Приёмы проектирования в Обероне-2 и Компонентном Паскале

Translated excerpts from
Objektorientierte Programmierung in Oberon-2
by **Hanspeter Mössenböck**
Authorized translation of Chapter 9 «Entwurfsmuster» of Hanspeter Mössenböck:
Objektorientierte Programmierung in Oberon-2, 3rd Edition, 1998.
Permission granted by Springer, Heidelberg.
English translation: **Bernhard Treutwein**

Русский перевод: **Иван Кузьмицкий, Александр Шелюгов, Пётр Кушнир, Александр Ильин.**

9 Приёмы проектирования

Опытные программисты отличаются тем, что имеют набор готовых решений для типичных проблем. Это определяет их экспертный уровень. Когда перед ними ставится очередная задача, им не приходится разрабатывать решение заново, они могут использовать опробованные ранее подходы.

Такие стандартные решения называются *приёмами проектирования*. Они предоставляют схемы решений наиболее часто встречающхся проблем. В мир объектно-ориентированного программирования приёмы проектирования вошли благодаря замечательной книге Гаммы и других авторов. Приёмы проектирования не являются изобретением ООП. Приёмы проектирования являются ничем иным, как алгоритмами и структурами данных объектно-ориентированного программирования. Наиболее подходящий синоним - объектные структуры.

9.1 Мотивация

Паттерны в обычных программах

Перед тем, как перейти к коллекции полезных паттернов, вкратце рассмотрим особенности и преимущества приёмов проектирования. Возьмём такой хорошо известный пример из области обычного программирования, как структуру двоичного дерева, показанного на рис. 9.1. Это типичный *паттерн*. Двоичные деревья известны своей эффективностью при поиске в больших объёмах данных. Если вам нужен быстрый поиск, то двоичное дерево - проверенный путь для решения этой задачи.

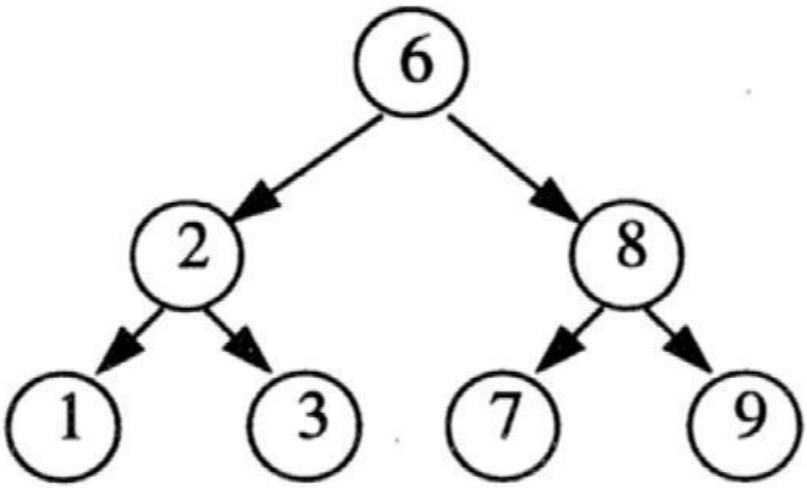


Рис. 9.1: Двоичное дерево как пример паттерна

Составляющие элементы паттерна

Из чего состоит паттерн? Главным образом, из **имени**, **описания проблемы** и **решения**.

Имя относится к удобной и знакомой терминологии, используемой разработчиками для общения. Сказать "создай двоичное дерево" куда проще, нежели "создай структуру, каждый элемент которой имеет двух потомков и значение левого потомка меньше либо эквивалентно значению правого".

Описание проблемы для паттерна двоичного дерева выглядит так: используйте двоичное дерево если у вас имеется большая структура данных в оперативной памяти и вам нужны эффективные способы добавлять или удалять данные в этой структуре, либо производить поиск в ней. При описании проблемы можно опустить упоминание ограничений или подводных камней. Например, двоичные деревья не подходят в случаях, когда данные изменяются слишком часто. Это может привести к опасности дегенерации дерева.

Третьим компонентом паттерна является его **решение** или **реализация**, которые зачастую намеренно абстрагированы от деталей. Это защищает паттерн от особенностей языка реализации или конкретных структур данных. Исполнение двоичного дерева целиком зависит от связей между узлами дерева и их потомками.

Паттерны в объектно-ориентированных программах

Приёмы проектирования (паттерны) в **объектно-ориентированном** смысле помогают описывать взаимодействия классов и\или объектов в процессе поиска решения задачи. Для подобных описаний зачастую применяются диаграммы классов. Чтобы зафиксировать динамику взаимодействия классов, диаграммы расширяются фрагментами исходных текстов или описанием их поведения с помощью диаграмм взаимодействия.

Гамма и др. (1995 г.) предоставили каталог из двадцати пяти приёмов проектирования. Здесь же мы хотим отметить лишь наиболее важные из них, поскольку многие паттерны до некоторой степени схожи между собой. Кроме этого, мы покажем некоторые довольно полезные паттерны, не включенные в вышеупомянутый каталог Гаммы и др.

Гамма и др. подразделяют паттерны на три основных категории: п**орождающие паттерны**, служащие для гибкого создания объектов и\или объектных структур (классов); **структурные паттерны**, описывающие наиболее часто

варианты агрегирования объектов в большие структуры; **паттерны поведения**, которые помогают выразить часто встречающиеся варианты поведения объектов.

9.2 Порождающие паттерны

В большинстве случаев память под объекты выделяется динамически. Для этого в Oberon-2 предназначена стандартная процедура NEW. Прочие языки программирования предоставляют похожие конструкторы. Однако, простое выделение памяти зачастую связано с некоторыми сложностями. Иногда, дополнительно необходимо совершать определенные действия (например, инициализацию), часто заранее неизвестно, какой тип объекта будет создан. В этой главе мы познакомим вас с тремя порождающими шаблонами, которые могут обработать подобные ситуации:

- Конструктор Создание и инициализация объектов
- Фабрика Создание объекта определенного типа
- Прототип Создание объекта определенного типа

9.2.1 Конструктор

При создании объекта желательно инициализировать его поля. Если вы не хотите забыть об инициализации, разумно будет объединить создание и определение в одно действие. Для этого и существует шаблон *конструктор*.

Многие языки программирования (напр. C++ или Java) уже обладают конструкторами, включёнными в описание языка. Oberon-2 и Компонентный Паскаль не имеют конструкторов, но их легко смоделировать с помощью шаблона *конструктор*. Компонентный Паскаль предоставляет для записей атрибут LIMITED, который явно запрещает размещение с помощью NEW вне данного модуля.

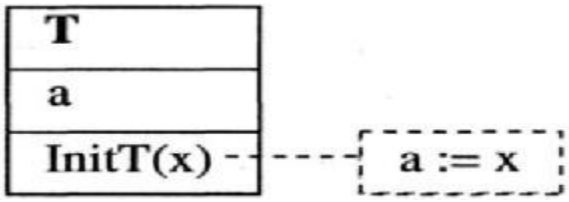


Рис. 9.2 Класс, содержащий метод инициализации

Рассмотрим класс *T*, содержащий метод *InitT*, который инициализирует поле *a* (Рис 9.2). Конструктор для *T* будет реализован как функция *NewT*, создающая объект типа *T* и вызывающая процедуру инициализации *InitT*:

```
TYPE
        T* = POINTER TO RECORD
                a- : REAL
        END;

PROCEDURE ( t : T ) InitT* ( x : REAL );
        BEGIN
                t.a :=x
END InitT;

PROCEDURE NewT* ( x : REAL ) : T;
        VAR t : T;
        BEGIN
                NEW(t); t.InitT(x);
                RETURN t
END NewT;
```

Теперь, всякий раз при необходимости в объекте типа *T*, последний может быть создан и проинициализирован вызовом своим конструктором:

```
obj := NewT(x)
```

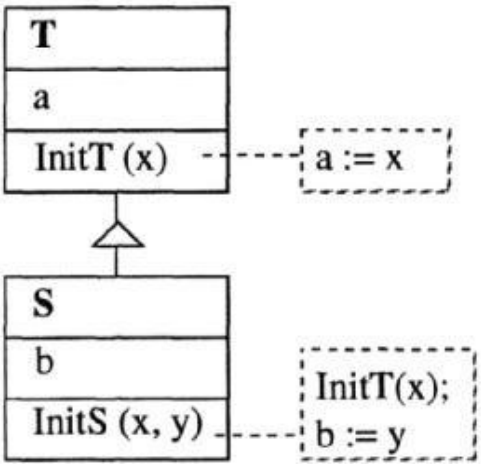
Данный шаблон легко применить для любого другого класса путем замены "*T*" на определенное имя класса (напр. *NewRectangle* и *InitRectangle*).

При создании объекта подкласса, необходимо инициализировать как атрибуты подкласса, так и атрибуты базового класса (классов). Для этой цели шаблон

конструктор может быть изменен. *S* является подклассом для *T*, конструктор *NewS* реализован:

```
obj := NewS(x,y)
```

для создания нового объекта типа *S*. Новый объект *S* определяется при вызове процедуры *InitS*, которая, в свою очередь, инициализирует атрибуты базового класса обращением к процедуре *InitT*.




```
TYPE
        S* = POINTER TO RECORD(T)
                b- : REAL
        END;

PROCEDURE ( s: S ) InitS* ( x, y : REAL );
        BEGIN
                s.b := x; s.InitT(x)
END InitS;

PROCEDURE NewS* ( x, y : REAL ) : S;
        VAR s : S;
        BEGIN
                NEW(s); s.InitS(x, y);
                RETURN s
END NewS;
```

Рис. 9.3 Конструктор базового класса

9.2.2 Фабрика

Назначением паттерна *фабрика* является создание объектов в условиях, где динамический тип объекта заранее не зафиксирован (не определён статически) в программе.

Это хорошо видно в следующем примере. Предположим, существуют различные реализации видов текста, происходящие от абстрактного класса *Text* (Рис. 9.4). Здесь, *SimpleText* отвечает за простые ASCII-тексты, а *StyledText* может применять различные шрифты и стили.

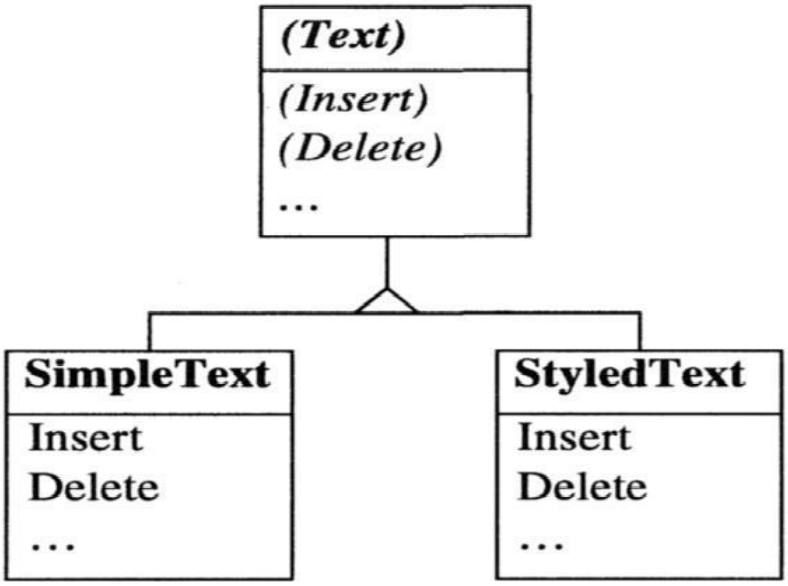


Рис. 9.4 Различные виды текста

Давайте представим некоторый редактор, который должен работать как с *SimpleText* так и с *StyledText,* по выбору пользователя. Следовательно, редактор имеет атрибут *t* типа абстрактного класса *Text*, который может хранить в себе объекты *SimpleText* или *StyledText* (рис. 9.5).

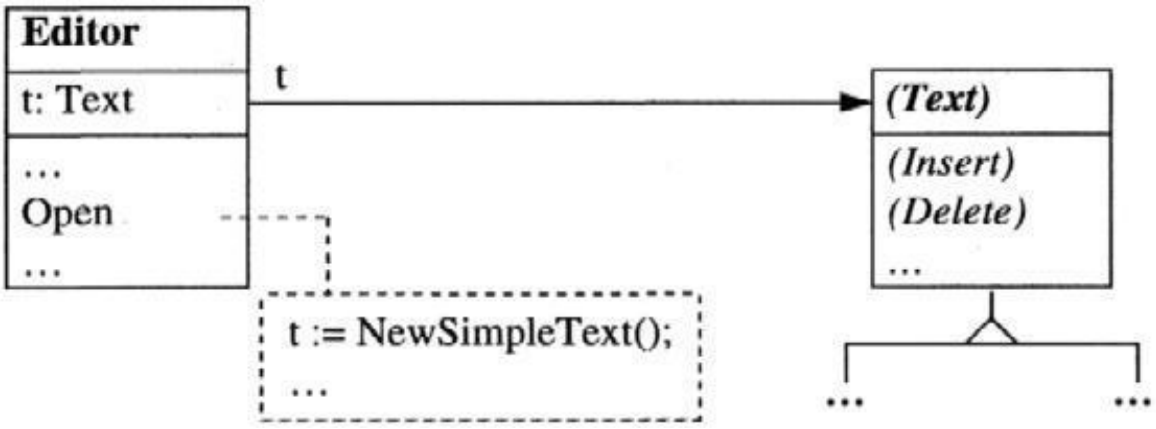


Рис. 9.5 Динамический тип t статически зафиксирован

Объект типа *Text* размещается в памяти при вызове редактором метода *Open*. Редактор должен решить, какой тип объекта будет создан. На Рис. 9.5 редактор создает объект типа *SimpleText*. Это отражено в коде программы без малейшей возможности изменить подобный выбор. Если вы хотите создавать объекты типа *StyledText*, вам нужно изменять

исходный текст.

