а модуль как фактический барьер для клиентов, скрытие информации сохраняется.

4. Реализация

В этом разделе описываются решения по реализации, в основном представление объектов и классов во время выполнения и код, сгенерированный для операций с объектами.

Object Oberon был реализован в Oberon на рабочей станции Ceres [5]. В качестве отправной точки мы использовали оригинальный компилятор Oberon, написанный Н. Виртом. Объект Oberon является надмножеством Oberon. Каждая программа Oberon также может быть скомпилирована объектным компилятором Oberon.

4.1 Представление объектов и классов во время выполнения.

Объекты реализуются как указатели на записи, содержащие переменные экземпляра, тег типа и указатель на таблицу методов. Для каждого класса существует одна такая таблица методов, на которую ссылаются все объекты этого класса (см. рис. 1).

Дескриптор типа уже присутствует в реализации записей Oberon и может быть позаимствован для объектов. Он используется сборщиком мусора и для динамических тестов типов. Мы не трогали информацию в дескрипторах типов. Для классов мы добавили таблицу методов, которая реализована в виде массива переменных процедуры. Она индексируется номером метода. Каждая запись содержит дескриптор процедуры (указатель модуля и относительный программный счетчик).

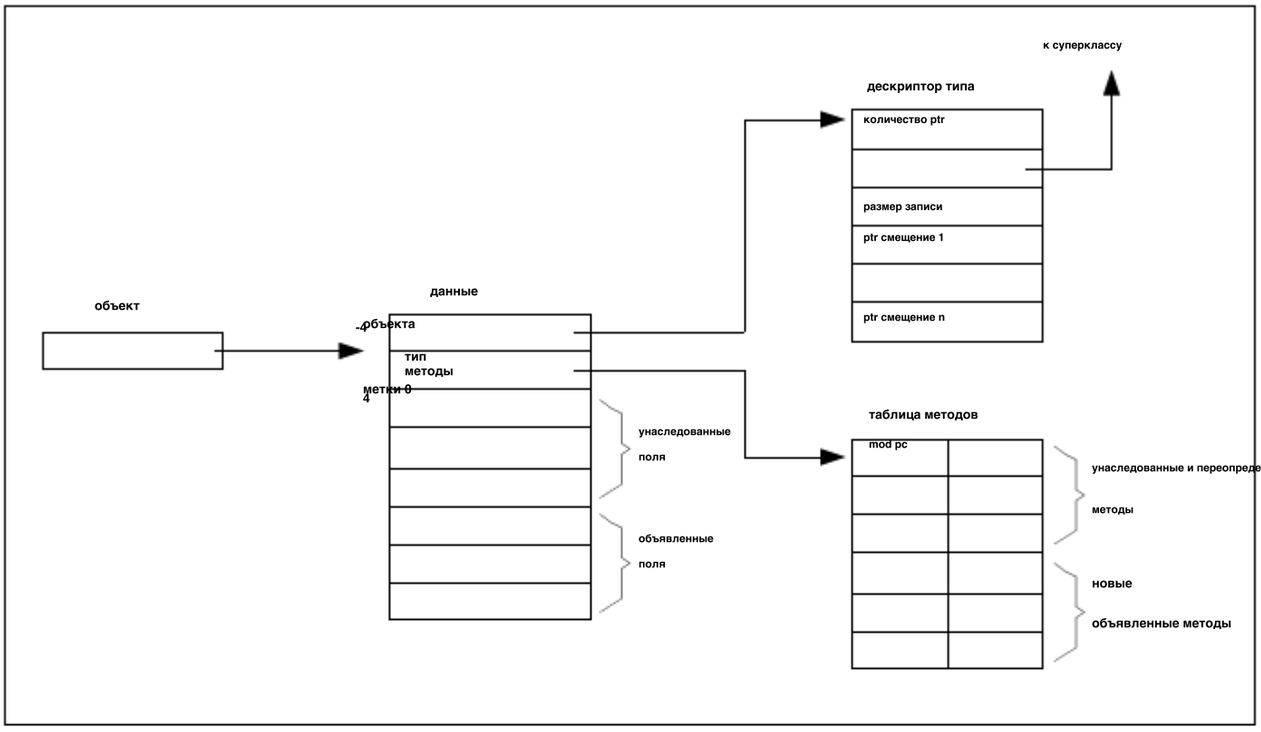


Рис. 1. Представление объектов и классов во время выполнения в Object Oberon

Альтернативой отдельной таблице методов было бы сохранение таблицы методов с дескриптором типа. Это позволило бы сохранить дополнительный указатель на таблицу методов в каждом объекте. Естественным решением было бы получить доступ к методам с отрицательными смещениями относительно дескриптора типа. Но это вызвало бы серьезные изменения в распределении и обработке дескрипторов. Итак, мы рассмотрели возможность добавления метода table в конец дескриптора типа. Но, к сожалению, здесь тоже есть загвоздка: смещения унаследованных методов к началу дескриптора типа должны быть одинаковыми в суперклассе и в подклассе. Но это не может быть гарантировано, потому что в подклассе могут быть объявлены новые указатели, что приведет к росту дескриптора его типа . Решением было бы получить длину дескриптора типа из количества указателей, хранящихся в нем, и использовать это как смещение всякий раз, когда вызывается метод (это смещение неизвестно во время компиляции!). Это привело бы к следующему коду для вызова метода:

```
. . .                                          objectR7
MOVD MOVXBD   -4(R7), R7                       =R7=адрес дескриптора
              0(R7), R6                        R6:=тип номер. методов, вызывающих
CXPD          8+m(R7)[R6:W]                    метод m
```

Используя отдельную таблицу методов, мы можем избежать инструкции MOVXBD и регистра индекса в CXPD, экономя 4 байта и 2 обращения к памяти для каждого вызова метода. Еще одно преимущество заключается в том, что нам не нужно полагаться на формат дескриптора типа и, таким образом, на нас не повлияют будущие изменения. Стоимость этого составляет накладные расходы в размере 4 байт на каждый объект. Однако столько же потребуется и при установке обработчика, как в объектах системы Oberon. Кроме того, мы считаем, что объекты следует использовать не для небольших объектов, а только для достаточно сложных данных с разумными операциями над ними. Для таких объектов небольшие накладные расходы на хранение должны быть незначительными.

4.2 Выделение таблицы методов

Таблица методов класса настраивается во время инициализации модуля, содержащего объявление класса . Она создается с помощью следующих шагов:

(1) Выделите таблицу методов в куче;

(2) Скопируйте дескрипторы унаследованных методов из таблицы методов суперкласса;

(3) Загрузите записи новых методов из таблицы ссылок процессора (см. [2]).

Это составляет от 10 до 35 байт (в зависимости от суперкласса) плюс 5 байт на каждый объявленный метод. Указатель на таблицу методов хранится в области констант сразу за адресом дескриптора типа этого класса. Всякий раз, когда создается объект, этот указатель должен быть установлен в него.

4.3 Создание объекта

Когда объект создается с помощью стандартной процедуры NEW, выполняются следующие действия:

(1) Выделите объектному пространству в куче;

(2) Установите указатель на таблицу методов в объекте; (3) Вызовите тело класса объекта.

Это составляет 19 байт.

4.4 Вызов метода

Методы подобны переменным процедуры. Таблица методов индексируется номером метода, а найденный в ней дескриптор процедуры принимается в качестве аргумента инструкции вызова.

получатель сообщения неявно передается в качестве первого параметра вызова метода.

