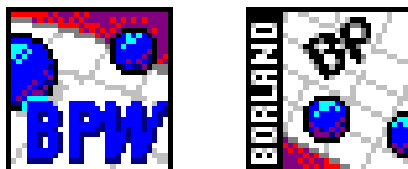


Міністерство освіти України
Львівський державний університет імені Івана Франка

С. А. Ярошко

**Основи об'єктно-орієнтованого програмування
з ілюстраціями на Borland Pascal та
Borland Pascal for Windows**

Тексти лекцій



Львів ЛДУ 1998

Ярошко С. А. Основи об'єктно-орієнтованого програмування з ілюстраціями на Borland Pascal та Borland Pascal for Windows. Тексти лекцій. – Львів: Ред.-вид. відділ ВЦ ЛДУ ім. І. Франка, 1998. – 50 с.

У текстах лекцій розглянуто основи об'єктно-орієнтованої технології розробки програмного забезпечення. На прикладах продемонстровано весь процес розробки від постановки задачі та проектування ієрархії класів до отримання готового продукту. Створена DOS-програма перенесена в середовище операційної системи Windows.

Для студентів факультету прикладної математики та інформатики, механіко-математичного факультету, усіх, хто цікавиться питаннями створення програмного забезпечення.

Рекомендовано до друку
науково-методичною радою факультету
прикладної математики та інформатики
протокол № 12 від 19.11.98

Рецензент	канд. фіз.-мат. наук доц. Р. Є. Рикалюк (Львів. ун-т)
Відповідальний за випуск	канд. фіз.-мат. наук доц. А. О. Музичук
Редактор	М. М. Мартиняк

© С. А. Ярошко, 1998

ЗМІСТ

1. Складність, притаманна програмному забезпеченню
2. Об'єктно-орієнтоване програмування – засіб оволодіння складністю
3. Основні поняття об'єктно-орієнтованого програмування
 - 3.1. Що таке ООП
 - 3.2. Структура програми
 - 3.3. Об'єкти
 - 3.4. Класи
 - 3.5. Взаємодія
 - 3.6. Виконання
4. Успадкування, поліморфізм і ООП
 - 4.1. Поліморфні структури даних
 - 4.2. Класифікація
 - 4.3. Успадкування
 - 4.4. Проектування ієрархії класів
 - 4.5. Тестовий приклад
5. Поліморфізм у дії
 - 5.1. Чим поліморфізм відрізняється від успадкування
 - 5.2. Поліморфізм узагальнює диспетчеризацію
 - 5.3. Динамічне (пізніє) зв'язування
 - 5.4. Обертання поліморфних об'єктів-фігур
6. Розробка і реалізація класу tScreen
7. Програмування для Microsoft Windows
 - 7.1. Графічний інтерфейс користувача
 - 7.2. Керування подіями
 - 7.3. Апаратно незалежна графіка
 - 7.4. Багатозадачність
 - 7.5. Гнучке керування пам'яттю
 - 7.6. Структура Windows-програми
 - 7.7. Обробка повідомлень
 - 7.8. Використання ресурсів
 - 7.9. Посипання на вікно, екземпляр програми, контекст дисплея
8. Програма обертання об'єктів-фігур
 - 8.1. Модифікація класу tScreen для Microsoft Windows
 - 8.2. Головна програма
9. Список літератури

“Більшість книг, присвячених питанням методики і технології об’єктно-орієнтованого програмування (ООП), розпочинають обговорення предмета з пояснення інструментальних засобів деякої конкретної мови програмування. Такий підхід до вивчення ООП часто дезорієнтує: як можуть нові засоби полегшити життя, якщо вони такі складні до застосування? Таке питання виникає природно, якщо намагались вивчити ООП, починаючи із засобів мови і структури об’єктів. Але запровадження ООП у практику починалось не з цього, а з ідеї створити нову структуру цілої програми. Структуру, яка б полегшила побудову і розуміння програми. Тому ООП треба сприймати як новий спосіб мислення і вивчати його, знаходячи і розбираючи цілі закінчені приклади реальних програм.” *Грег Восс* [1].