Эту проблему можно решить с помощью шаблона проектирования *фабрика*. Вместо *статически* зафиксированного типа текста, поручим создание текста *объекту-фабрике*. Для каждого вида текстов определим специальную фабрику (*SimpleFactory*, *StyledFactory* и т.д.), каждая из которых производит соответствующий объект типа *Text*. Все фабрики будут происходить от абстрактного класса-фабрики, схема которого изображена на Рис. 9.6

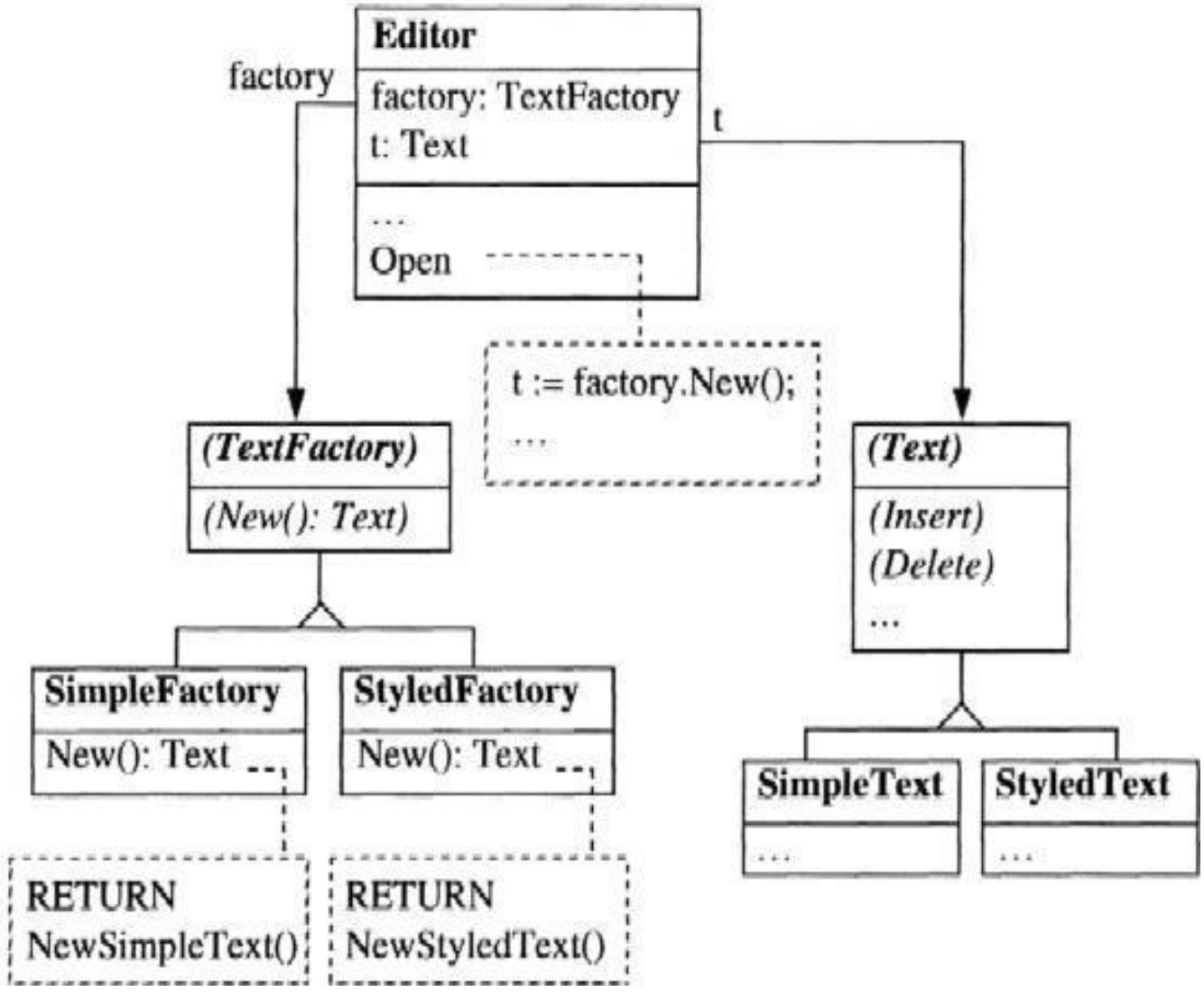


Рис. 9.6 Генерация текста объектом-фабрикой

Теперь, когда редактор инициализирован, объект типа *SimpleFactory* или *StyledFactory* передан атрибуту *factory*

редактора. В случае, когда выбран объект типа *StyledFactory*, вызов метода *Open* порождает вызов метода *New*, определенного именно в *StyledFactory*. Таким образом генерируется объект типа *StyledText* и возвращается редактору.

Код метода *Open* не отталкивается от того, какой вид текста будет генерироваться. Сам текст становится результатом работы фабрики, которую можно выбрать в процессе инициализации редактора и поменять на другую во время работы программы.

Подводя итог: если требуется гибко выбирать тип создаваемых динамических объектов, то не размещайте их в памяти напрямую, а запрашивайте у объекта-фабрики.
Последний может содержать в себе различные виды фабрик, генерирующих отдельные виды объектов. Предпочтительный тип фабрики выбирается в процессе инициализации системы.

9.2.3 Прототип

Шаблон *прототип* существует для той же цели, что и *фабрика*. А именно, для обеспечения гибкости в вопросах динамической типизации создаваемых объектов. Реализация *прототипа* несколько отличается от реализации *фабрики* и является более простой во многих отношениях.

Применим он в случаях, когда требуется новый объект некоторого типа, не создаваемый непосредственно, но копируемый из *объекта-прототипа*, который уже имеет требуемый тип.

Вернемся к редактору из главы 9.2.2. На этот раз мы имеем различные виды текстов, которые могут быть чередующимися в редакторе. Для каждого из разновидностей текста создан свой объект-прототип (т.е. объекты типа *SimpleText* и *StyledText*). Один из этих прототипов во время инициализации редактора сохраняется в атрибут *protoText*. В методе редактора *Open* создается копия атрибута *protoText* и используется как текст в окне редактора.

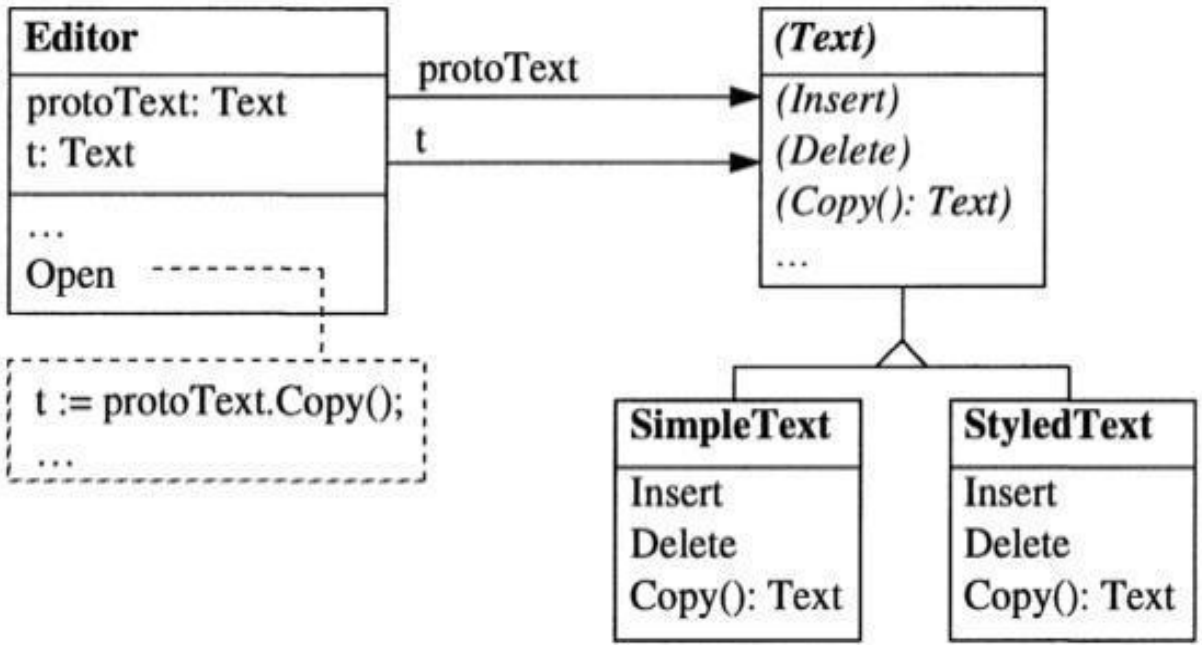


Рис. 9.7 Генерация текста путем копирования объекта-прототипа.

В чем разница между шаблоном-прототипом и шаблоном-фабрикой? Шаблон-прототип требует существования объектов (здесь, объекты типа *Text*), для клонирования самих себя, т.е. применяющих метод *Copy*. Это не добавляет никаких ограничений, но может быть непрактичным для иерархий классов. Шаблон-фабрика не имеет таких ограничений и может произвести подходящую инициализацию вновь созданного (совместимого по типу) объекта, который может зависеть от состояния программы на данном этапе разработки. Проблемой для шаблона-фабрики является необходимость постоянного соблюдения иерархии классов для *фабрик*, что усложняет всю систему в целом.

Другим преимуществом шаблона-прототипа является возможность создания копии не только единичного объекта, но также *подсистемы*, состоящей из множества объектов. Такие подсистемы могут быть собраны в течение работы программы и каждая операция копирования воспроизведёт полную копию подсистемы. Подобное весьма сложно проделать, используя объект-фабрику.

9.3 Структурные паттерны

Объекты нечасто существуют изолированно друг от друга, обычно поставленные задачи решаются совместно с другими объектами. Для этих целей объекты объединяются в часто встречающиеся структуры определённых форм. Эти формы, отнесённые в категорию структурных паттернов, и будут обсуждаться ниже. Мы ограничимся рассмотрением следующих шаблонов:

- Семейство — Выстраивание иерархий классов
- Адаптер — Подгонка чужеродного класса к семейству
- Компоновщик — Сборка частей в новую часть
- Декоратор — Последовательное функциональное соединение
- Двойник — Уход от множественного наследования

9.3.1 Семейство

Семейство - это очень простой до тривиальности паттерн. Он нам понадобился только для того, чтобы зафиксировать имя шаблона. Семейство содержит абстрактный класс и его подклассы. На рис.9.8 продемонстрировано семейство GUI-объектов, а на рис. 9.4 семейство текстовых объектов. Все члены семейства имеют тот же самый интерфейс, что и его базовый абстрактный класс. Поэтому члены семейства являются взаимозаменяемыми. Программа, которая умеет работать с GUI-объектами, сможет воспользоваться возможностями таких объектов, как Checkbox, Button и Scrollbar

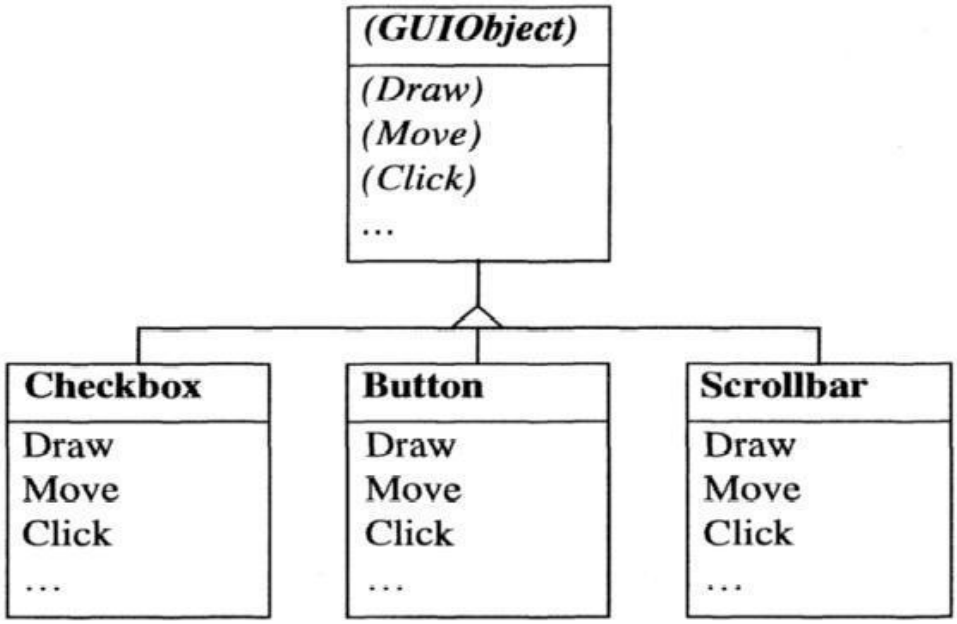


Рис. 9.8 Семейство GUI-объектов.

9.3.2 Адаптер

Иногда может потребоваться сделать существующий класс совместимым с некоторым семейством. Другими словами: рассмотреть класс как член этого семейства. Для этого пришлось бы его унаследовать от базового класса семейства. Но это зачастую невозможно, поскольку наш класс уже может быть наследником другого базового класса и мы не хотим или не имеем возможности использовать множественное наследование. Кроме того, может сложиться и такая ситуация, когда мы не имеем доступа к исходным текстам и потому не можем изменять отношения наследования.

Решить эту проблему можно введением нового члена семейства, который работает словно *адаптер* для инородных классов. Он реализует сообщения семейства путем трансляции сообщений и переадресации их чужому классу.

Рассмотрим пример: графический редактор оперирует с семейством фигур (линий, прямоугольников, окружностей и т.д.). Также редактор должен представлять возможность работы с текстом. Допустим, существует класс *Text*, но он не

является членом семейства фигур. Последнее означает, что *Text* не может быть обработан как фигура. Следовательно, требуется ввести *TextAdapter*, который преобразует сообщения класса *Figure* в сообщения класса *Text* (см. рис. 9.9). Например, метод *Draw* в текстовом адаптере будет реализован путем считывания одного символа текста за другим и прорисовкой каждого на экране.

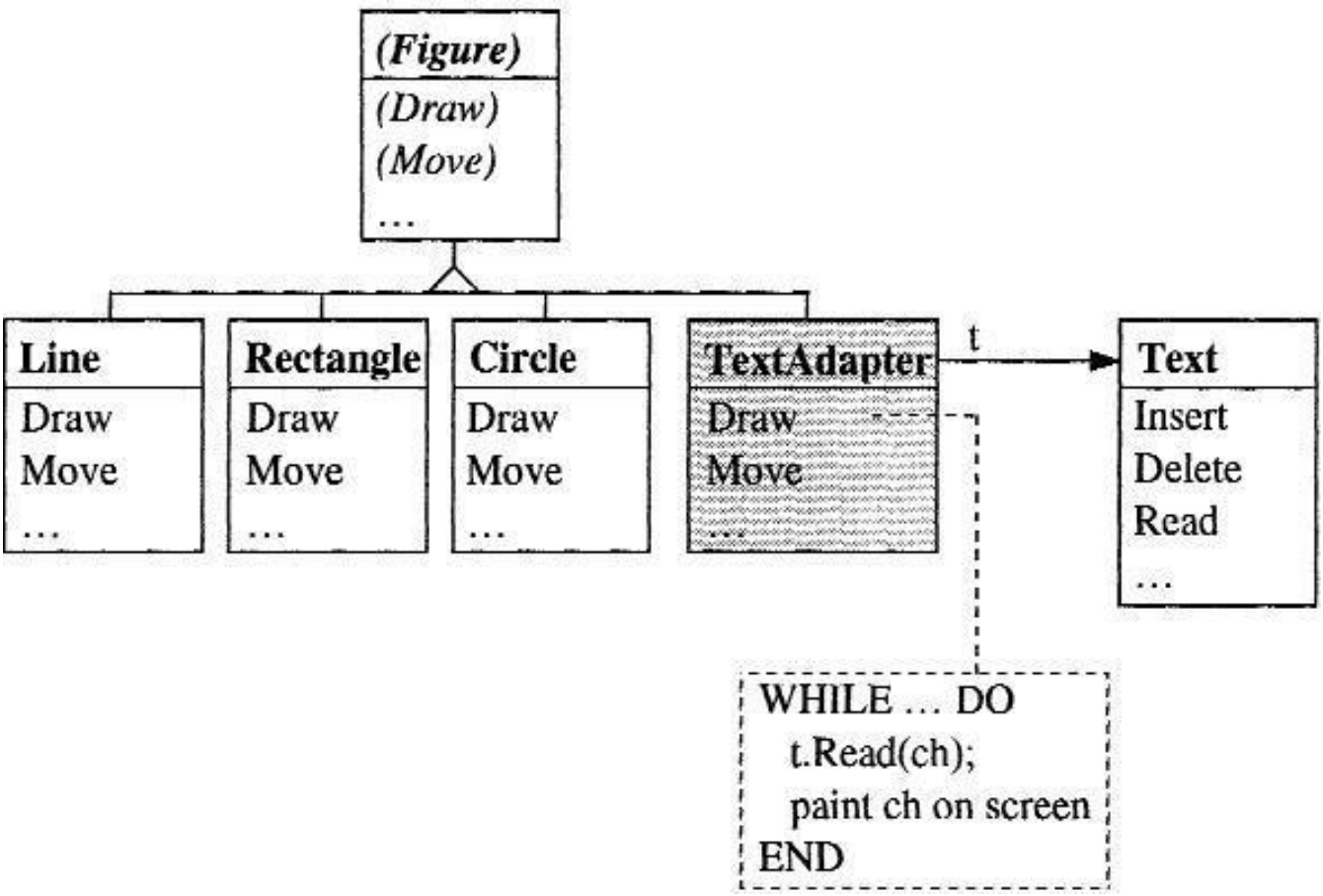


Рис. 9.9 Внедрение класса *Text* в семейство *Figure* при помощи адаптера *TextAdapter*

Можно рассматривать текстовый *адаптер* как компоновочный слой, перекрывающий класс *Text* и предоставляющий ему другой интерфейс. По этой причине он иногда также называется *обёрткой*.