MOVD	объект		3
MOVD	(FP), R7 R7, TOS	Нажимной приемник	2
MOVD	0(R7), R7	R7 := адрес таблицы методов	3
Параметры отправки			
CXPD m(R7)		Вызов метода m	3

Числа справа указывают длину инструкций: для вызова метода требуется 11 байт (без передачи параметра). Это на 5 байт меньше, чем для вызова обработчика в Oberon.:

MOVQW	0, идентификатор (FP)	msg.id := 0	3
Настройка параметров в			
msg MOVD	объект (FP), TOS	Нажимной приемник	3
MOVD ДОБАВЛЕНИЕ	Дескриптор, TOS	Введите информацию о сообщении	3
сообщения (FP),		сообщения адресе сообщения	3
TOS CXPD	дескриптор (объект (FP))	сообщения объекта вызова ^.дескриптор	4

Для вызова метода с использованием псевдопеременной SUPER может быть сгенерирован еще более короткий код, поскольку местоположение таблицы методов известно во время компиляции и его не нужно извлекать из объекта.

[illegible]		Нажимной приемник	3
[illegible]		R7 := адрес таблицы методов суперкласса	3
Параметры толчка			
CXPD	m(R7)	Вызов метода m	3

4.5 Переменные сообщения и константы сообщений

когда возникает константа сообщения, ее фактические параметры помещаются в стек процедур и создается дескриптор сообщения следующего вида.

Адрес фактических	4
параметров Адрес дескриптора класса	4
Размер фактических	4
параметров номер метода	4

Каждая переменная сообщения представлена таким дескриптором. Когда сообщение, хранящееся в переменной message, отправляется объекту, параметры копируются в верхнюю часть стека процедур и вызывается метод с сохраненным номером. Дескриптор класса используется для проверки типа, чтобы определить, понимает ли объект-получатель сообщение.

5. Измерения

Поскольку компилятор Object Oberon был производным от компилятора Oberon, имеет смысл сравнить размеры двух компиляторов, чтобы определить стоимость расширения. Изменения в основном затрагивают анализатор, обработчик таблиц (OCT) и части генератора кода (OCC и OCH). Исходный код увеличился с 3956 до 4429 строк (11,9%), объектный код - с 42184 до 47504 байт (12,6%). На рис. 2 показаны затраты на расширение в нескольких модулях компилятора.

Также интересно посмотреть на затраты на использование классов, т. е. Каковы временные затраты на вызов метода по сравнению с вызовами других процедур. Чтобы сделать вызовы процедур сравнимыми с вызовами методов, мы передали указатель в качестве параметра процедурам таким же образом, как это делается для методов. Это реально, поскольку процедуры, работающие с абстрактными данными, обычно получают указатель на данные в качестве параметра (например, процедуры, работающие с текстами в системе Oberon). Мы измерили вызов локальной процедуры

ПРОЦЕДУРА P0 (obj: Ptr); НАЧАЛО ЗАВЕРШЕНИЯ P0;

вызов внешней процедуры

ПРОЦЕДУРА * P1 (obj: Ptr); НАЧАЛО ЗАВЕРШЕНИЯ P1;

вызов обработчика (следующее является оптимистичным предположением для обработчика. Обычный обработчик намного дороже: он содержит тесты типов, вложенные операторы if и т.д.)

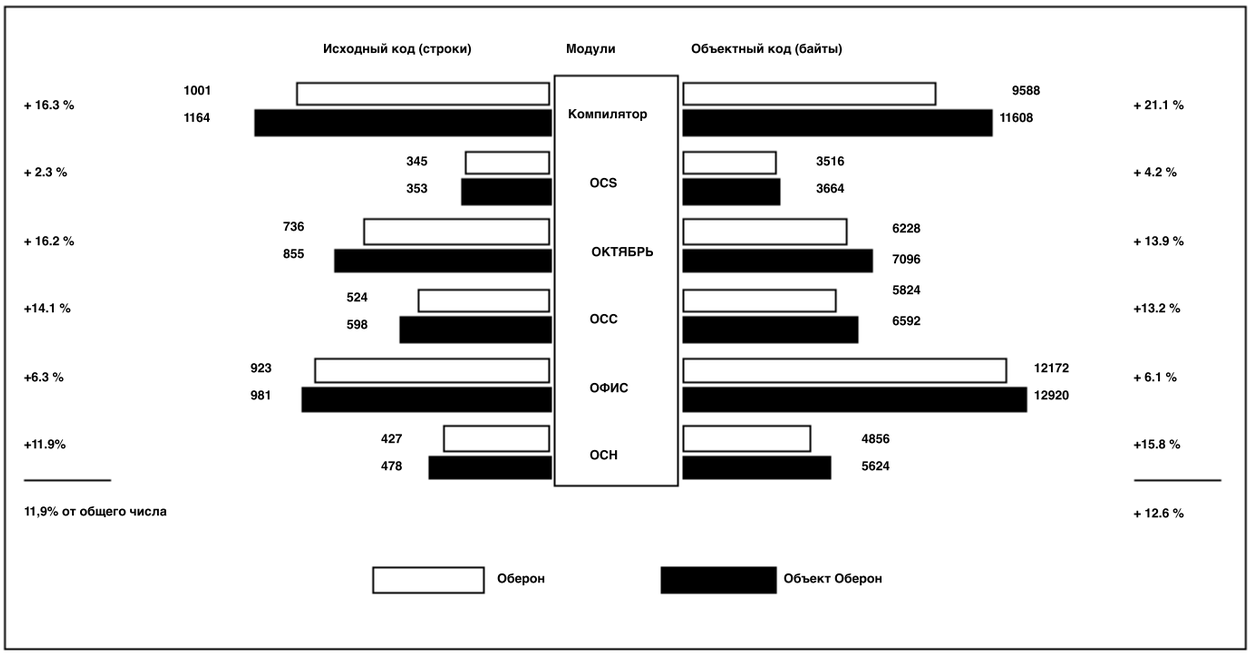


Рис. 2 Размер компилятора Oberon и объекта Oberon compiler

```
ПРОЦЕДУРА * Дескриптор (obj:
Ptr; VAR msg: Message); НАЧАЛО
    РЕГИСТР msg.id 0: КОНЕЦ
КОНЕЧНЫЙ дескриптор;
```

и вызов метода (принимающий объект передается как неявный параметр)

```
КЛАСС C;
    ПРОЦЕДУРА P; BEGIN
END P; END C;
```

Объявления для вышеуказанных процедур следующие

```
Тип
    Message = ИДЕНТИФИКАТОР ЗАПИСИ:
    КОНЕЦ ЦЕЛОГО ЧИСЛА; Ptr =
    УКАЗАТЕЛЬ На Rec; Rec = ЗАПИСЬ
        дескриптор:
    ПРОЦЕДУРА (Ptr, сообщение VAR) КОНЕЦ;
```

Мы вызывали каждую процедуру миллион раз и извлекли время для одного вызова. В следующей таблице показаны результаты.:

Вызов локальной процедуры	
Вызов внешней	μμ2,29 с
процедуры Вызов	μ3,69 с
обработчика Вызов метода	μ6.13 с
	3,96 с

Основным результатом является то, что вызов метода выполняется на 7,5% медленнее, чем вызов внешней процедуры, тогда как вызов обработчика выполняется на 66% медленнее. Это говорит о том, что при интенсивном использовании объектов программирование с классами и методами явно эффективнее, чем с записями и установленными обработчиками.