Багато непорозумінь виникає, коли намагаються пояснити об’єктно-орієнтоване програмування (ООП), роблячи основний наголос на засобах конкретної мови програмування. Однак ООП є не просто новим засобом, а передусім новою технологією структурування, новим способом мислення. Тому вивчення слід починати з концепцій ООП, проілюстрованих наочними прикладами.

Тексти лекцій складаються з двох основних частин. У першій з них (параграфи 2–6) продемонстровано весь процес створення об’єктно-орієнтованої програми: від проектування структури класів об’єктів до написання кінцевого продукту. За мову реалізації вибрано Borland Pascal. Друга частина (параграфи 7, 8) присвячена вивченню основних прийомів побудови Windows-програм та перенесенню розробленої DOS-програми в середовище Windows (з використанням мови Borland Pascal for Windows). Тексти лекцій розраховані на читача, який володіє основними засобами мови Borland Pascal (Turbo Pascal) і має деякий досвід використання операційної системи Windows будь-якої версії. Їх доцільно вивчати разом з методичними вказівками [4], оскільки вони взаємно доповнюють один одного.

1. Складність, притаманна програмному забезпеченню

Очевидно, не всі програмні системи є складні. Є багато програм, задуманих, розроблених, супроводжуваних і використовуваних однією людиною. Але вони, як правило, мають обмежену галузь застосування і коротку тривалість використання. Системи банківських розрахунків, резервування залізничних квитків, операційні системи та інші належать до індустріально розробленого програмного забезпечення (ПЗ). Його створює і вдосконалює колектив авторів, використовує тривалий час багато користувачів. Суттєвою рисою такого ПЗ є його велика складність: практично неможливо одному розробникові охопити всі тонкощі системи.

Можна виділити чотири основні причини складності ПЗ:

Складність проблеми. Прикладні проблеми часто містять складні елементи, до яких ставиться багато різних, часто суперечливих вимог. Ситуація погіршується тим, що розробники ПЗ можуть мати недостатню кваліфікацію в галузі проблеми, а замовники часто не до кінця уявляють, що власне їм потрібно. Додаткова складність зумовлюється зміною вимог до програми в процесі розробки, оскільки вже наявність проекту системи змінює саму проблему і ступінь її розуміння. На жаль, розробку не можна щоразу починати з самого початку, тому великі системи мають тенденцію до еволюції.

Складність керування процесом розробки. Головне завдання розробників полягає у створенні ілюзії простоти, яка захищатиме коритсувачів від складності процесу, який описують. Однак складність проблеми і велика кількість вимог до ПЗ вимагає колективної праці розробників. У цій ситуації важливим завданням є координація робіт і підтримання цілісності основної ідеї.

Гнучкість програмного забезпечення. Програміст може сам забезпечити себе всіма необхідними елементами конструкції ПЗ, які належать до будь-якого рівня абстракції, починаючи з найнижчих. Ця спокуслива властивість та ще й відсутність єдиних стандартів на такі елементи часто призводить до того, що програми починають писати “з нуля”. Тому розробка ПЗ залишається дуже копіткою справою.

Складність опису поведінки окремих підсистем. Програма сучасного (цифрового) комп’ютера є системою зі скінченою кількістю дискретних станів, причому їх може бути дуже і дуже багато. Переходи системи з одного стану в інший не можна моделювати неперервними функціями, тому іноді, у випадку несприятливого збігу обставин, зміна стану однієї частини системи може привести до катастрофічних змін у інших частинах. Адже всеохоплююче тестування великої програмної системи провести просто неможливо.

Прикладом складної системи може бути персональний комп’ютер (ПК). Основними складовими ПК є системний блок, монітор та клавіатура. Відповідно кожна з цих частин теж можна розділити на складові: наприклад, системний блок містить процесор, оперативну пам’ять, накопичувачі на магнітних дисках, блок живлення. Далі можна розглянути влаштування процесора і т. д. Так ми одержимо *структурну ієрархію* ПК. Комп’ютер працює добре, коли злагоджено функціонують усі його частини. Кожна з частин є відносно незалежною від інших.