Шаблон проектирования *адаптер* один из наиболее часто используемых шаблонов. Проблема соединения двух ранее несвязанных классов возникает практически всюду в объектно-

ориентированных программах. *Адаптер* решает эту проблему весьма изящно.

9.3.3 Компоновщик

Часто бывает нужно собрать несколько объектов в один более крупный, который ведет себя как целостный объект, в свою очередь, могущий агрегироваться с другими объектами в еще более крупный объект.

Структура, получаемая из рекурсивного группирования обособленных объектов, называется *компоновщик*. Встретить его можно, например, в графических редакторах, где несколько независимых фигур объединяются в группу, которая рисуется и перемещается как один графической объект. Рис. 9.10 демонстрирует эту ситуацию.

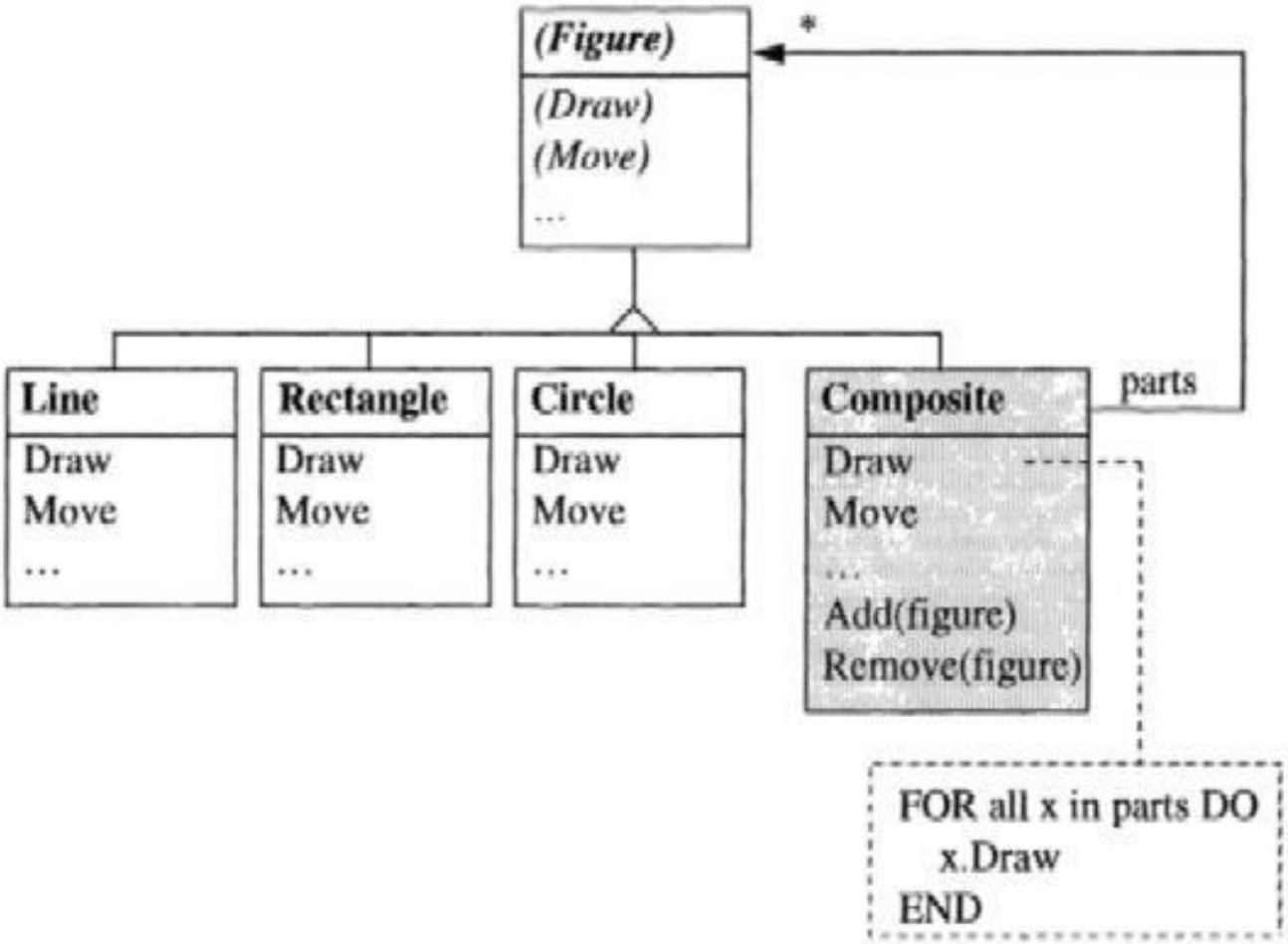


Рис. 9.10 Использование компоновщика для объединения фигур в графическом редакторе

Компоновщик является наследником класса *Figure*, потому он может быть обработан как фигура. Он содержит набор частей (фигур), где эти части в свою очередь могут быть другими *компоновщиками*. Отправленные *компоновщику* сообщения преобразуются в сообщения для вложенных объектов. Сообщение *Draw*, посланное группе, порождает несколько сообщений *Draw* для всех ее частей. Также, зачастую бывает полезно части ссылаться на содержащий её компоновщик. Удобно представить такую ссылку в виде атрибута класса *Figure*.

У компоновщика есть методы для добавления (Add) и удаления (Remove) частей из группы. У GoF [GHJV95] *Add* и *Remove* являются методами абстрактного базового класса *Composite*, но это выглядит несколько искусственно, так как для объектов-одиночек их нет смысла реализовывать.

Существует множество применений компоновщика. Например, он полезен для управления окнами, где один фрейм может содержать множество вложенных фреймов. Другим примером является редактор математических (или химических) формул, где одна величина (напр. интеграл) может как содержать вложенные величины (напр. дроби), так и в свою очередь сама быть вложенной в другую формулу.

9.3.4 Декоратор

Шаблон *декоратор* применим в случаях, когда требуется добавить новое поведение классу, не изменяя и не наследуя его. Эта новая модель поведения накрывает собой класс и может быть добавлена или убрана в любой момент работы программы.

Возьмем пример из GoF [GHJV95], чтобы наглядно проиллюстрировать этот шаблон. Система окон использует кадры (*Frames*) как прямоугольные области для рисования

текста или графики. Существует также семейство кадров, включающее в себя подклассы *TextFrame* или *GraphicFrame*. Мы хотим добавить некоторые свойства этим кадрам, например, добавить полосу прокрутки для перемещения по содержимому кадра, или хотим предоставить различные стили оформления для окантовки кадра.

Если реализовать эти свойства путем наследования, то иерархия классов станет слишком громоздкой в силу большого числа возможных комбинаций. Рис. 9.11 показывает, что происходит, если кадры с полосами прокрутки и кадры с различными рамками реализовать в отдельных подклассах.

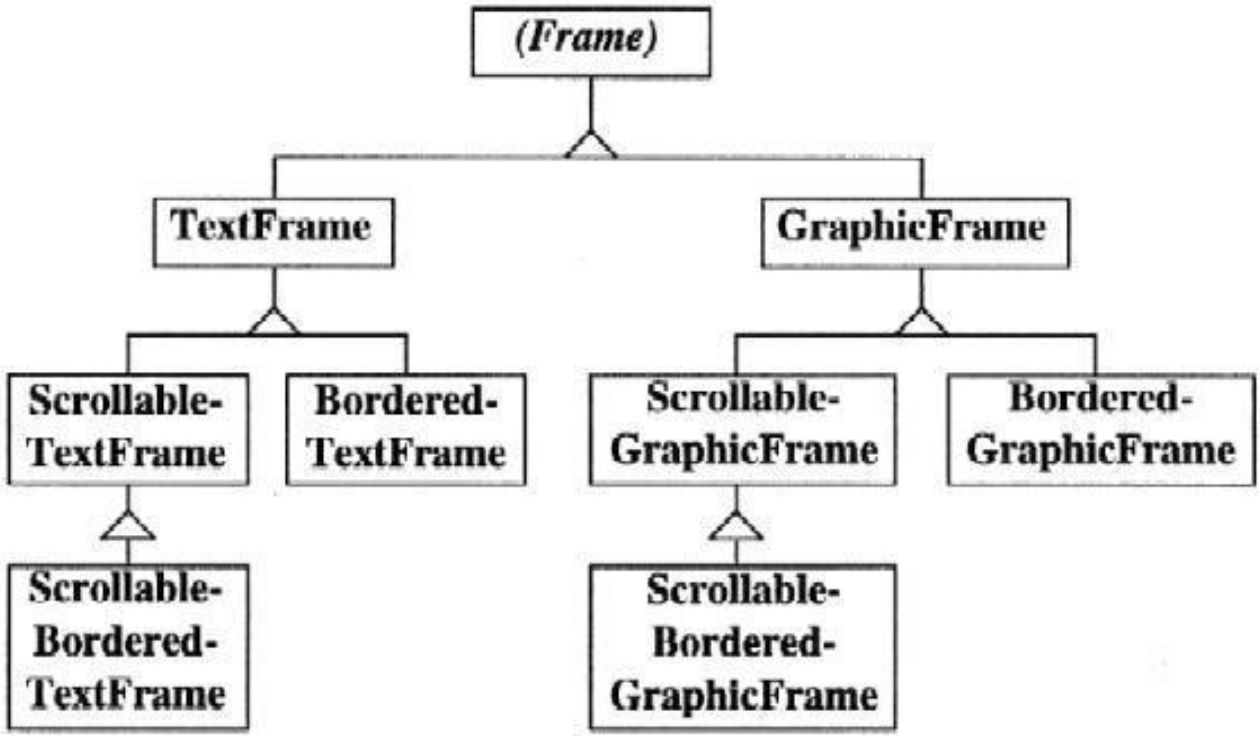


Рис. 9.11 Взрывное увеличение числа классов при реализации свойств как подклассов

Идея шаблона *декоратор* заключается в реализации различных возможностей в отдельном классе *поверх* главного класса, вместо унаследования. Рис. 9.12 показывает схему этого примера. Поверх объекта типа *TextFrame* располагается объект *ScrollDecorator*. При получении последним сообщения *Draw*, объект рисует полосу прокрутки и пересылает сообщение объекту *TextFrame*, который впоследствии нарисует собственный текст.

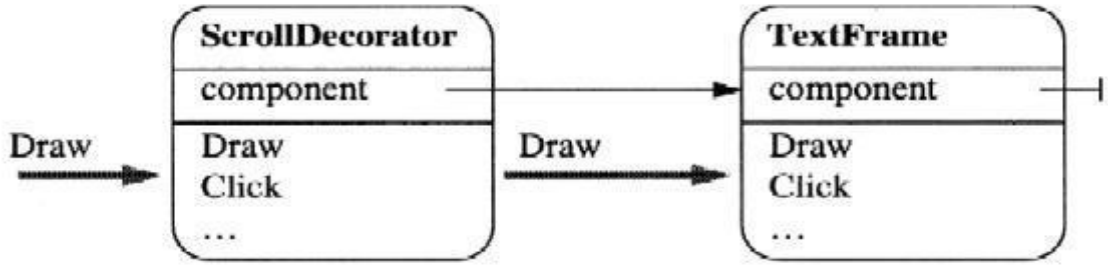


Рис. 9.12 Объект ScrollDecorator располагается перед TextFrame

Все клиенты, работавшие с объектом *TextFrame* теперь должны отправлять свои сообщения объекту *ScrollDecorator*. Это похоже на *заместителя* объекта *TextFrame*, вдобавок объект *ScrollDecorator* должен иметь тот же интерфейс, или, другими словами, принадлежать тому же семейству классов, что и объект типа *TextFrame*. Ну а на рис. 9.13, у нас получилась диаграмма классов, иллюстрирующая шаблон *декоратор.*

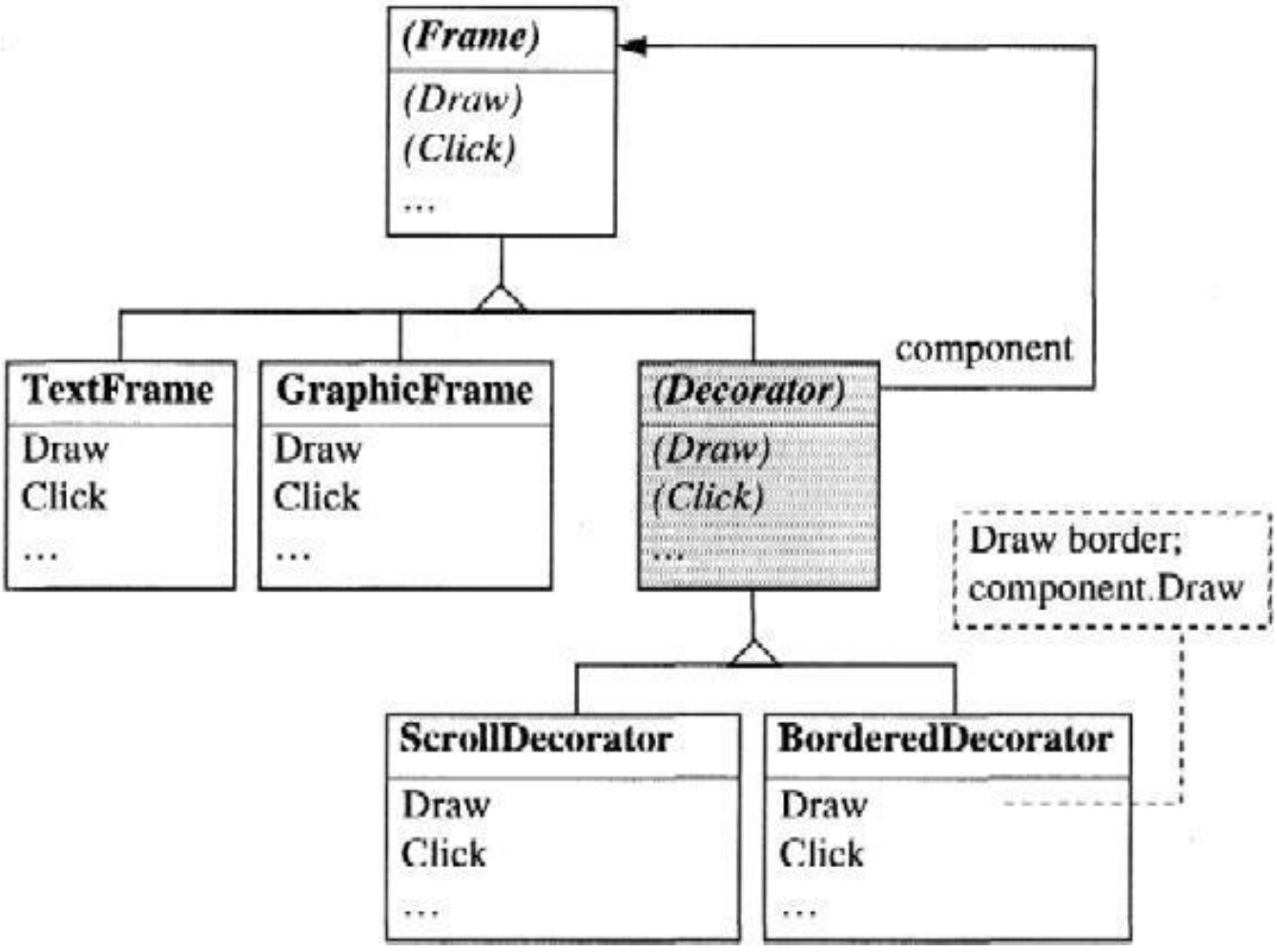


Рис. 9.13 Объект *ScrollDecorator* поверх *TextFrame*

Класс *Decorator* принадлежит семейству *Frame*, потому может обрабатываться как любой другой кадр. Благодаря атрибуту *component* декоратор может быть соединен с любым другим членом семейства *Frame*, в том числе и с другим декоратором.