6. Обсуждение

Класс - это нечто среднее между модулем и записью. Он похож на модуль, потому что инкапсулирует данные и операции. Он также похож на запись, потому что содержит структурированные данные и может использоваться как тип данных для переменных. Классы в Object Oberon отражают оба сходства. Синтаксически они напоминают модули, семантически они похожи на записи (например, правила области видимости).

Классы и записи имеют много общего. Фактически, можно было бы подумать о том, чтобы пожертвовать записями (по крайней мере, расширяемыми) и рассматривать их как особые случаи классов без методов. Можно возразить, что объекты всегда размещаются в куче, сохраняя сборщик мусора занятым, тогда как записи могут быть размещены в стеке процедур. Но в системе Oberon записи, которые играют центральную роль в структуре данных, в любом случае размещаются в куче, а записи, которые лучше разместить в стеке процедур, появляются не часто (исключением являются записи сообщений в текущей системе Oberon. Однако, когда доступны классы и методы, записи сообщений больше не нужны).

По разным причинам программирование с помощью классов удобнее, чем программирование с записями и обработчиками, как в системе Oberon: • Обозначения вызовов методов лаконичны. Запись сообщения не должна инициализироваться перед вызовом, и целевой объект не обязательно передавать методу явно.

• Поиск метода, то есть инициализацию таблицы методов, не нужно программировать вручную. Это делает программы короче и безопаснее.

• Объекты инициализируются автоматически при создании. Это служит как удобству, так и безопасности.

• Из объявления класса очевидно, какие операции определены для класса. Это не так однозначно когда используется обработчик, поскольку соответствие между номерами сообщений и методами скрыто в части реализации модуля. Случайно ввести неправильный номер сообщения легко, особенно после того, как были вставлены новые номера сообщений.

С другой стороны, использование обработчиков вместо методов имело следующие преимущества. • Языковое расширение не требуется. • Если вводятся новые сообщения, интерфейс для клиентов не изменяется (хотя это также может быть расценено как недостаток!).

• Обработчику могут быть отправлены сообщения, которые не соответствуют ни одному методу. Это может быть ошибка или желаемый эффект. Обработчик может передать сообщение какому-либо другому обработчику или просто ничего не делать (но это может привести к тому, что ошибки останутся незамеченными).

• Легко реализовать широковещательную передачу любого сообщения любому объекту. Обработчики и записи сообщений являются низкоуровневыми конструкциями для объектно-ориентированного программирования. Они более гибкие по той же причине, по которой низкоуровневые средства в Modula-2 более гибкие, чем обычные языковые конструкции. Используя обработчики, компьютер не может проверить, разрешено ли отправлять сообщение объекту и были ли предоставлены все необходимые параметры. Таким образом, во время выполнения могут возникать ошибки, которые могли бы быть обнаружены во время компиляции, если бы использовались классы.

Введение классов не привело к резкому увеличению размера компилятора (на 12%). Хранилище

накладные расходы для каждого объекта такие же, как и для обработчика в записи Oberon (4 байта. Вызов метода всего на 7% медленнее обычного вызова процедуры, но явно эффективнее вызова обработчика как по размеру кода (45%), так и по времени выполнения (55%).

Object Oberon - это попытка извлечь максимум из минимального набора новых конструкций. Это в духе Oberon и в отличие от многих других объектно-ориентированных языков, таких как Smalltalk или C ++. Новые конструкции представляют собой классы и переменные сообщения. В то время как классы необходимы для объектно-ориентированного программирования, переменные message - нет. Их следует рассматривать как эксперимент по предоставлению эффективного и гибкого механизма трансляции сообщений, который недоступен в большинстве объектно-ориентированных языков, за исключением Smalltalk.

Ссылки

[1] Вирт Н.: Язык программирования Оберон. Программное обеспечение-практика и опыт, 18 (1988)

[2] Вирт Н., Гуткнехт Дж.: Система Оберон. Report 88, ETH Zürich, 1988

[3] Голдберг А., Робсон Д.: Smalltalk-80, Язык и его реализация. Аддисон-Уэсли, 1983.

[4] Страуструп Б.: Язык программирования C ++. Аддисон-Уэсли, 1986

Разработка и анализ компьютера для рабочей станции.[5] Eberle H.: Докторская диссертация, ETH Zürich, 1987

Приложение 1: Грамматика Object Oberon

Расширения выделены жирным шрифтом.

Oberon = ИДЕНТИФИКАТОР МОДУЛЯ ";" [ImportList] DeclSeq [BEGIN StatSeq] END ident
"." | ИДЕНТИФИКАТОР ОПРЕДЕЛЕНИЯ ";" [ImportList] КОНЕЧНЫЙ

ImportList идентификатор DefSeq ".". = IMPORT ident [":" ident] {", ident [":" ident]} ";".

DefSeq = { CONST {ConstDecl} | ТИП {TypeDecl} | VAR {VarDecl} | Проверка | **ClassDef** }.

DeclSeq = { CONST {ConstDecl} | TYPE {TypeDecl} | VAR {VarDecl} | ProcDecl | **ClassDecl** }.

ConstDecl = ident "="ConstExpr ";".

TypeDecl = ident "=" Тип";".

VarDecl = idList ":" Тип";".

Обработка = ИДЕНТИФИКАТОР ПРОЦЕДУРЫ [FormPars] ";".

ProcDecl = PROCEDURE "^" идентификатор [FormPars] ";"
| PROCEDURE ["^"] ident [FormPars] ";" DeclSeq [BEGIN StatSeq] END ident ";".

ClassDef **= ИДЕНТИФИКАТОР КЛАССА ["^"] ["("Qualident")"]";"**
FieldListSeq {Проверка} КОНЕЧНЫЙ

ClassDecl **идентификатор ";". = CLASS ["^"] ident ["(" Qualident")"]";"**
FieldListSeq {ProcDecl} [BEGIN StatSeq] КОНЕЧНЫЙ

FormPars **идентификатор ";".** = "(" [FPSection {";" FPSection)}]")" [":" Qualident].

FPSection = [VAR] idList ":" FormalType.

FormalType = {МАССИВ} Qualident.

Тип = Qualident
| МАССИВ ConstExpr { ","ConstExpr} типа
| ЗАПИСЬ ["(" Qualident ")"] FieldListSeq END
| УКАЗАТЕЛЬ НА тип
| ПРОЦЕДУРА ["(" [[VAR] FormalType {"," [VAR FormalType}]]")" [":" Квалидент]].

FieldListSeq = FieldList {";" Список полей}.

Список = [idList ":" Тип].

полей idList = ident {"," ident}.

ConstExpr = Выражение.

Выражение = SimExpr [("=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS) SimExpr].

SimExpr = ["+"|"-"] Термин {("+"|"-"|ИЛИ) Термин}.

Term = Коэффициент {("*"|"/"|DIV|MOD)|"&") Factor}. = число

Factor | строка | НОЛЬ | Набор | Обозначение [ActPars] | "(" Выражение")" | "~" Factor.

Set = "{" [Элемент {","Elem}]"}".

Elem = Выражение [".." Выражение].

ActPars = "(" [Выражение {","Выражение}]")".

StatSeq = Stat {";" Stat}.