Следовательно, возможно скомбинировать свойства *BorderedDecorator* и *ScrollDecorator* перед применением *GraphicFrame*. Когда *BorderedDecorator* получит сообщение *Draw*, он нарисует окантовку и передаст сообщение *Draw* декоратору *ScrollDecorator*, который, в свою очередь, нарисует полосу прокрутки и передаст сообщение *Draw* объекту *GraphicFrame*.

Стоит упомянуть, что комбинация всех свойств может не только изменяться произвольно без внедрения огромного числа подклассов, но и изменяться в течение работы программы. Например, можно размещать рамку на кадре только, если указатель мыши наведен на данный кадр. Наследование менее гибко в данном случае. Отношение наследования не может быть изменено в течение работы программы.

Главной проблемой шаблона *декоратор* является потеря связи с декорируемым объектом. Клиенты обращаются не к объекту *Frame*, а к объекту *Decorator*. Если последний создан динамически, вы должны проследить за тем, чтобы все существующие клиентские ссылки на объект Frame уже были установлены. У клиентов более нет возможности обратиться к атрибутам объекта Frame напрямую с тех пор, как они не имеют ссылки на этот объект.

Шаблон *декоратор* представляется очень мощным. Небольшая модификация этого шаблона позволяет нам расширить класс сразу в нескольких независимых направлениях. Рассмотрим другой пример.

В главе 6 мы говорили об абстрактном классе *Stream*, имеющего несколько расширений, например *Terminal*, *File* или *Network*. Они расширяют класс в *одном* направлении, относящееся к исходящим данным. Если мы захотим создать другое направление, которое сделает возможным применение

различных алгоритмов шифрования (напр. RSA, DES и т.д.) нам нужно будет второе, ортогональное измерение. Как видно на рис.9.14, каждый из видов исходящих данных может быть скомбинирован с любым алгоритмом шифрования. Если бы подобные комбинации реализовывались с применением подклассов, для каждого пересечения на нашей «сетке» нам бы потребовалось отдельное расширение класса *Stream* (т.е., *RSATerminal*, *RSAFile*, *RSANetwork* и т.д.). Расширение оказалось бы весьма сложным решением. При добавлении нового вида исходящих данных потребовалось бы скомбинировать его с каждым возможным алгоритмом шифрования, что дало бы в результате огромное число новых подклассов.

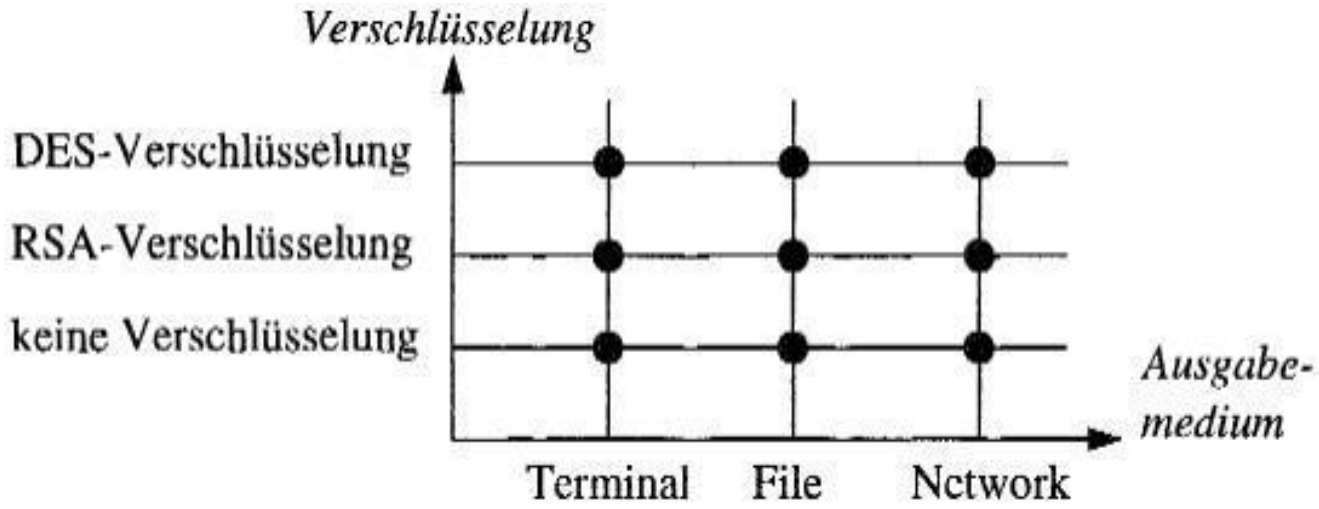


Рис. 9.14 Расширение класса Stream для различных видов данных и алгоритмов шифрования

С помощью шаблона *декоратор* можно избежать взрывного роста количества классов. Алгоритм шифрования представим как *декоратор,* применимый к классу *Stream*. На рис 9.15 *Encoder* исполняет роль декоратора.

Чтобы совместить алгоритм шифрования DES с файлом, объект *File* накрывается объектом DESEncoder. Если объекту DESEncoder будет отправлено сообщение *Write*, символ *ch* будет зашифрован и затем передан в *поток*. Вызов *Write(ch)* перенаправляется объекту *File*. Представляется возможным добавление нового вида данных (напр. *MemoryFile*) или нового

алгоритма шифрования так, что мы по-прежнему можем комбинировать каждый вид исходящих данных с каждым алгоритмом шифрования.

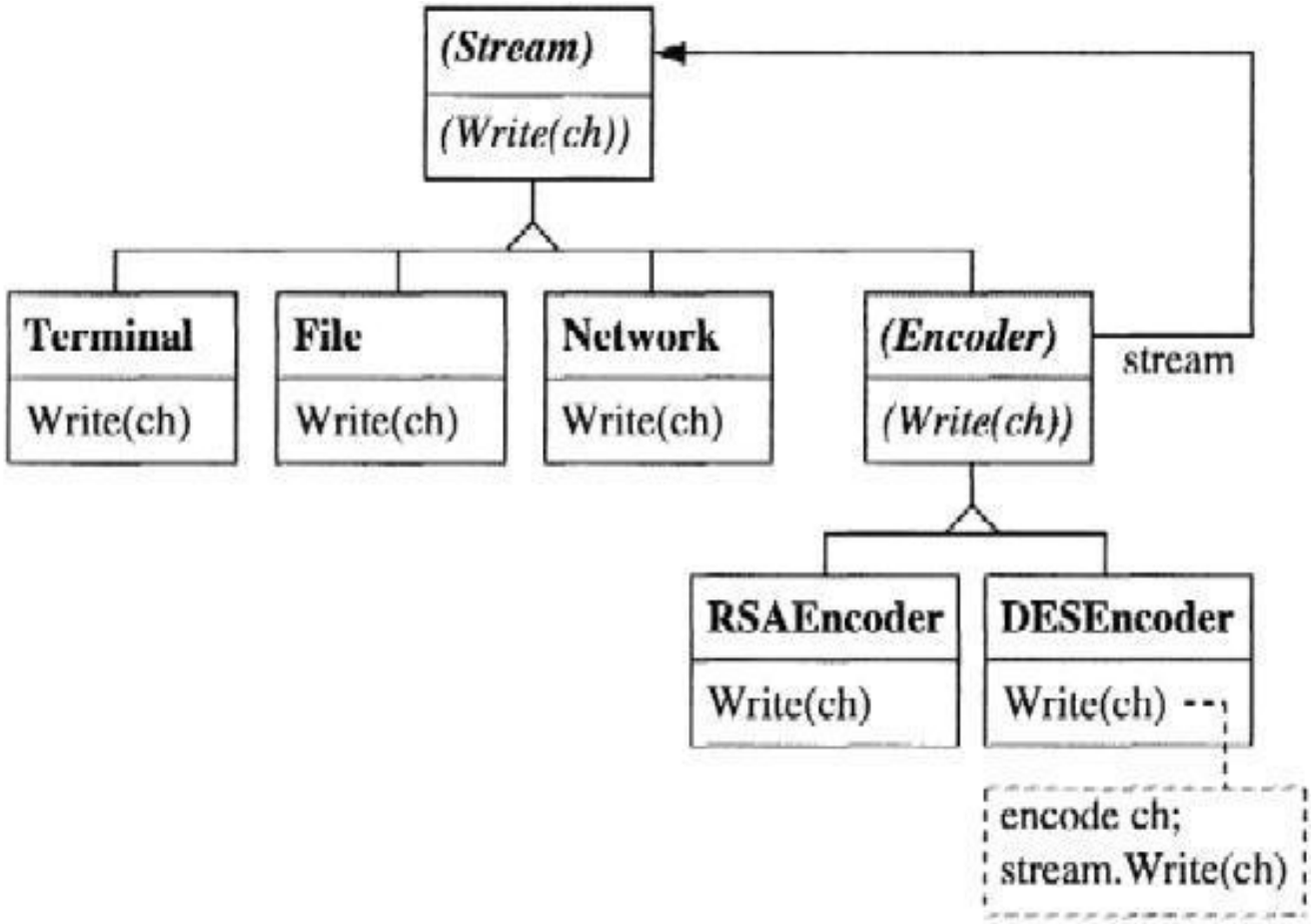


Рис 9.15 Шаблон *декоратор* применяется для расширения класса *Stream* в двух направлениях

9.3.5 Двойник

Некоторые языки, подобно C++ или Eiffel, предлагают множественное наследование. Как было показано в главе 5, это позволяет унаследовать атрибуты и методы более чем от одного базового класса, и, что может оказаться более важным, сделать подкласс совместимым с несколькими базовыми классами. С другой стороны, множественное наследование порождает проблемы, такие как конфликт имён или слишком

сложная иерархия классов.
Поэтому следует как можно чаще избегать множественного наследования и полагаться лишь на одиночное наследование. Такие языки, как Oberon-2 и Компонентный Паскаль, вовсе не поддерживают множественное наследование. Следовательно, мы можем использовать только одиночное наследование. Но в этом случае, помочь может шаблон *двойник*.

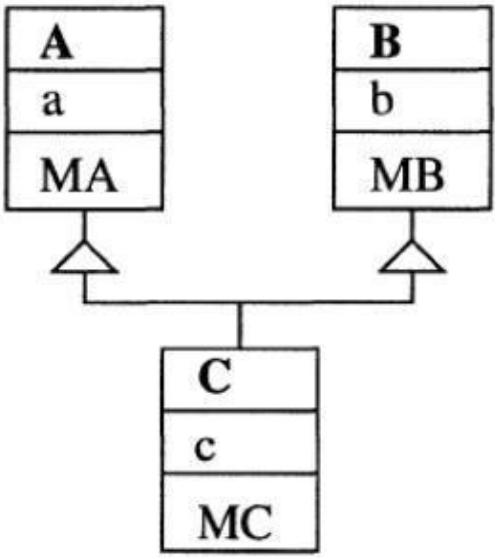


Рис. 9.16 Множественное наследование

Рис. 9.16 изображает базовую схему множественного наследования. Класс C получен из двух классов A и B и наследует атрибуты a и b совместно с методами MA и MB. C совместим с A и B. Объекты C могут быть привязаны как к списку объектов A, так и к списку объектов B.

Шаблон *двойник* же основан на идее разделения класса C на два класса-близнеца CA и CB, взаимно связанными указателями друг на друга (рис. 9.17). CA выведен из A и CB выведен из B, каждое наследование является полноценным. Методы и атрибуты C привязаны к одному из двух классов-двойников, напр. CA.

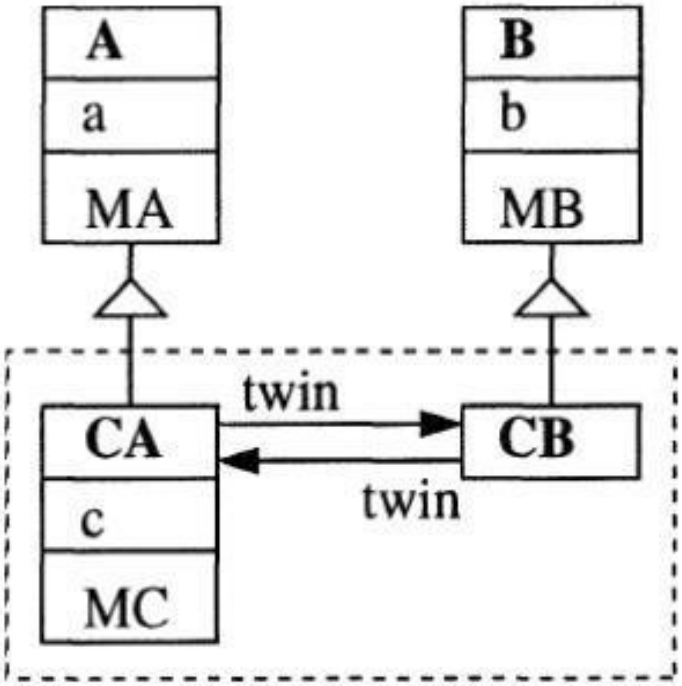


Рис. 9.17 Классы-близнецы CA и CB

Вместо объекта C создается пара объектов CA и CB. Объект CA может быть добавлен в список объектов A, объект CB может быть добавлен в список объектов B. Потому класс двойников совместим с A и B так же, как и при множественном наследовании. Вызов MA можно совершить через CA, а вызов MB можно совершить через CB.

Наследованные атрибуты вне класса доступны так:

```
ca.a
ca.twin.b
ca.MA
ca.twin.MB
```

Доступ к компонентам B получается косвенным. Это цена, которую приходится платить, дабы избежать множественного наследования, впрочем, в большинстве случаев дорого она не обходится. Одного уровня косвенности можно избежать, если объявить MB как метод класса CA, который передаст соответствующее сообщение в класс CB (Рис. 9.18).

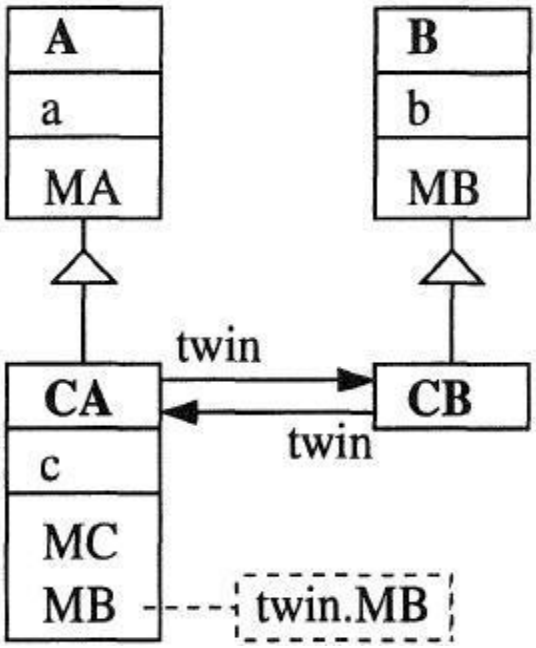


Рис. 9.18 Шаблон "двойник"

Следующий пример покажет применение шаблона *двойник* в конкретном приложении. Представим компьютерную игру, содержащую некоторые артефакты, например мячи и ракетки, которые получены из общего класса *Item*. Мячи являются *активными* артефактами, находящимися в постоянном движении. Такие артефакты выводятся из базового класса *Process*; это создает диаграмму классов, изображенную на рис. 9.19, которая показывает реализацию множественного наследования. Мячи могут быть членами списков артефактов и в тоже время списков процессов.

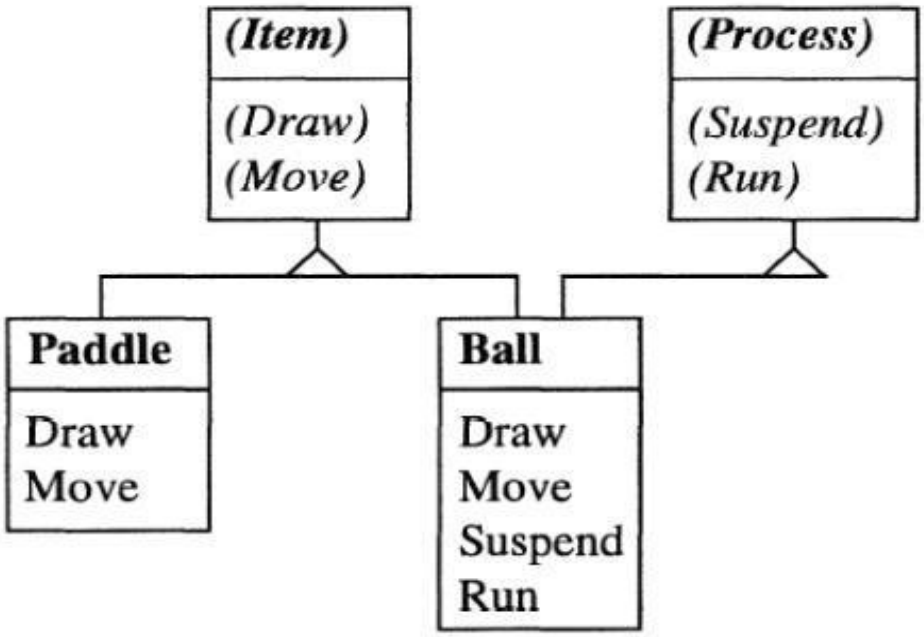


Рис. 9.19 Применяющая множественное наследование игра в мяч

Объекты, унаследованные от класса *Process* могут получать сообщение *Run* несколько раз в секунду. Мяч, реагируя на эти сообщения, будет перемещаться в пространстве. Непрерывное перемещение мяча на экране обеспечивается множеством сообщений *Run*. Если пользователь нажмет любую клавишу, все процессы получат сообщение *Suspend*. Мяч среагирует на это, замерев или изменив цвет.

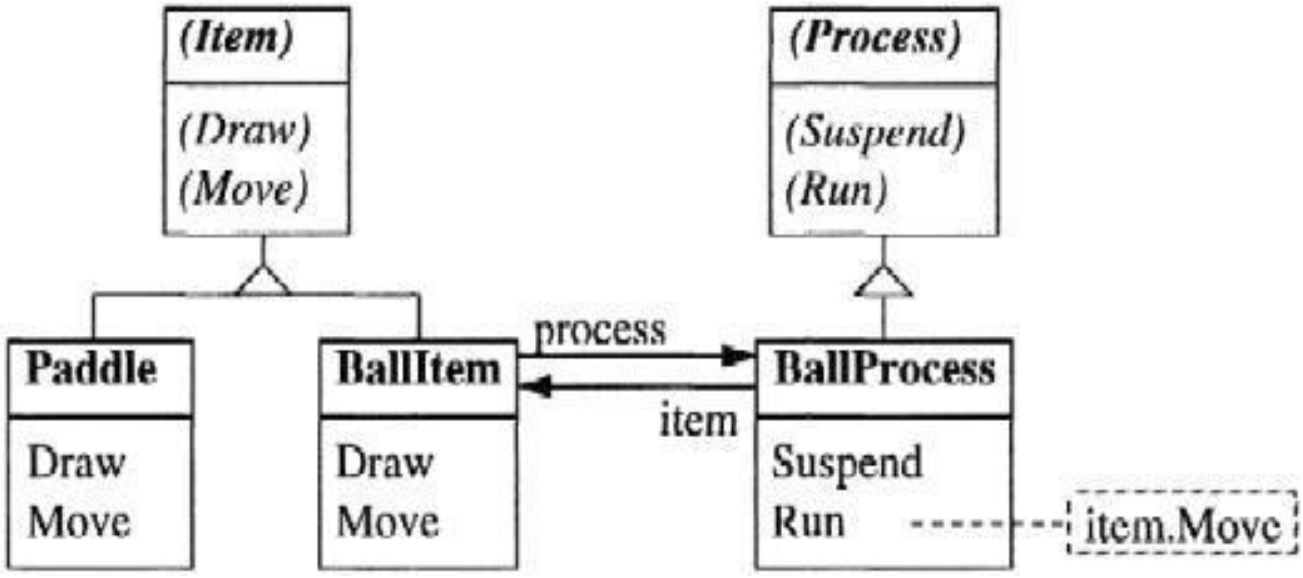


Рис. 9.20 Разделение класса *Ball* на пару двойников: *BallItem* и *BallProcess*

Теперь, смоделируем эту ситуацию с помощью шаблона *двойник*. Класс *Ball* разделим на классы *BallItem* и *BallProcess* (рис. 9.20). Объекты *BallItem* привязаны к списку артефактов. Объекты *BallProcess* привязаны к списку активных процессов. Если объект *BallProcess* получит сообщение *Run*, он обратится к своему двойнику *Item* и слегка его переместит в пространстве. Когда потребуется перерисовка всей доски, все объекты списка артефактов (включающего в себя *BallItems*) получат сообщение *Draw*.

В этом примере, мы обошлись без множественного наследования и соблюли все требования по совместимости относительно артефактов и процессов.

9.4 Паттерны поведения

Третья категория приёмов проектирования относится к так называемым паттернам поведения. Чтобы решать проблемы, возникающие вокруг объектов, существуют различные методы. Мы сфокусируемся на следующих паттернах:

- Сообщение Сообщение рассматривается как объект
- Итератор Обход элементов набора объектов
- Наблюдатель Реагирование на изменение состояния
- Шаблон Алгоритм с внедряемым управлением
- Клон Клонирует объекты
- Персистентность Ввод и вывод объектов
- Расширение Расширение системы в рантайме

9.4.1 Объект-сообщение

Одним из доступных способов работы с сообщениями

являются методы. Но можно относиться к термину "отправка сообщения" буквально: в этом случае сообщением является *пакетом данных* (объект *сообщение*) который рассылается от одного объекта-обработчика к другому. Для исполнения подобного, нам необходимы различные типы объектов-сообщений и метод, в котором эти объекты-сообщения обрабатываются.

Вернемся к нашему примеру с фигурами, прямоугольниками и окружностями. Фигуры могут получать сообщения *Draw*, *Store* или *Move*. Если мы реализуем эти сообщения как объекты, то получим следующую структуру:

```
TYPE
    Message* = ABSTRACT RECORD END (* базовый тип всех
сообщений*)
    DrawMsg* = RECORD (Message) END;
    StoreMsg* = RECORD (Message) rider: OS.Rider END;
    MoveMsg* = RECORD (Message) dx, dy: INTEGER END;
```

Конкретные сообщения, расширяя абстракцию *Message*, содержат фактические параметры в виде атрибутов (полей записи). Записи такого типа могут быть переданы в так называемый обработчик сообщений. Как показано на рис. 9.21, таким обработчиком является метод Figure:

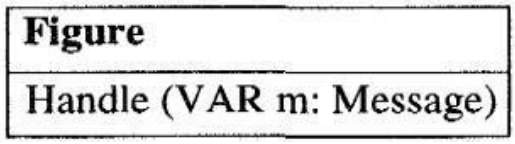


Рис 9.21 Класс Figure с обработчиком сообщений

Обработчик сообщений *Handle* анализирует входящий объект сообщения, реагируя соответственно его динамическому типу. Каждый объект класса *Figure* переопределяет данный обработчик под свои нужды. Например, используемый в классе *Rectangle* обработчик изображен на рис.9.22:

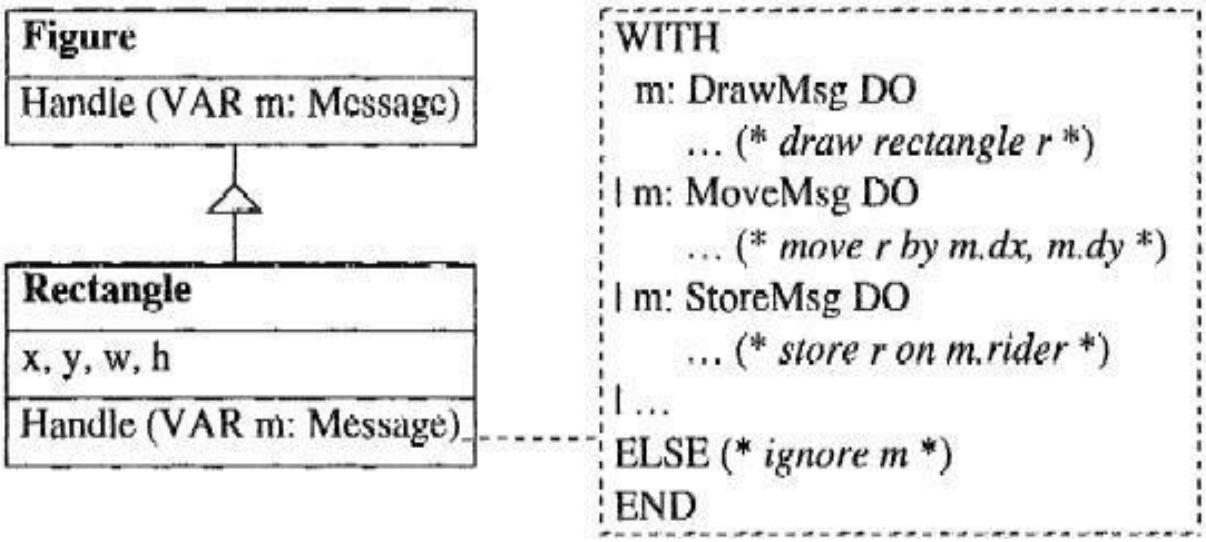


Рис. 9.22 Реализация обработчика сообщений в классе *Rectangle*

Для обработки сообщения *m* будем использовать оператор WITH, использующий варианты, которые можно организовать следующим образом: если *m* имеет динамический тип *DrawMsg*, оператор обратится к своей первой операции DO; если же *m* имеет тип *MoveMsg*, оператор обратится ко второй операции DO; если же совпадения с возможными вариантами отсутствуют, обращение переходит к ветке ELSE. Если последний отсутствует, генерируется ошибка в работе приложения.

В данном примере *Handle* игнорирует неизвестные сообщения: блок ELSE оператора WITH пуст. Однако по-прежнему возможно сгенерировать ошибку или переслать неизвестный тип сообщения обработчику базового класса.

Если же мы захотим отправить сообщение фигуре, мы сможем создать соответствующий объект сообщений и отправить его обработчику фигур:

```
VAR
        t : Figure; move : MoveMsg;
...
move.dx := 10; move.dy := 20;
f.Handle(move);
```

В зависимости от динамического типа объекта *f*, обработчик сообщений будет интерпретировать сообщение *move* определенным образом.

Возвращаемые значения сохраняются в сам объект сообщения. Например, если необходимо вычислить площадь фигуры, мы можем отправить сообщение *getArea*. Обработчик вернет значение площади в атрибут *getArea.value*.

```
TYPE
    GetAreaMsg = RECORD (Message) value : INTEGER END;
VAR
    getArea : GetAreaMsg;
    area : INTEGER;

f.Handle(getArea);
area := getArea.value;
```

В объектно-ориентированном программировании объекты сообщений обрабатываются образом, схожим с обработкой сообщений языком *Smalltalk*. В *Smalltalk* обработчик интерпретирует сообщения и вызывает для их обработки соответствующий метод. Однако, обработчик в *Smalltalk* встроен в саму систему; в Oberon же должен быть реализован программистом. Событийная модель в *Java* также использует объекты сообщений.
Система Oberon была целиком реализована с использованием объектов-сообщений. Так, система Oberon0, которая будет представлена в гл.12 использует сообщения для работы с диалоговыми окнами на экране.

Объекты сообщений, по сравнению с методами, обладают следующими преимуществами:

- Объекты-сообщения являются *пакетами данных*. Они могут быть сохранены и отосланы позже.
- Объект-сообщение может быть передан в процедуру, которая распространит это сообщение другим объектам (которые могут быть неизвестны отправителю). Это называется "широковещание" и легко реализуется с

помощью методов, с одним исключением: отправитель знает каждого получателя и несёт ответственность за получение адресатом сообщения.

- Иногда бывает удобнее, если получатель понимает сообщение - тогда отправителю не надо заботиться о доставке. Предположим, что у нас есть список различных фигур, где сообщение о заливке понимают только прямоугольники и окружности, но не линии. Намного легче (но и гораздо дороже) отправить сообщение о заливке сразу всем объектам. Иначе необходимо сперва проверить, возможно ли передать данное сообщение этому объекту. Такой подход невозможно осуществить, используя методы, поскольку уже компилятор, ещё до запуска программы, проверяет соответствующие методы в классе получателя.
- Наконец, возможно реализовать обработчик сообщений не как метод, а как переменную процедурного типа. В таком случае обработчик можно заменять в течение работы программы, динамически изменяя тем самым поведение самих объектов.

Однако, объекты-сообщения, имеют и недостатки:

- вы не имеете возможности понять, глядя на класс, какие сообщения вы можете ему посылать. Хотя вы и можете угадать, судя по различным типам записей сообщений, но совершенно необязательно, что все типы сообщений будут объявлены в одном модуле. Для полной уверенности, вам придётся взглянуть на реализацию обработчика сообщений.
- Обработчик анализирует сообщения во время выполнения программы с помощью оператора WITH. Ветки оператора WITH обрабатываются последовательно. Это, как правило, медленнее вызова метода, реализуемого путём прямого доступа к таблице методов (см. Приложение А.12.4).
- Отсылка объектов-сообщений приводит к увеличению исходного кода по сравнению с вызовом метода. Сначала

параметры должны быть размещены в записи сообщения, после чего вызывается обработчик сообщений и в конце результаты должны быть извлечены из записи.

```
msg.inPar := ....;
obj.Handle(msg);
... := msg.outPar
```

- То, что ранее рассматривалось как преимущество, может оказаться и недостатком: компилятор не может проверить, понимает ли объект определённое сообщение. Нижеследующий фрагмент кода полностью корректен с точки зрения компилятора:

```
TYPE
    NonsenseMsg = RECORD (Message) END;
VAR
    t : Figure;
    nonsense: NonsenseMsg;
    ...
    f.Handle(nonsense);
    ...
```

В течение времени выполнения, *f* не поймет сообщения *nonsense*. Объект (в лучшем случае) проигнорирует сообщение. В худшем случае, программа остановится с выводом ошибки. Подобная ошибка может скрываться месяцами и трудна для обнаружения.

Объекты-сообщения имеют преимущества и недостатки. Использование методов в большинстве случаев более эффективно, безопасно и читабельно. Но в некоторых ситуациях (например, широковещание) может оказаться выгоднее использовать большую гибкость объектов-сообщений.

9.4.2 Итератор

Зачастую необходимо проделать какое-то действие над рядом объектов, но вы не знаете, как именно обойти все элементы некоторой абстрактной структуры данных (массив, линейный список, дерево и т. д.), которая скрывает конкретную реализацию. Примером такой ситуации может быть класс *Directory*, который управляет набором объектов класса *Element* (Рис. 9.23).