Stat = [Обозначение {":=" Выражение | [Действующие лица])
IF Expr THEN StatSeq {ELSIF Expr THEN StatSeq} [ELSE StatSeq] END
|РЕГИСТРОВОЕ выражение Case {"|" Case} [ELSE StatSeq] END
| WHILE Expr ЗАВЕРШАЕТ ВЫПОЛНЕНИЕ StatSeq
| ПОВТОРЯТЬ StatSeq ДО ТЕХ ПОР, ПОКА Expr
| ЗАВЕРШАЕТ ЦИКЛ StatSeq
| НЕ ВЕРНЕТ [Выражение]
| | С СООТВЕТСТВУЮЩИМ
ЗАВЕРШАЕТ
значением ":" Qualident ЗАВЕРШАЕТ ВЫПОЛНЕНИЕ StatSeq
].

Кейс = [CaseLabels { ","CaseLabels}":"StatSeq].

Квалификационные метки Обозначение = ConstExpr [".." ConstExpr].
= ident {"." ident | "[" Expr {"," Expr} "]" | "^" | "(" Qualident")"}.
= ident ["." ident].

Приложение 2: Новые сообщения об ошибках компилятора 150

Идентификатор не соответствует
классы должны быть объявлены на уровне модуля
имени класса 151
152
базовый тип не
неразрешенное прямое определение класса
является классом 153
154
неправильный порядок методов между объявлением класса и прямым
объявлением 155
множественные определения
пересылки классов 156 константа сообщения
не должна обозначать функцию

Приложение 3: Базовая библиотека классов для объекта Oberon

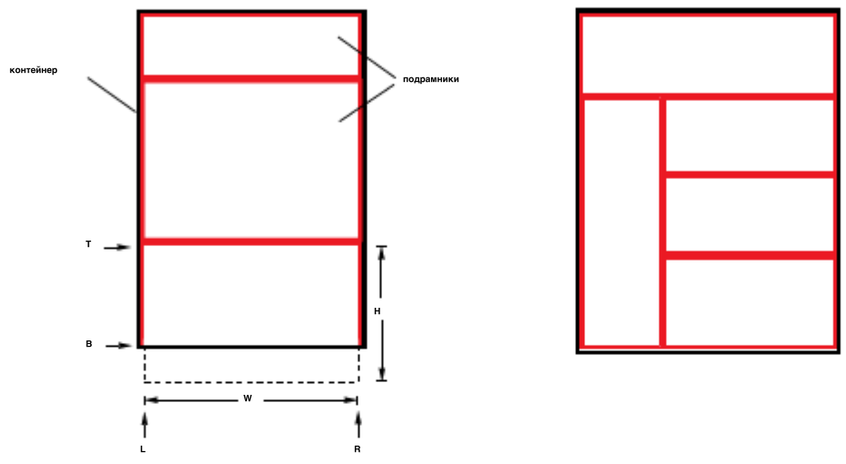
Это базовая библиотека классов для использования в программах Object Oberon. Она поддерживает средства просмотра, различные виды фреймов, текста и графики. Следующие модули основаны на оригинальных системных модулях Oberon, таких как дисплей, средства просмотра и тексты. В основном это делается для сохранения совместимости с существующей системой Oberon. На этом примере мы надеемся продемонстрировать полезность class construct при создании расширяемых программных модулей.

Фреймы

Фреймы - очень часто используемый класс. Каждая прямоугольная область на экране является производной от рамки: средства просмотра, меню, строки заголовков и даже сам экран являются рамкой. Фрейм - это контейнер для текста, графики или других фреймов.

```
ФРЕЙМЫ ОПРЕДЕЛЕНИЯ;
    ИМПОРТИРУЙТЕ
    объекты, файлы; CONST
        по вертикали = 0; по горизонтали = 1; относительно = 2; (* стратегии распределения подкадров для SetLT *)

    ФРЕЙМ КЛАССА (Объекты.Object);
        L, T, R, B: ЦЕЛОЕ ЧИСЛО;                    (* абсолютные координаты кадра: слева, сверху, справа,
        W, H: ЦЕЛОЕ ЧИСЛО;                          снизу *) (* желаемая ширина и высота; по умолчанию:
        вверх, вниз, далее: кадр;                   как можно больше *) (* рамка контейнера, первый
        видимый: ЛОГИЧЕСКИЙ;                        подкадр, следующий кадр *) (* TRUE, если виден (частично) *)
        УСТАНОВКА ПРОЦЕДУРЫ (f: фрейм);
        Удаление ПРОЦЕДУРЫ (f: фрейм);
        НАСТРОЙКА ПРОЦЕДУРЫ (f: фрейм; стратегия: SHORTINT);

        ОТОБРАЖЕНИЕ ПРОЦЕДУРЫ;
        Скрытие ПРОЦЕДУРЫ;
        Изменение размера ПРОЦЕДУРЫ (r, b: ЦЕЛОЕ ЧИСЛО);
        Перемещение ПРОЦЕДУРЫ (dx, dy: ЦЕЛОЕ ЧИСЛО);
        Копирование ПРОЦЕДУРЫ (ПЕРЕМЕННАЯ f: кадр);

        Расфокусировка ПРОЦЕДУРЫ;
        Нейтрализация ПРОЦЕДУРЫ;
        Клавиша обработки ПРОЦЕДУРЫ (ch: CHAR);
        МЫШЬ обработки ПРОЦЕДУРЫ (x, y: ЦЕЛОЕ ЧИСЛО; кнопки: УСТАНОВИТЬ);

        ПРОЦЕДУРА This(x, y: ЦЕЛОЕ ЧИСЛО): кадр;
        Трансляция ПРОЦЕДУРЫ (m: СООБЩЕНИЕ);
        Загрузка ПРОЦЕДУРЫ (VAR r: Файлы.Rider);
        Хранилище ПРОЦЕДУР (VAR r: Файлы.Rider);
    КОНЕЧНЫЙ фрейм;

    VAR
        Экран: Рамка;                               (* весь экран *) (* текущая
        Фокус: Рамка;                               рамка фокусировки *)

ПРОЦЕДУРА SetFocus(f: Кадр);
ПРОЦЕДУРА RemoveMarks(f: кадр);
ЗАВЕРШИТЕ кадры.
```

Каждый фрейм имеет физический и логический размер. Физический размер (R-L, B-T) - это размер кадра в том виде, в каком он отображается на экране. Логический размер (W, H) определяет максимальный размер, который может принимать фрейм, если фрейм контейнера достаточно велик. Он может увеличивать физический размер вправо и вниз (рис. а). Более сложный пример для вложенных фреймов показан на рис. б.



(a) (b)

Один из кадров является рамкой фокусировки, которая принимает символы, введенные с клавиатуры. Это должен быть подкадр текущего средства просмотра фокуса. Программист несет ответственность за соответствующую настройку рамки фокусировки .

Большинство из следующих методов вызываются системой автоматически (например, Show вызывается при установке фрейма). Их основное назначение - поддерживать согласованность данных фрейма и распространять сообщения на подфреймы. В подклассах методы Frames обычно должны быть переопределены (расширены).

устанавливает рамку в качестве подрамника в приемник в соответствии со значениями параметра , , • Л Т В , и . Н Install(f) Рассчитываются значения параметра и. Установленная рамка получает Показать Сообщение.