Рис. 9.23. Набор объектов неизвестной реализации

Мы не знаем, как реализован класс-словарь *Dictionary*. Какие возможности у нас есть для вывода на печать всех элементов этого словаря? Простейшее решение - создать метод *PrintAll* класса *Dictionary*, который распечатает все свои элементы:

```
PROCEDURE (VAR d: Dictionary) PrintAll;
    VAR e: Element;
BEGIN
    e := d.firstElem;
    WHILE e# NIL DO e.Print; e := e.next END
END PrintAll;
```

Метод *PrintAll* локален по отношению к классу *Dictionary*, вследствие чего имеет доступ к его реализации. Но такое решение является неудовлетворительным. Для любой другой операции вам понадобится заводить отдельный метод, например *StoreAll* для сохранения всех элементов или *SelectAll* для выбора всех элементов, удовлетворяющих некоему критерию. Вдобавок, эти методы не могут использовать операции, которые определены в потомках класса *Element*.

Другим решением может быть класс *Iterator*, объявленный в том же модуле, где и *Dictionary*, как показано на рис. 9.24.

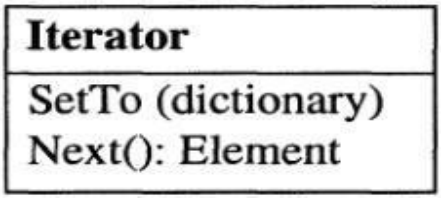


Рис. 9.24 Класс Iterator

Iterator это такой объект, который перемещается по структуре с данными. Метод *SetTo* устанавливает итератор в начало структуры, а метод *Next* даёт последующий элемент. С помощью итератора осуществляется последовательный обход всех элементов *Dictionary*, с применением к ним произвольных операций.

```
Iterator.SetTo(dictionary);
elem := Iterator.Next();
WHILE elem # NIL DO
        elem.Print;
        elem := Iterator.Next()
END
```

Второе решение, хотя и универсально, однако требует реализации алгоритма обхода элементов для каждого клиента. Проблема возникает тогда, когда структура данных, например, дерево, эффективнее просматривать рекурсивно. В таком случае довольно проблематично реализовать переход к следующему элементу.

В рассматриваемой ситуации *Next* возвращает результат типа *Element*. Действительным типом результата может быть расширение *Element* (напр. *MyElement*). С применением охраны типа можно отправлять результату *Next* сообщения для *MyElement*, которые не предусмотрены для *Element*.

```
Iterator.SetTo(dictionary);
elem := Iterator.Next();
WHILE elem # NIL DO
        IF elem IS MyElement THEN
                elem(MyElement).Store(rider)
        END;
```

```
        elem := Iterator.Next()
    END
```

Третий вариант — использовать объекты сообщений. Вы можете обращаться к словарю с сообщением-объектом, которое реализует операцию, применяемую к элементам словаря. Такое сообщение будем рассылать всем элементам. Каждый элемент должен иметь обработчик, который среагирует на сообщение по-своему. К сожалению, данное решение представляется слишком дорогим для простейших приложений, подобным распечатке всех элементов.

Наконец, можно обеспечить словарь универсальным методом *ForAll*, с процедурой в качестве параметра. Эта процедура будет вызвана для всех элементов:

```
PROCEDURE (VAR d: Dictionary) ForAll (P: PROCEDURE(e: Element);
        VAR e : Element;
        BEGIN
                e := d.firstElem;
                WHILE e# NIL DO P(e); e := e.next END
END ForAll;
```

Вызов этой серии методов будет выглядеть примерно так:

```
dictionary.ForAll(Print);
dictionary.ForAll(Store);
```

где процедуры *Print* и *Store* принадлежат клиенту:

```
PROCEDURE Print (e: Element);
        BEGIN
                e.Print
END Print;

PROCEDURE Store (e: Element);
        BEGIN
                e(MyElement).Store(rider)
END Print;
```

В результате возможно применить практически любую операцию ко всем элементам данного набора.

Последнее решение из представленных является простейшим и наиболее читаемым для языков Оберон-2 и Компонентный Паскаль. Некоторые другие языки (напр. Sather) имеют специальные конструкции-итераторы или так называемые объекты-блоки (напр. Smalltalk), позволяющие итераторам быть реализованными еще проще.

9.4.3 Наблюдатель

Наблюдателем назовём объект, которому интересно знать состояние другого объекта. Наблюдатель регистрируется у него для получении информации об изменениях состояния.

Шаблон *Наблюдатель* часто применяется в графических пользовательских интерфейсах. На рис. 9.25. показан пример объекта-измерителя, работающего в диапазоне от 0 до 100. Два графических объекта, ползунок и линейка, регистрируются как наблюдатели измерителя. Когда тот изменяет своё значение, то ползунок со слайдером получают уведомление и обновляют своё графическое представление.

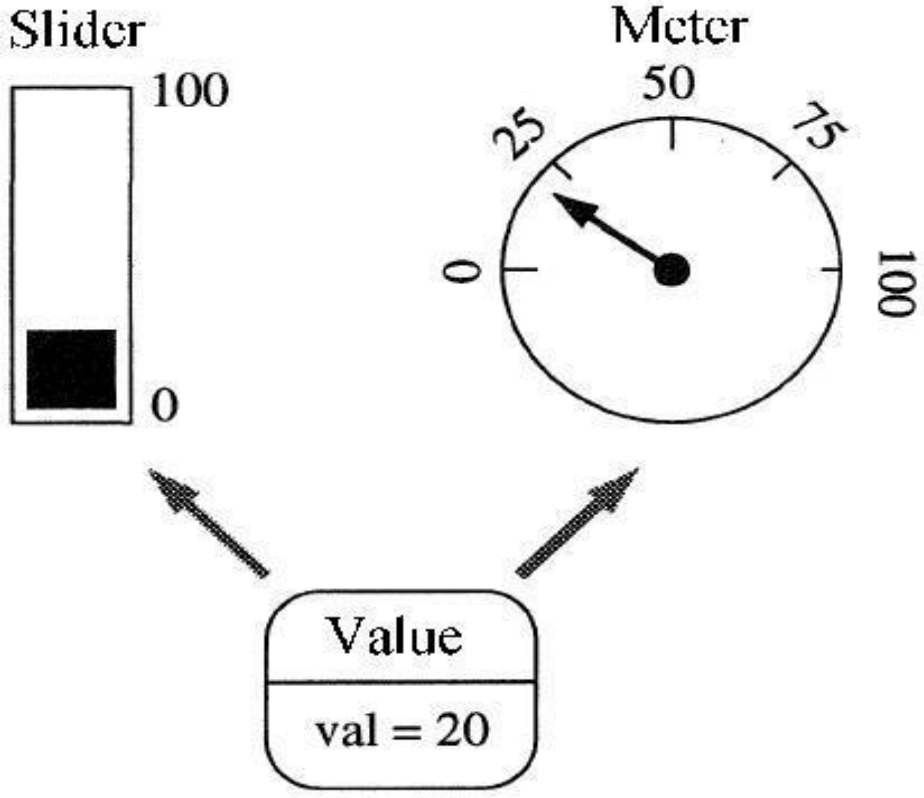


Рис. 9.25 Ползунок и линейка в схеме "наблюдатель измерения"

Шаблон *Наблюдатель* обслуживает и другую задачу. Он гарантирует *согласованность* для всех наблюдателей объекта. Получая мгновенные уведомления о любом изменении объекта, все наблюдатели в любой момент времени могут быть уверены в том, что каждый из них видит актуальное состояние объекта.

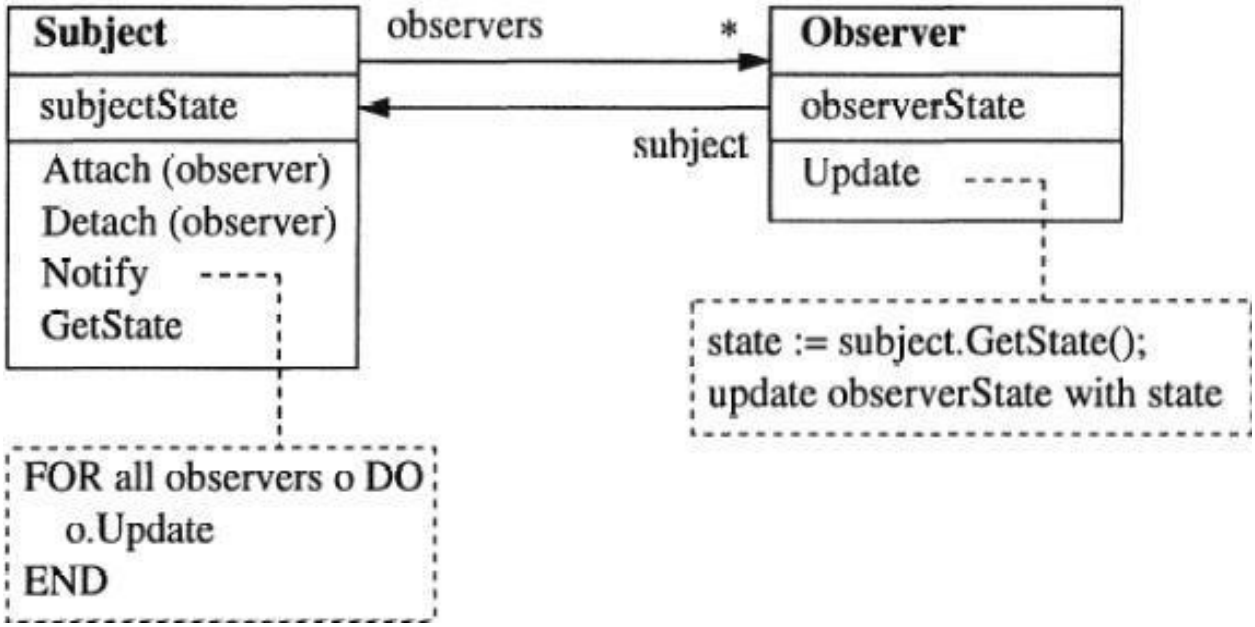


Рис. 9.26 Шаблон Наблюдатель

Диаграмма классов на рис. 9.26 показывает отношения между частями паттерна *наблюдатель*. *Субъект* (наблюдаемый объект) при своём изменении рассылает *Уведомление*. Метод уведомления, в свою очередь, передаёт сигнал *Update* всем зарегистрированным наблюдателям. Наблюдатели запрашивают текущее состояние *Субъекта* вызовом метода *GetState* и обновляют своё собственное состояние.

Важно учитывать тот факт, что наблюдатели могут динамически регистрировать и дерегистрировать себя с помощью методов *Attach* и *Detach*. Следовательно, отношение между *Субъектом* и его наблюдателями является временным и может измениться в любой момент. Может возникнуть такая ситуация, когда не окажется ни одного зарегистрированного

наблюдателя. В этом случае *Уведомление* отключится и никто не будет уведомлён об изменении состояния.

Когда наблюдатель получит уведомление об изменении состояния, ему потребуется информация о том, что же было изменено. Для этого есть два варианта. Первый из них, это т.н. *модель проталкивания*: состояние передаётся как параметр в вызове метода *Update*, то есть наблюдатель получает информацию только о том, что изменилось. Второй вариант заключается в добывании нужной наблюдателю информации посредством вызова *GetState* или аналогичного ему (*модель вытягивания*). Проталкивание более удобно для распространения простых изменений, а вытягивание для более сложных.

На рис. 9.27 показана диаграмма взаимодействия между субъектом и его наблюдателями. Вертикальные линии изображают объекты, горизонтальные - сообщения. Прямоугольники на линиях объектов показывают время существования вызываемых методов. Подобные диаграммы прекрасно подходят для описания динамических свойств программ.

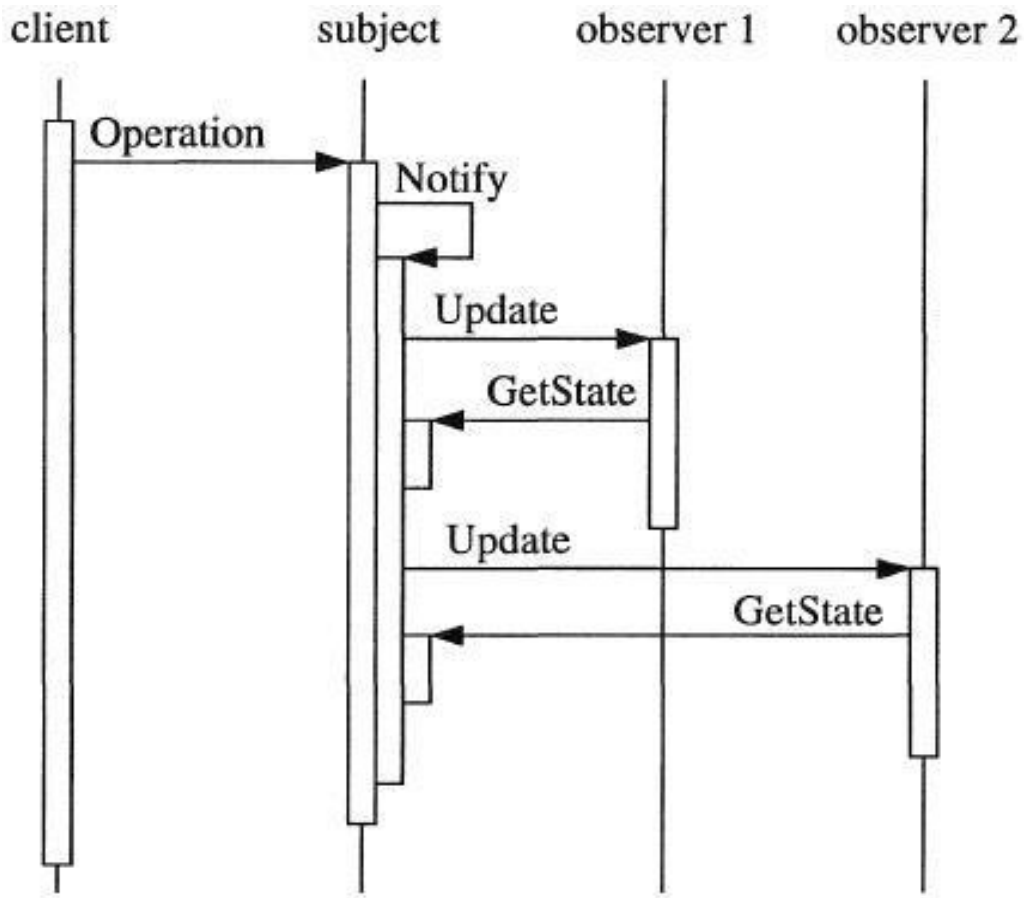


Рис. 9.27 Взаимодействия в паттерне *Наблюдатель*

9.4.4 Шаблонный метод

Метод-шаблон определяет алгоритм в виде последовательностей вызовов абстрактных методов. Исполнение каждого шага алгоритма остаётся открытым и осуществляется путём реализации абстрактных методов в подклассах.