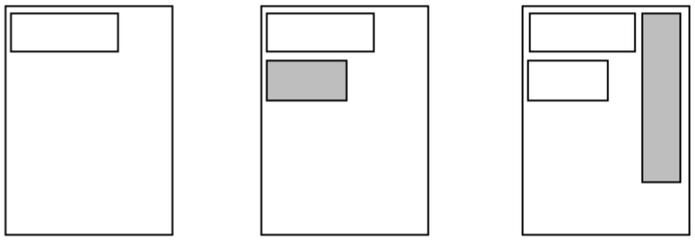
• Убрать(f) извлекает подрамник из приемника. Фрейм контейнера не восстанавливается, и другие подфреймы не затрагиваются.

• вычисляет верхний левый угол кадра в соответствии со стратегией . f s SetLT(hs)

• = по вертикали: размещается ниже ранее установленных подкадров.

• = по горизонтали: размещается справа от ранее установленных подкадров.

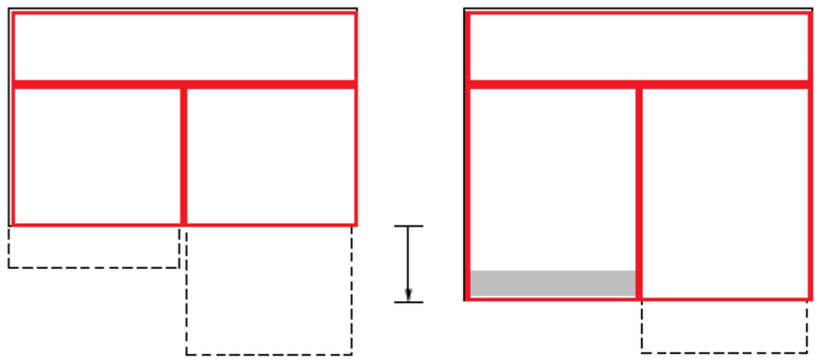
• = относительный: перед вызовом (,) функций задаются относительно Л Т Л Т f (,) контейнера. Результатом будет (,) из в абсолютных координатах.



s = по вертикали s = по горизонтали

отображает данные фрейма и пересылается во все вложенные фреймы. (В меню • Фреймы класс существует Показать ничего не должно отображаться, но подклассы могут переопределять Показать для отображения их данных.) Показать is автоматически вызывается, когда ранее скрытый фрейм становится видимым.

• Скрыть вызывается, когда кадр становится невидимым (закрытым или наложенным). Он пересылается на все подкадры. Подклассы могут переопределить его, чтобы освободить структуры данных, которые не нужны, когда фрейм невидим. • Изменить размер (r, h), rb изменяет правый нижний угол рамки на и, но не восстанавливает содержимое рамки на экране. Сообщение пересылается на все частично видимые подкадры. На следующем рисунке показан эффект изменения размера нижней части рамки. (Пунктирные линии обозначают логический размер подкадров. Рамка никогда не превышает свой логический размер.)



Если подкадр становится скрытым с помощью операции Изменения размера , он получает сообщение Скрыть. Если скрытый подкадр становится видимым, он получает сообщение Показать.

• Переместить (dx, dy) перемещает рамку и ее подкадры на dx и dy без обновления экрана.

• Копировать (f) возвращает глубокую копию получателя в f. Если вызывается с помощью f= NIL копия получает класс приемника .

• Расфокусировка устанавливает текущий кадр фокусировки равным нулю.

• Нейтрализовать удаляет все пользовательские метки во фрейме и его подфреймах (пустой в этом классе).

• Клавиша-манипулятор (ch) отправляется в текущий фрейм фокусировки всякий раз, когда символ ch был введен с клавиатуры .

• HandleMouse (x, y, кнопки) вызывается повторно, пока мышь находится внутри фрейма. Если имеется подфрейм, содержащий x, y сообщение пересылается в этот подкадр.

• This(x, y) возвращает подкадр, содержащий точку (x, y) или НОЛЬ, если такого подкадра нет.

• Широковещательная передача (m) отправляет сообщение m ко всем видимым подкадрам.

• Загрузить (r) отправляет сообщения о загрузке во все подкадры для считывания их содержимого из райдера r.

• Сохранить (r) отправляет сообщения хранилища во все подкадры для записи их содержимого пользователю. r.

• Настройка фокуса (f) позволяет f новая рамка фокусировки. Старый фокус расфокусирован.

• RemoveMarks (f) удаляет все системные метки (курсор и маркер в форме звезды) с f.

Экран также является рамкой, которая содержит все видимые средства просмотра и его подкадры. Его можно использовать, например, для следующих операций:

Экран.Это (x, y) возвращает средство просмотра, содержащее (x,y)

Экран.Показать отправляет сообщения о показе всем зрителям

Экран.Нейтрализовать удаляет метки пользователя во всех зрителях

Средства просмотра

Средство просмотра - это рамка с линией границы. Обычно оно содержит строку заголовка и меню (также рамки). Его можно изменять в размерах, копировать, расширять до высоты или ширины экрана, сохранять и закрывать. Обычно средство просмотра не должно использоваться в качестве области рисования, но оно должно содержать рамки, в которых рисуется текст или графика. Один и тот же тип средств просмотра может использоваться для всех видов данных, отображаемых на экране, поэтому средство просмотра текста ничем не отличается от средства просмотра графики, оно только содержит другой тип фрейма.

```
Представления определений;
    ИМПОРТИРУЙТЕ фреймы;

    CONST userTrack = 0; systemTrack = 1;

    Средство просмотра КЛАССОВ (Фреймы.Фрейм);
        (* переопределенные методы *)
        ОТОБРАЖЕНИЕ ПРОЦЕДУРЫ;

        Изменение размера ПРОЦЕДУРЫ (r, b: ЦЕЛОЕ
        ЧИСЛО); Перемещение ПРОЦЕДУРЫ (dx, dy: ЦЕЛОЕ
        ЧИСЛО); Копирование ПРОЦЕДУРЫ (VAR f:
        Фреймы.Рамка); ПРОЦЕДУРА Установки (f:
        Рамки.Рамка); ПРОЦЕДУРА удаления (f: Рамки.Рамка);

        ПРОЦЕДУРА SetLT (f: Рамки.Кадр; стратегия: SHORTINT);
    КОНЕЧНЫЙ зритель;

    НОВАЯ ПРОЦЕДУРА (title, меню: МАССИВ СИМВОЛОВ; стратегия: SHORTINT):
    Средство просмотра; Заголовок ПРОЦЕДУРЫ (title: МАССИВ СИМВОЛОВ): Фреймы.Фрейм;
    Меню ПРОЦЕДУРЫ (manu: МАССИВ СИМВОЛОВ): Фреймы.Фрейм;

(* стандартные команды меню просмотра *)
Закрыть ПРОЦЕДУРУ;
Скопировать ПРОЦЕДУРУ;
Увеличить ПРОЦЕДУРУ;
Сохранить ПРОЦЕДУРУ;
ЗАВЕРШИТЬ просмотры.
```

• **Показать** рисует границу и строку заголовка.

• **Изменение размера (r, b)** восстанавливает границы средства просмотра. Любая расширенная часть средства просмотра будет удалена.

• **Переместить (dx, dy)** перемещает растровое изображение вьювера на **dx** и **dy**.

• **Копировать(f)** возвращает полную копию программы просмотра. Если **f = НОЛЬ** перед вызовом, он получает средство просмотра классов.

• **Install(f), Remove(f) и SetLT(f, s)** похожи на фреймы.Обрамление, но учитывайте также линию границы средства просмотра

.