В качестве примера рассмотрим класс *Frame*, относящийся к окнам на экране. Для обновления какого-то участка окна необходимо удалить возможное выделение, задать область отсечения и перерисовать содержимое окна. Эти три шага не зависят от того, текстовое ли окно или же графическое и могут быть объединены в шаблон-метод *Restore*, как показано на рис. 9.28

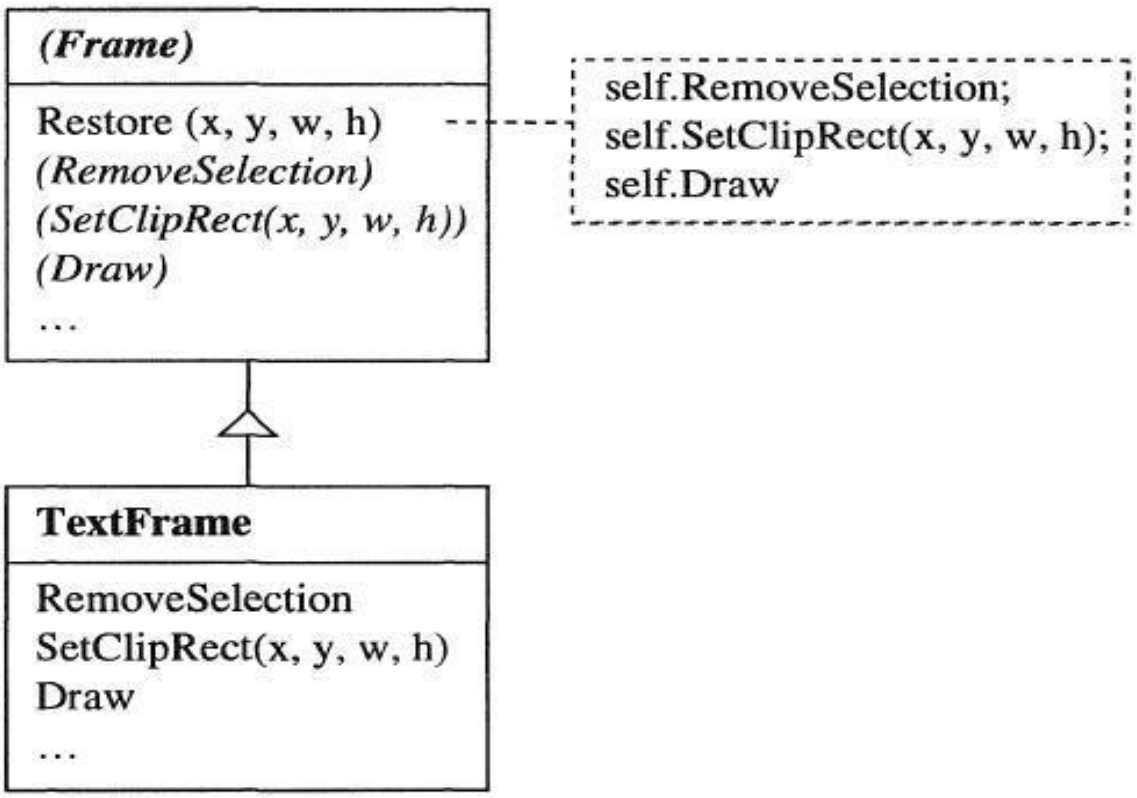


Рис. 9.28. Шаблон-метод Restore

Методы, вызываемые из *Restore*, разумеется, не могут быть реализованы в абстрактном классе *Frame*, поскольку для текстовых и графических фреймов они будут разными. Пример реализации можно увидеть в подклассе *TextFrame*. Тем не менее, в методе *Restore* зафиксирована правильная последовательность вызовов, что даёт возможность послать сообщение *Restore* в текстовый фрейм и не беспокоиться о промежуточных шагах алгоритма отрисовки. И, конечно же, метод-шаблон гарантирует исполнение этих промежуточных шагов в одной и той же последовательности для всех типов фреймов.

Нет необходимости делать все промежуточные шаги полностью абстрактными. Некоторые из них могут воплощать отдельный алгоритм, вызывая пустые методы в особых местах. Программист имеет возможность переопределить эти пустые методы для встраивания своих алгоритмов в общий ход выполнения метод-шаблона. Эти пустые методы называются *хуками* (the hook - крюк, прим. перев.), потому что они позволяют "прицепить" свой код к существующему алгоритму.

Давайте снова посмотрим на пример. Графическое окно имеет метод *TrackMouse*, призванный отслеживать движения мыши и рисовать её указатель. Если вам требуется ограничивать движение указателя путём привязки его к координатной сетке, вы можете вызвать пустой метод *Constrain*. Этот метод может быть переопределён в подклассах таким образом, что координаты мыши будут пересчитаны для попадания в ближайшую точку сетки (рис. 9.29).

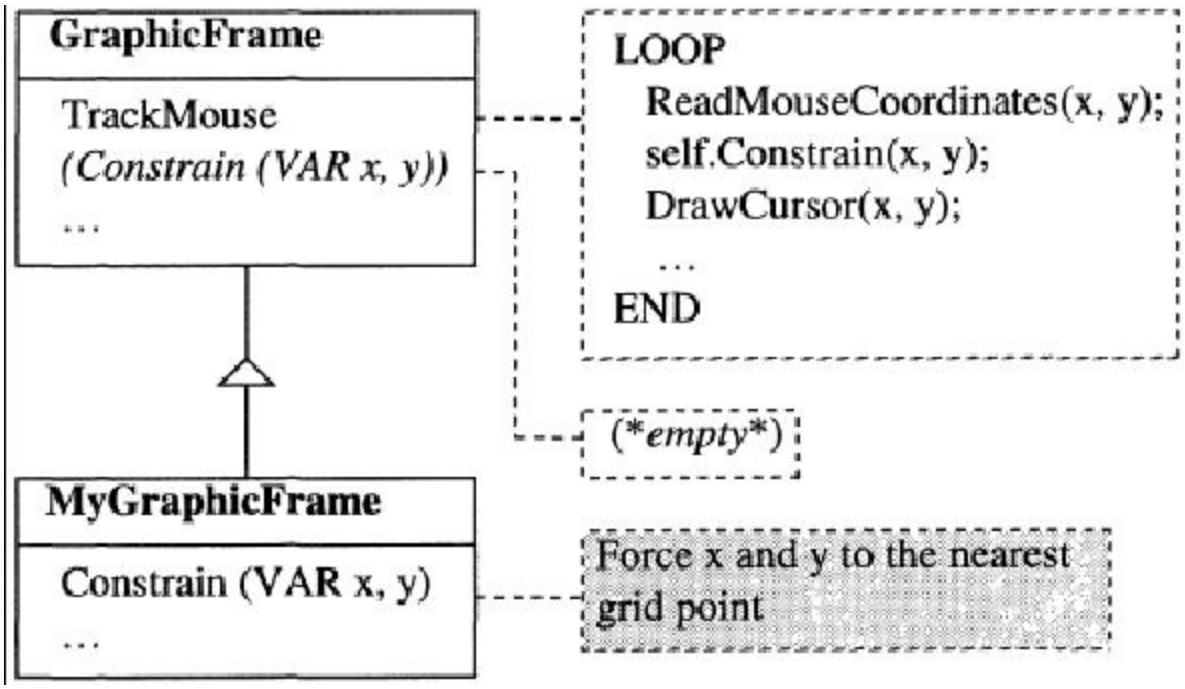


Рис. 9.29. Ограничение как хук в методе TrackMouse

Чтобы сделать алгоритм максимально гибким, надо предоставить столько хуков, сколько нужно. Но если хуков будет слишком много, то может пострадать эффективность алгоритма, а интерфейсом метод-шаблона трудно будет воспользоваться.

9.4.5 Клон

Задача копирования или клонирования объектов кажется тривиальной, но перестаёт быть таковой, если вам неизвестен динамический тип копируемого источника. Как же быть?

В голову приходит очевидное решение - послать объекту сообщение о копировании. При помощи динамического связывания будет вызван метод *Copy* соответствующего типа объекта. Этот метод разместит новый объект и проинициализирует атрибуты оригинальными значениями. Такое, вполне простое решение, имеет подводные камни, показанные на рис. 9.30.

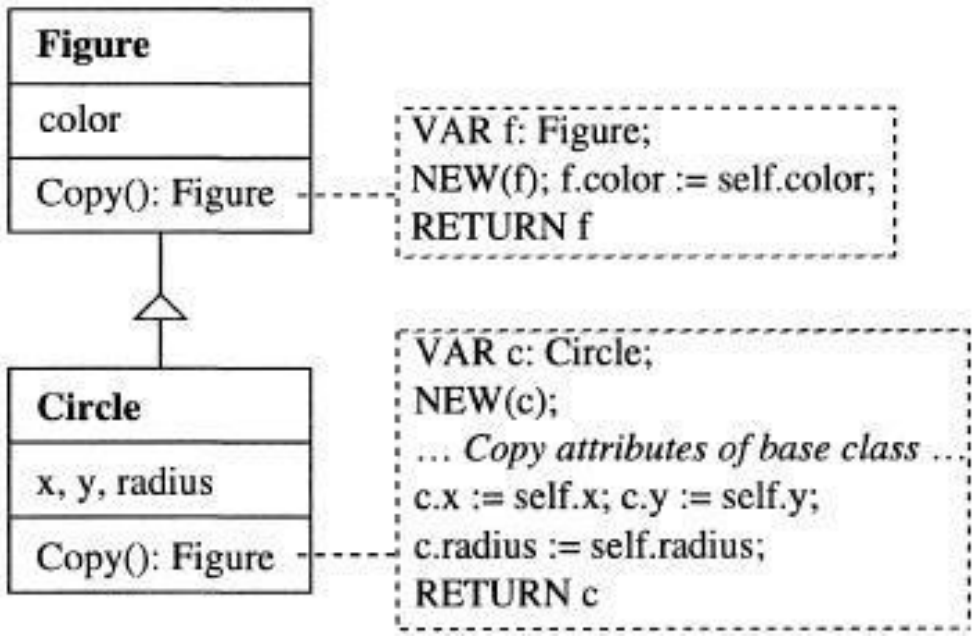


Рис. 9.30 Проблемы копирования объектов

Метод *Copy* класса *Circle* должен скопировать не только атрибуты своего класса, но и базового класса *Figure*. Такое возможно только для экспортированных атрибутов *Figure*. Использовать метод *Copy* класса *Figure* нельзя, так как этот метод создаст новый объект вместо копирования базовых атрибутов в новый объект *c*.

Решение простое: передать новый объект в процедуру *Copy* как дополнительный VAR-параметр. Если этот параметр NIL, то метод *Copy* поймёт, что необходимо создать экземпляр объекта перед копированием атрибутов, а иначе нужно использовать входной объект. Такое поведение проиллюстрировано рис. 9.31.

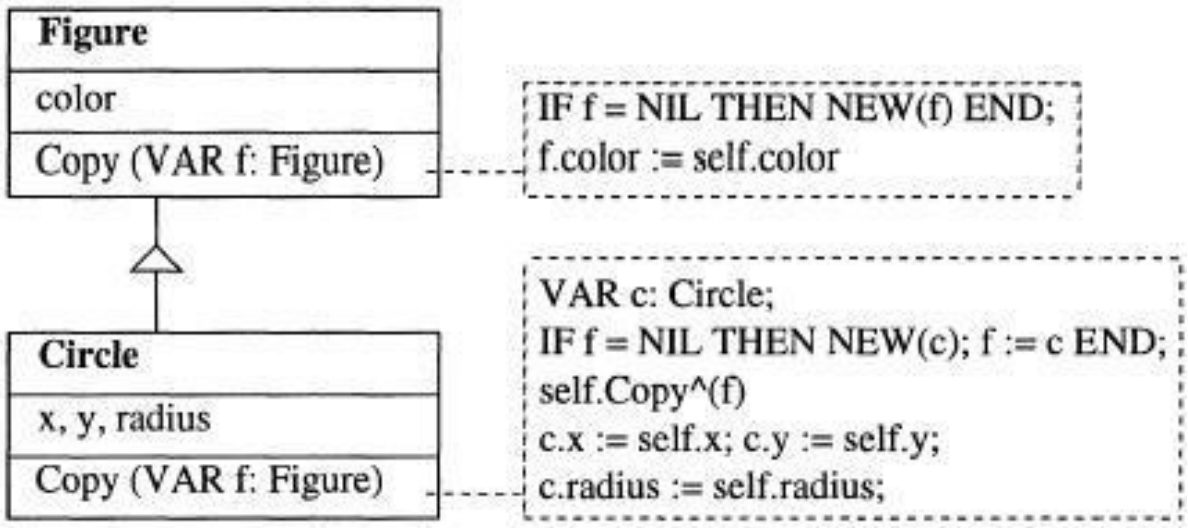


Рис. 9.31 Копирование объектов

Чтобы получить копию f объекта g, вы пишете так:

```
f := NIL; g.Copy(f)
```

Если *g* имеет динамический тип *Circle*, то будет вызван метод *Circle.Copy*. А раз *f* имеет значение NIL, то возникнет новый объект *Circle*. После вызова метода *Copy* базового класса, значение будет отличаться от NIL, никакого нового объекта не появится и скопированы будут только атрибуты цвета.

Немного неудобно принудительно выставлять значение параметра метода *Copy* в NIL. Однако возможно избежать такой необходимости если система времени выполнения обеспечивает операции с типами для создания объектов определённых типов. Система Oberon обеспечивает такие операции в модуле *Types* (см. Приложение Б). BlackBox предлагает подобное в модуле *Kernel*. Соответствующие части этих модулей выглядят так:

BlackBox:

```
DEFINITION Kernel;

TYPE
```

```
        Name = ARRAY 256 OF SHORTCHAR;
        Module = POINTER TO RECORD
                ....
                name-: Name
        END;
        Type = POINTER TO RECORD
                mod-: Module;
                id-: INTEGER;
                base-: ARRAY 16 OF Type;
                fields-: Directory;
                ptroffs-: ARRAY 1000000 OF INTEGER
        END;

PROCEDURE TypeOf(IN rec: ANYREC): Type;
PROCEDURE NewObj(VAR obj: SYSTEM.PTR; t: Type);
PROCEDURE ThisType(mod: Module;
name: ARRAY OF SHORTCHAR): Type;
PROCEDURE GetTypeName(t: Type; VAR name: Name);
...
END Kernel.
```

Type является *дескриптором типа*, т.е. описывает определённые свойства типа, такие как имя или модуль, в котором он объявлен. Если *p* - указатель на запись типа *T*, то вызов *TypeOf*(p) вернёт дескриптор типа *T* (SYSTEM.PTR тоже тип, который совместим с любым типом указателя). *NewObj(t, obj)* создаёт новый объект того типа, который описан дескриптором типа *t*. В Системе Oberon процедура *This (m, name)* и в BlackBox *ThisType(m, name)* возвращает дескриптор типа с именем *name*, объявленным в модуле m.

При помощи низкоуровневых модулей *Types* или *Kernel*, метод *Copy* может быть реализован более элегантно:

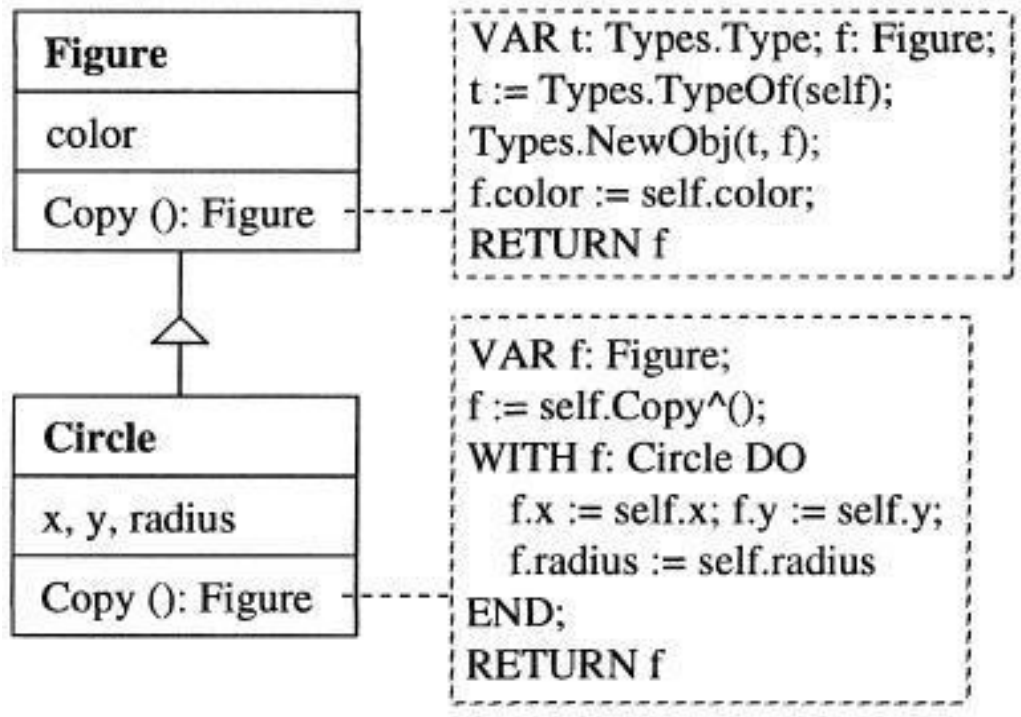


Рис. 9.32 Копирование объектов при помощи системы времени выполнения Types в Системе Oberon, в BlackBox: IMPORT Types := Kernel

Чтобы создать новый объект g, достаточно проделать следующее:

```
f := g.Copy()
```

Если *g* имеет динамический тип *Circle*, то будет вызван метод *Copy* базового класса. При помощи модуля *Types* появится новый объект *f* того же типа, что и приёмник *g*, в нашем случае это экземпляр *Circle*. После копирования атрибута *color*, *f* вернётся в метод *Circle*.*Copy*. Оставшиеся атрибуты скопируются уже в *Circle*. Оператор WITH приведёт статический тип *f* к *Circle*, следовательно, атрибуты *f.x, f.y* и *f.radius* доступны для операции копирования.