• **Создать (заголовок, меню, ы)** возвращает и отображает новое средство просмотра в порядке, указанном **s. Если** Название # "" рамка заголовка с указанным именем устанавливается в качестве первого подрамника. Если меню # "" меню с заданным текстом меню устанавливается в качестве второго подфрейма. Если установлен маркер в форме звезды, он используется для определения верхней части нового средства просмотра, в противном случае для верхней части выбирается значение по умолчанию.

• **Заголовок (t)** возвращает рамку заголовка с текстом **t**.

• **Menu(m)** возвращает статическую текстовую рамку с текстом **m**.

• **Закрыть** закрывает программу просмотра, содержащую эту команду. Если это единственная программа просмотра в своей дорожке и дорожка является

при наложении дорожка закрывается и отображаются наложенные средства просмотра. • **Копировать** копии средства просмотра, содержащие эту команду

• **Увеличивать** расширяет программу просмотра на всю высоту экрана или, если она уже имеет эту высоту, на всю ширину экрана.

• **Сохранить** сохраняет содержимое программы просмотра под именем файла, содержащимся в строке заголовка, и создает

резервная копия *.Bak.

Текстовые рамки

Текстовая рамка отображает на экране текст с различными шрифтами. Текстовые фреймы уже поддерживают большую часть стандартного поведения текстового редактора (например, ввод, выделение, копирование, удаление, прокрутка и т.д.).

```
ОПРЕДЕЛЕНИЕ TFrames;
    ИМПОРТИРУЙТЕ фреймы, текст, шрифты, файлы

    Константа заменена = 0; вставлена = 1; удалена = 2; (* операции
    обновления *) ТИП
        Местоположение = ЗАПИСЬ
        org, pos: LONGINT;                  (* org: начало строки; pos: позиция символа *)
        dx, x, y: ЦЕЛОЕ                     (* положение экрана (относительно рамки); dx: ширина символа в точках x, y *)
        ЧИСЛО; КОНЕЦ;

    КЛАСС Frame (Рамки.Frame);
        текст: Txt.Text;                    (* отображаемый текст *)
        org: LONGINT;                       (* позиция первого отображаемого символа *)
        lsp, asr, dsr: ЦЕЛОЕ ЧИСЛО;         (* интервал между строками, верхний и нижний регистры
        margW: ЦЕЛОЕ ЧИСЛО;                 отображаемого текста *) (* левое поле (полоса прокрутки) *)
        время: LONGINT; каретка:            (* время последнего выделения *) (* расположение курсора;
        местоположение; selbeg, selend:     caret.pos < 0, если курсор не установлен *) (* диапазон выделения;
        Местоположение; Автоматический      selbeg.pos < 0, если выделение не установлено *) (*
        поиск: ЛОГИЧЕСКОЕ значение;         автоматический отступ для вкладок (по умолчанию: TRUE) *)

        Обновление ПРОЦЕДУРЫ (beg, end: LONGINT; mode: INTEGER);
        Удаление ПРОЦЕДУРЫ (beg, end: LONGINT);
        Вставка ПРОЦЕДУРЫ (pos: LONGINT);
        Запись ПРОЦЕДУРЫ(ch: CHAR);
        НАСТРОЙКА ПРОЦЕДУРЫ (beg, end: LONGINT; fnt:
        Шрифты.Шрифт); ПРОЦЕДУРА setColor(начало, конец: LONGINT;
        цвет: SHORTINT); ПРОЦЕДУРА SetCaret (начало: LONGINT);
        ПРОЦЕДУРА RemoveCaret;
        ПРОЦЕДУРА setSelection(начало, конец: LONGINT);
        ПРОЦЕДУРА RemoveSelection;

ПРОЦЕДУРА TrackCaret(VAR x, y: INTEGER; VAR buttons: SET);
ПРОЦЕДУРА TrackSelection(VAR x, y: INTEGER; VAR buttons: SET);
ЛИНИЯ отслеживания ПРОЦЕДУРЫ (VAR x, y: INTEGER; VAR org: LONGINT; VAR buttons: SET);
СЛОВО отслеживания ПРОЦЕДУРЫ(VAR x,y: INTEGER; VAR pos: LONGINT; VAR buttons: SET);
ПРОЦЕДУРА ShowFrom(pos: LONGINT);
ПРОЦЕДУРА Pos(x, y: INTEGER): LONGINT;
ПРОЦЕДУРА MarkBusy;
(* переопределенные методы *)
Расфокусировка ПРОЦЕДУРЫ;
Нейтрализация ПРОЦЕДУРЫ;
Отображение ПРОЦЕДУРЫ;
Скрытие ПРОЦЕДУРЫ;
Изменение размера ПРОЦЕДУРЫ (r, b: ЦЕЛОЕ ЧИСЛО);
Копирование ПРОЦЕДУРЫ (VAR f: кадры.Frame);
PROCEDURE HandleMouse(x, y: ЦЕЛОЕ ЧИСЛО; кнопки: НАБОР);
PROCEDURE HandleKey(ch: CHAR);
PROCEDURE Load(VAR r: Файлы.Rider);
Хранилище ПРОЦЕДУР (VAR r: Файлы.Rider);
КОНЕЧНЫЙ фрейм;
КОНЕЧНЫЕ TFrames.
```

Вновь выделенный текстовый фрейм инициализируется значениями по умолчанию, но текст не устанавливается. Когда подкласс является производным от TFrames.HandleMouse и HandleKey обрабатывать обычно приходится переопределять два метода пользовательский ввод особым образом. В большинстве случаев остальное можно унаследовать как есть.

• Обновить (beg, end, m) восстанавливает экран после изменения диапазона beg..end в тексте в соответствии с операцией m. Вставить, удалить, записать setFont Вызывается из setColor, и .

• Удалить (beg, end) удаляет диапазон текста beg..end и восстанавливает экран. • Вставить(p, b) вставляет буфер b в нужную позицию текста и восстанавливает экран. b) p • Написать(ch) вставляет символ ch вставляет курсор в текст и выводит его на экран шрифтом, соответствующим предыдущему символу.

• setFont (начало, конец, f) изменяет шрифт диапазона к и восстанавливает экран. начало ..конец

• setColor (начало, конец, c) изменяет цвет диапазона начало .. конец к и восстанавливает экран. c

• SetCaret (pos) устанавливает курсор в положение позиция и делает приемник текущим кадром фокусировки. Это также устанавливает программу просмотра фокусировки Oberon. Ранее отображенный курсор удаляется. • RemoveCaret удаляет курсор, если он установлен. • setSelection(начало, конец) выбирает диапазон начало..конец. Старое выделение в этом фрейме удалено.

• RemoveSelection удаляет выделение, если оно существует.

• TrackCaret (x, y, b) устанавливает курсор в положение на экране x, y и отслеживает это до тех пор, пока все кнопки мыши не будут отпущен. Кнопки, нажатые во время отслеживания, b возвращаются (слева = 2, посередине = 1, справа = 0). • Выбор треков (x, y, b) x, y Правая кнопка мыши нажата в точке . Текст выделяется в соответствии с движением мыши до тех пор, пока не будут отпущены все кнопки. Возвращаются кнопки, нажатые во время отслеживания b. b

• Линия пути (x, y, org, b) . Мышь находится в x, y. Движения мыши отслеживаются, и строка, содержащая , подчеркивается до тех пор, пока не будут отпущены все кнопки. По завершении текущего у org является позицией первого символ в этой строке и представляет собой набор кнопок, нажатых во время отслеживания. b

• Ключевое слово (x, y, pos, b) . Мышь находится в x, y. Отслеживаются движения мыши и слово, содержащее x, утекущийподчеркивается до тех пор, пока не будут отпущены все кнопки. По завершении позициявляется позицией подчеркнутого слова и b представляет собой набор кнопок, нажатых во время отслеживания. Слово представляет собой строку символов, размер которой больше пустого.

• ShowFrom(pos) перерисовывает текст с первой строки, начиная со следующей позиция .

• Pos(x, y) возвращает положение текста, соответствующее координатам экрана x, y.

• MarkBusy включает и выключает стрелку в нижней части полосы прокрутки.

Переопределенные методы (см. Описания в суперклассе):

• Расфокусировка удаляет курсор из приемника и устанавливает текущий кадр фокусировки равным нулю. • Нейтрализовать удаляет все текстовые метки с приемника (курсор, выделение, метка прокрутки) • Показать перерисовывает весь текст рамки с позиции org.

• Скрыть удаляет все метки в приемнике и освобождает внутренние структуры данных.

• Изменить размер (r, б) б расширяет правое и нижнее поля приемника до и соответственно и перерисовывает r текст в любой расширенной части рамки.

возвращает полную копию получателя в If = NIL перед вызовом, он получает класс получателя. • Копировать(f)

• HandleMouse(x, y, b) рисует указатель мыши, когда мышь находится внутри рамки. При щелчке мышью (b # {}) он реагирует соответствующим образом, прокручивая, устанавливая курсор или выбирая. Он также обрабатывает сочетания клавиш для удаления и копирования выделенного текста.

• HandleKey (ch) вызывается, когда символ ch введен, а приемник является рамкой фокусировки. IT в положении курсора.

chпишет. Считывает текст от райдера и вставляет его в рамку, не отображая. • Загрузить (r) • Сохранить (r) записывает текст в рамке для райдера r.

Статические текстовые фреймы

Следующий модуль предоставляет класс для статических текстовых фреймов, то есть фреймов, которые отображают текст, но не поддерживают прокрутку и редактирование. Фреймы меню или строки заголовков программ просмотра являются примерами таких фреймов.

```
ОПРЕДЕЛЕНИЕ STFrames;

    ИМПОРТ TFrames, файлов;

    КЛАСС Frame (TFrames.Frame);
        (* переопределенные методы *)
        PROCEDURE HandleMouse(x, y: ЦЕЛОЕ ЧИСЛО; кнопки:
        НАБОР); PROCEDURE HandleKey(ch: СИМВОЛ);
        PROCEDURE Copy(VAR f: Фреймы.Frame);
        Загрузка ПРОЦЕДУРЫ (VAR r: Файлы.Rider);
        Хранилище ПРОЦЕДУР (VAR r: Файлы.Rider);
    КОНЕЧНЫЙ фрейм;

КОНЕЧНЫЕ STFrames.
```

Тексты

В этом модуле библиотеки классов просто устанавливает объектно-ориентированный интерфейс к модулю Txt Тексты из оригинальной системы Oberon. Переопределять тексты было неразумно, поскольку существующие приложения, такие как компилятор, полагаются на исходный текстовый модуль и не будут работать с измененными текстами. Семантика классов в Txt такой же, как в оригинальной системе Oberon, и поэтому не описывается далее.

```
ОПРЕДЕЛЕНИЕ Txt;

    ИМПОРТИРУЙТЕ объекты, файлы, шрифты, тексты;

    CONST Name = 1; String = 2; Int = 3; Real = 4; LongReal = 5; Char = 6;

    БУФЕР КЛАССА (Объекты.Object);
        b: Texts.Buffer;
        len: LONGINT;
        ПРОЦЕДУРА открыта;
        ПРОЦЕДУРА CopyTo(DB: Buffer);
    КОНЕЧНЫЙ буфер;

    Класс Text (Objects.Объект);
        t: Тексты.Текст;
        len: LONGINT;
        ПРОЦЕДУРА добавления (B: Буфер);
        ПРОЦЕДУРА изменения шрифта (начало, конец: LONGINT; fnt:
        шрифты.Шрифт); ПРОЦЕДУРА changeColor(beg, end: LONGINT; col:
        SHORTINT); ПРОЦЕДУРА ChangeOffset(beg, end: LONGINT; voff:
        SHORTINT); ПРОЦЕДУРА Delete(beg, end: LONGINT);
        ПРОЦЕДУРА Insert(pos: LONGINT; B: буфер);
        Загрузка ПРОЦЕДУРЫ (VAR R: Файлы.Rider);
        Открыть ПРОЦЕДУРУ (имя: МАССИВ СИМВОЛОВ);
        Сохранить ПРОЦЕДУРУ (начало, конец: LONGINT; B: Буфер);
        Сохранить ПРОЦЕДУРУ (VAR W: Файлы.Райдер);
    КОНЕЧНЫЙ текст;
```

```
КЛАСС Reader (Объекты.Object);
    r: Texts.Reader;
    ПРОЦЕДУРА Открыта (T: Text; pos: LONGINT);
    ПРОЦЕДУРА Pos(): LONGINT;
    ПРОЦЕДУРА прочитана (VAR ch: CHAR);
КОНЕЧНЫЙ считыватель;

ЗАПИСЬ КЛАССА (Objects.Объект);
    w: Тексты.Средство записи;
    ПРОЦЕДУРА сброса;
    ПРОЦЕДУРА Buf(): буфер;
    ПРОЦЕДУРА setFont (fnt: Шрифты.Шрифт);
    НАБОР цветов ПРОЦЕДУРЫ (col:
    SHORTINT); НАБОР параметров
    ПРОЦЕДУРЫ (voff: SHORTINT); Запись
    ПРОЦЕДУРЫ(ch: CHAR); ЗАПИСЬ ПРОЦЕДУРЫ;
    ПРОЦЕДУРА writeInt(x: LONGINT; n: LONGINT);
    ПРОЦЕДУРА WriteHex(x: LONGINT); ПРОЦЕДУРА
    WriteString(s: МАССИВ СИМВОЛОВ); ПРОЦЕДУРА
    WriteReal(x: ВЕЩЕСТВЕННЫЙ; n: ЦЕЛОЕ ЧИСЛО);
    ПРОЦЕДУРА WriteLongReal(x: LONGREAL; n: INTEGER);
    ПРОЦЕДУРА WriteLongRealHex(x: LONGREAL);
    ПРОЦЕДУРА WriteRealFix(x: REAL; n, k: INTEGER);
    ПРОЦЕДУРА WriteRealHex(x: REAL);
КОНЕЧНЫЙ писатель;

КЛАСС сканера (считывателя);
    nextCh: CHAR;
    строка: ЦЕЛОЕ
    ЧИСЛО; класс:
    INTEGER; i:
    LONGINT; x:
    РЕАЛЬНЫЙ; y: LONGREAL;
    c: CHAR;
    len: SHORTINT;
    s: МАССИВ 32 СИМВОЛОВ;
ПРОЦЕДУРА открыта (T: текст; pos: LONGINT);
Сканирование ПРОЦЕДУРЫ;
КОНЕЧНЫЙ сканер;

Журнал изменений: текст;          (* текст в окне просмотра журнала *)

Вызов ПРОЦЕДУРЫ(VAR B: Buffer);
ParText ПРОЦЕДУРЫ(): текст;                    (* эквивалент Oberon.Par.text *)
ПРОЦЕДУРА ParPos(): LONGINT; (* Oberon.Par.pos *)
КОНЕЦ текста.
```