9.4.6 Персистентность: ввод\вывод объектов.

Почти каждой программе время от времени требуется записывать объекты в файл и затем читать их оттуда. Эта ситуация поднимает проблему, аналогичную клонированию объектов - как можно читать и писать объекты неизвестного динамического типа?

Запись таких объектов не встречает трудностей. Достаточно послать объекту сообщение *Store* и удостовериться, что объект знает атрибуты, которые он должен записать в файл. Чтение объектов значительно труднее, потому что перед тем как прочитать объект, вы должны его создать. Но как узнать динамический тип создаваемого объекта?

Решение заключается в том, чтобы сохранять не только атрибуты объекта, но и наименование его типа. С помощью модуля *Types (Blackbox: Kernel)* возможно получить имя типа любого объекта и, наоборот, создать объект по имени его типа. Следующие две процедуры, *StoreObj* и *LoadObj* записывают произвольный объект, включая имя его типа, в файл и читают его:

BlackBox:

```
PROCEDURE (VAR r: OS.Rider) StoreObj* (x: Object);
	VAR type: Kernel.Type; name : Kernel.Name;
BEGIN
	IF x = NIL THEN r.Write(0X)
	ELSE type := Kernel.TypeOf(x); Kernel.GetTypeName(type, name );
		r.WriteString(type.name); r.WriteString(type.name); x.Store(r)
	END
END StoreObj;

PROCEDURE (VAR r: OS.Rider) LoadObj* (VAR x: Object);
	VAR name1, name2: ARRAY 32 OF CHAR; type: Types.Type;
BEGIN r.ReadString(name1);
	IF name1 = "" THEN x := NIL
	ELSE r.ReadString(name2);
```

```
            type := Kernel.ThisType(Kernel.ThisMod(name1), name2);
            Kernel.NewObj(x, type); x.Load(r)
        END
END LoadObj;
```

Оба метода предполагают, что все передаваемые им объекты наследуются от класса *Object*, который поддерживает сообщения *Load* и *Store*. Тип *OS.Rider* представляет позиции записи-чтения в файле (см. Приложение B.5). Процедура *StoreObj* определяет дескриптор типа *obj* и записывает имя модуля (*type.module.name*) вместе с именем типа (Oberon System: *type.name*; BlackBox: *Kernel.GetTypeName*) в файл. Процедура *LoadObj* читает имена модуля и типа, после чего определяет дескриптор модуля *mod* и дескриптор типа *type*, и генерирует новый объект полученного типа с помощью *NewObj*.

Процедура *StoreObj* может быть реализована как метод класса *Object*, но вот для *LoadObj* такое невозможно, ведь нельзя послать сообщение объекту, который ещё не существует.

Рис. 9.33 показывает, как реализованы методы *Load*- и *Store*- класса *A* и его подкласса *B* для корректного чтения и записи всех атрибутов. Нужно помнить, что *B* имеет атрибут *b*, являющийся объектом. Этот атрибут становится выходным параметром для *StoreObj*, и считываемым для *LoadObj*. Считывание требует охраны типа, поскольку *LoadObj* возвращает параметр статического типа *Object*.

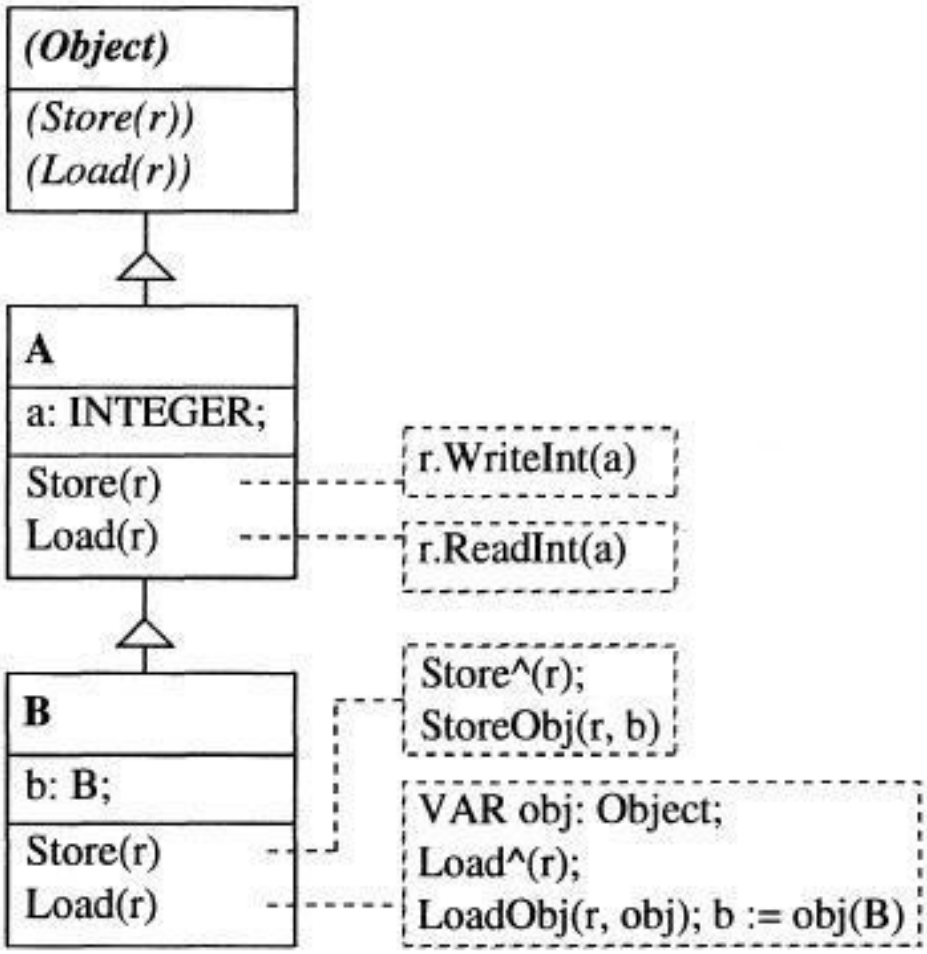


Рис. 9.33 Реализация методов Load- и Store-.

Если система времени выполнения не предоставляет средств конвертации объекта в имя его типа и наоборот, то можно использовать обходной путь. Каждый объект должен поддерживать сообщение *GetTypeName*, возвращая имя типа объекта. В дальнейшем понадобится таблица, где для каждого использованного типа сохраняется прототип с именем его типа. Если вам понадобится объект определённого типа, таблица найдёт нужный прототип и клонирует его. Ясно, что такой способ требует больше усилий со стороны программиста, чем набросанное выше решение.

Чтение и запись объектов является основой для реализации персистентных объектов. Объект называется персистентным, если он переживёт создавшую его программу. Такие объекты используются приложениями типа баз данных.

Они часто переплетаются с другими объектами паутиной графоподобных связей. Чтение и запись такой паутины требует особого внимания для того, чтобы гарантировать однократную запись каждого объекта, даже если на него ссылаются несколько указателей. Методы для решения подобных проблем легко можно отыскать в стандартной литературе по графовым алгоритмам.

9.4.7 Расширение системы «на лету»

В главе 8.3 мы увидели, что графический редактор во время выполнения может быть расширен для того, чтобы получить поддержку новых объектов (прямоугольников, окружностей, линий и пр.), которые не известны в момент первичной реализации редактора. Здесь же мы рассмотрим, как этого добиться в Обероне без непременной выгрузки, перелинковки и перезагрузки расширяемой программы.

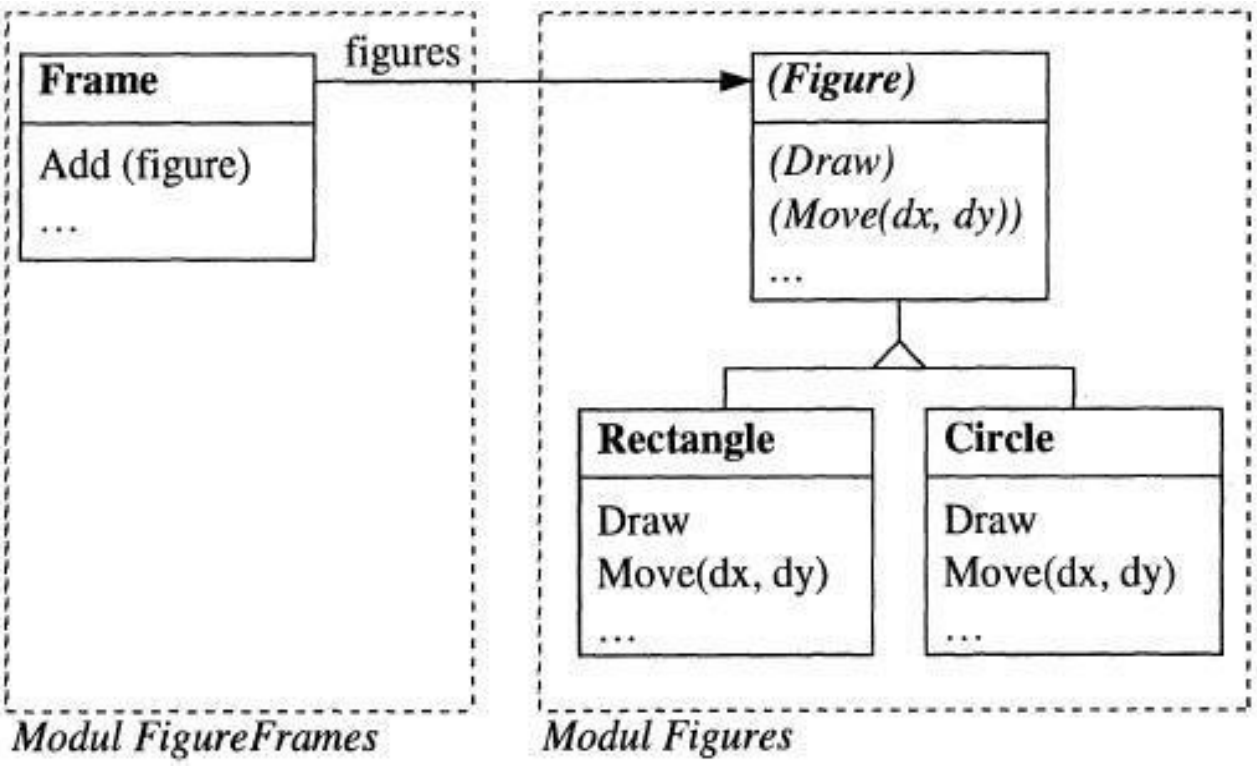


Рис. 9.34 Семейство фигур в графическом редакторе

Давайте вернёмся к примеру из главы 8.3: графический редактор работает с семейством фигур, содержащим прямоугольники, окружности или другие фигуры (рис. 9.34). Предположим, что все классы этого семейства объявлены в модуле *Figures*. Один из модулей редактора носит название *FigureFrames* и содержит класс *Frame*, ответственный за управление фигурами и за отображение их на экране. Для добавления новых фигур имеется метод *Add*.

При реализации ядра редактора пока что нет необходимости знать, фигуры какого типа появятся позже. Редактор может обрабатывать любые подклассы *Figure*.

Если мы хотим расширить редактор поддержкой эллипсов, необходимо проделать следующее:

1. Объявить класс *Ellipse* как подкласс *Figure* и перекрыть унаследованные методы (рис. 9.35)

2. Реализовать команду *New*, которая создаст объект *Ellipse* и добавит его в список с остальными фигурами на фрейме.

```
PROCEDURE New;
        VAR e: Ellipse;
BEGIN
        NEW(e);
        ... (* fille e.x, e.y, e.a, and e.b *)
        FigureFrames.currentFrame.Add(e)
END New;
```

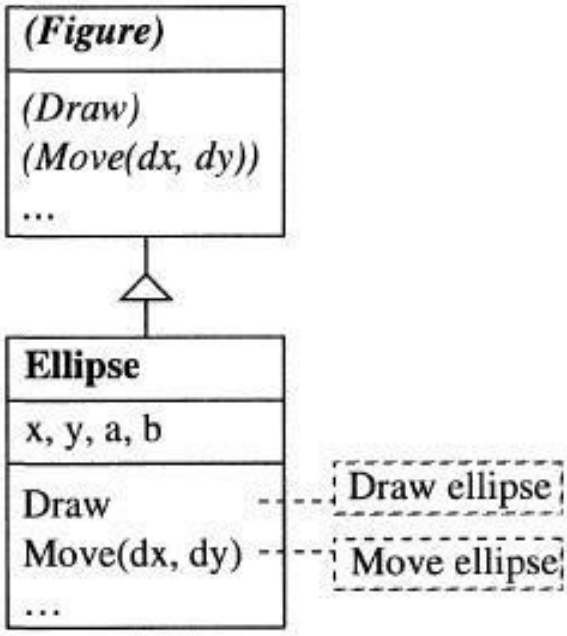


Рис. 9.35 Наследование нового подкласса *Ellipse* от *Figure*

Класс *Ellipse* и команда *New* размещаются в новом модуле *Ellipses*. Нет нужды трогать существующие модули редактора. Чтобы инсталлировать новый объект типа *Ellipse* в окно редактора, вызовите команду *New*. Произойдёт следующее:

1. Если модуль *Ellipses* ещё не был загружен, то он загрузится и расширит собою редактор во время выполнения.
2. Команда *New* выполнится и добавит новый *Ellipse*-объект в список фигур актуального фрейма.
3. Фрейм пошлёт вновь добавленной фигуре (при этом редактор не обязан знать тип новой фигуры) сообщение *Draw*, которое заставит эллипс отрисоваться.

Необходимо помнить, что модуль *Ellipses* будет загружен по запросу и окажется привязан к уже запущенному ядру редактора. Ни модуль *Figures*, ни *FigureFrames* не знают (т.е., не импортируют) модуля *Ellipses*. Следовательно, их обоих можно компилировать и использовать задолго перед тем, как будет создан модуль *Ellipses*. Однако, модуль *Ellipses* импортирует *Figures* и *FigureFrames*, т.е. построен на этих

модулях и расширяет их.

Ядро редактора может использовать неизвестный ему модуль *Ellipses* в результате динамического связывания. Ядро рассматривает объект *Ellipse* как воплощение абстрактного класса *Figure* и взаимодействует с ним посредством сообщений, которые приводят к вызову методов модуля *Ellipses*, лежащего на вершине иерархии импорта. Поэтому такие вызовы называются "восходящими" (up-calls).

Возможность расширять систему во время выполнения, без выгрузки, перекомпиляции и релинковки является одним из главных и сильных преимуществ объектно-ориентированного программирования. Система Oberon (и компонентный каркас BlackBox) с их командами и динамической загрузкой модулей, обеспечивают предпосылки для подобной расширяемости.