Графические рамки

Графические рамки представляют собой простое расширение стандартных рамок, поддерживающих некоторые графические примитивы для рисования в пределах прямоугольной области отсечения. Чтобы не иметь дела с "низкоуровневым" программированием поведения фрейма, методы Показать установите внутренний прямоугольник отсечения и вызовите метод Restore . Один просто переопределить Восстановить метод, с помощью процедуры, которая перерисовывает весь кадр. Каждый раз, когда часть кадра становится недействительным (например, Размер операция), будет установлен обтравочный прямоугольник соответствующим образом и будет вызван метод Restore .

```
ОПРЕДЕЛЕНИЕ GFrames;
    ИМПОРТ курсоров, рамок;

    CONST
        черный = 0; белый = 15;                          (* стандартные цвета
        заменить = 0; нарисовать = 1;                    рисования *) (* режимы
        инвертировать = 2; слева = 2;                    рисования *) (* кнопки мыши *)
        посередине = 1; справа = 0; нет =                (* не используйте шаблон для операций
        0; единица измерения = 36000;                    заливки *) (* один пиксель *)

    Тип
        Шаблон = LONGINT;

    ПРОЦЕДУРА серого цвета (плотность: РЕАЛЬНАЯ): Шаблон;

    КЛАСС Frame (Рамки.Рамка);
        x0, y0: ЦЕЛОЕ ЧИСЛО;                             (* исходное положение относительно левого верхнего кадра;
        сетка: LONGINT;                                  по умолчанию = (0, 0) *) (* сетка курсора; по умолчанию =
        масштабирование: SHORTINT;                       единица измерения *) (* мощность масштабирования; по умолчанию = 0
        маркер: курсоры.Маркер;                          *) (* маркер курсора; по умолчанию = Курсоры.Стрелка *)

        ПРОЦЕДУРА setOrigin(x, y: ЦЕЛОЕ ЧИСЛО);
        ПРОЦЕДУРА SetGrid(g: LONGINT);
        ПРОЦЕДУРА setZoom(z: SHORTINT);

        ПРОЦЕДУРА trackMouse(ПЕРЕМЕННЫЕ x, y: ЦЕЛОЕ ЧИСЛО;
        ПЕРЕМЕННЫЕ buttons: SET); ПРОЦЕДУРА EditFrame(x, y: ЦЕЛОЕ
        ЧИСЛО; кнопки: SET); ПРОЦЕДУРА setZoom(z: SHORTINT);

        ПРОЦЕДУРА trackMouse(ПЕРЕМЕННЫЕ x, y: INTEGER;
        ПЕРЕМЕННЫЕ buttons: SET); ПРОЦЕДУРА EditFrame(x, y: INTEGER;
        кнопки: SET); ПРОЦЕДУРА Restore(l, t, r, b: LONGINT);
        ПРОЦЕДУРА Clear;

        ПРОЦЕДУРА DrawDot(x, y: LONGINT; цвет, режим: SHORTINT);
        ПРОЦЕДУРА DrawLine(x1, y1, x2, y2: LONGINT; цвет, режим: ЦЕЛОЕ ЧИСЛО);
        ПРОЦЕДУРА drawRect(l, t, r, b, w: LONGINT; color, mode: SHORTINT; pattern: шаблон);
        ПРОЦЕДУРА drawCircle(x, y, d: LONGINT; color, mode: SHORTINT);
        ПРОЦЕДУРА DrawEllipse(x, y, a, b: LONGINT; color, mode: SHORTINT);
        ПРОЦЕДУРА fillRect(l, t, r, b: LONGINT; color, mode: SHORTINT; pattern: шаблон);
        ПРОЦЕДУРА fillCircle(x, y, d: LONGINT; color , режим: SHORTINT; шаблон: Pattern);
        ПРОЦЕДУРА FillEllipse(x, y, a, b: LONGINT; color, режим: SHORTINT; шаблон: Pattern);
        ПРОЦЕДУРА CopyRect(l, t, r, b, dx, dy: LONGINT; режим: SHORTINT);

        (* переопределенные
        методы *) ПОКАЗАТЬ ПРОЦЕДУРУ;
        Скрыть ПРОЦЕДУРУ;
```

Изменение размера ПРОЦЕДУРЫ (r, b: ЦЕЛОЕ ЧИСЛО);

Перемещение ПРОЦЕДУРЫ (dx, dy: ЦЕЛОЕ ЧИСЛО);

Копирование ПРОЦЕДУРЫ (ПЕРЕМЕННАЯ f: кадры.Frame);

PROCEDURE HandleMouse(x, y: ЦЕЛОЕ ЧИСЛО; кнопки: НАБОР); КОНЕЧНЫЙ фрейм;

КОНЕЧНЫЕ GFrames.

Все поля класса доступны только для чтения, за исключением маркер , который обозначает курсор, используемый в графической рамке; маркер может быть изменен пользователем, когда он захочет.

- setOrigin(x, y) устанавливает начало координат x, y в значение (относительно левого верхнего угла рамки. Все остальные координаты относятся к этой исходной точке.
- SetGrid(g) устанавливает сетку в . g
- setZoom (z) увеличивает рамку в соответствии с коэффициентом 2 . z • Мышь отслеживания (x, y, кнопки) может вызываться пользователем (например, в EditFrame) для отслеживания движения мыши и получения ее фактических (выровненных по сетке) координат (x, y) и кнопки были нажаты. Сама сетка мыши выровнена по началу кадра.
- Редактировать кадр (x, y, кнопки) вызывается системой автоматически всякий раз, когда мышь находится в пределах рамки и была нажата хотя бы одна кнопка мыши. Обычно переопределенный метод EditFrame использует Trackmouse для выполнения интерактивного редактирования. EditFrame в этом классе пустой.
- Восстановление (l, t, r, b) вызывается системой автоматически. Он устанавливает рамку отсечения в l, t, r, b.
- Очистить удаляет рамку.

- Показать вызывает перерисовку всей рамки с помощью вызова Восстановить.
- Скрыть устанавливает для внутреннего обтравочного прямоугольника нулевое расширение.
- Resize(r, b) вызывает перерисовку расширенной части рамки с помощью команды Restore после установки обрезка прямоугольника соответствующим образом.
- Переместить (dx, dy) перемещает рамку, включая ее начало координат, на dx, dy , не рисуя ее на экране.
- Копировать (f) возвращает глубокую копию получателя в f. Если f=NIL перед вызовом он получает класс получателя. • HandleMouse(x, y, b) рисует курсор мыши и вызывает команду EditFrame при нажатии кнопки.

Примечание Относительно этой версии

Оригинальная версия TR 109 вышла из печати и недоступна в электронном виде. Это представляет исторический интерес для развития языковой семьи Оберон. Теперь это в электронной форме для пользы заинтересованных. Было приложено все усилия, чтобы сохранить верность оригиналу, вплоть до копирования его макета и типографики.

Б. Смит-Манншотт
bsmithma@iiic.ethz.ch

Обратите внимание, что Object Oberon был разработан в то время, когда язык Oberon отличался от того, что есть сейчас известен как Oberon-1:

- Отдельные файлы ОПРЕДЕЛЕНИЯ и МОДУЛЯ, как в Modula-2.
- У типов процедур нет полного списка аргументов, а есть только список типов их аргументов. аргументы.
- Неявное разыменование указателей, похоже, отсутствует.
- Цикл FOR отсутствует.
- Нет вложенных комментариев.

1999.03.09/10:17