Складні системи є не просто ієрархічні: рівні ієрархії відображають різні рівні абстракції, що впливають один з одного, будучи деякою мірою автономними. Для кожної конкретної задачі ми розглядаємо відповідний рівень. Наприклад, щоб підготувати за допомогою ПК деякий текст, достатньо лише уявляти загальну будову комп’ютера (найвищий рівень абстракції). Проте таких знань буде замало, щоб написати ефективну програму мовою асемблера.

Можна також навести приклад іншої ієрархії. Різні ПК бувають оснащені різними типами процесорів. Маючи спільну властивість виконувати певний набір команд, процесори бувають різної потужності, різних фірм-виробників тощо. У цьому випадку говорять про *ієрархію типів* процесорів.

2. Об'єктно-орієнтоване програмування – засіб оволодіння складністю

Сучасне програмне забезпечення стає щораз складнішим. Зручний у використанні інтерфейс, розвинені можливості, керованість програми подіями, нові нетрадиційні сфери застосування – все це зумовлює ускладнення ПЗ. Можна сказати, що виготовлення ПЗ, зручного у використанні, робить це ПЗ складним. Сьогодні, щоб побудувати сучасну програму, не достатньо просто об'єднати в послідовність певні машинні інструкції, оператори мови високого рівня чи навіть набори процедур та модулів. Головним стало питання розробки виразної структури програми, придатної до легкої модифікації, вільної від помилок, стійкої до змін. Як сказав Алан Кей, розробник Smalltalk'у, зі зростанням складності архітектура домінує над матеріалами.

Об'єктно-орієнтована технологія створення ПЗ була задумана і розроблена як інструмент подолання складності. Вона успадкувала всі найкращі надбання структурного та модульного програмування, використавши їх для реалізації ряду принципово нових підходів до проектування ПЗ. Головним завданням об'єктно-орієнтованого підходу є забезпечити спосіб структурування програми та керування складними взаємозв'язками між великою кількістю компонентів системи.

Об'єктно-орієнтоване структурування зменшує число зв'язків між компонентами. Воно заставляє об'єкти взаємодіяти через вузькі інтерфейси, що дає змогу легко ізолювати помилки та визначати, котрий з методів є відповідальним за помилку, що виникла.

Об'єкти захищають свої дані від несанкціонованого доступу. Оголошені протоколи взаємодії дозволяють компіляторіві чи інтерпретаторіві попереджувати користувача про несанкціонований доступ до даних і навіть не допустити його. Добре спроектовані інтерфейси класів дають змогу будувати добре модульовані, легко перемішувані компоненти програми.

Найбільшою зручністю технології є безпосереднє проектування об'єктів прикладної галузі в об'єкти програми. Легше моделювати реальний світ у вигляді об'єктів, ніж проектувати його у вигляді процедур.

3. Основні поняття об'єктно-орієнтованого програмування

ООП є технологією структурування, а об'єкти – базовими елементами її конструкції. Однак просте розуміння, що таке об'єкт, чи використання об'єктів у програмі ще не означає, що ви програмуєте в об'єктно-орієнтованому стилі.

Якщо ви вмієте правильно писати процедури і використовуєте їх у програмі, але нехтуєте принципами надійної конструкції, не обмежуєте доступ до спільних даних, то ніхто не гарантує, що ваша програма буде без помилок. Так само і в об'єктно-орієнтованому підході: якщо ви не дотримуєтесь певних принципів, використання об'єктів не зробить програму кращою.

3.1. Що таке ООП

Якщо бути точним щодо розуміння ООП, то це є програмування з надсиланням повідомлень об'єктам невідомого типу. Такі об'єкти можна вибирати з деякої структури даних, наприклад, масиву чи колекції. Всі об'єкти в колекції мають певні спільні характеристики (наприклад, позиція на екрані, здатність переміщуватись, активуватись та деактивуватись). Крім того існує певний перелік повідомлень, на які можуть відповідати усі ці об'єкти. З точки зору програміста вибір об'єкта з колекції не надає інформації про його тип.

Якщо тип об'єкта невідомий, то не можна логічно передбачити, як саме він буде відповідати на отримане повідомлення, адже різні об'єкти можуть реагувати по-різному. Наприклад, у середовищі Microsoft Windows на екрані одночасно можуть бути зображені вікна, іконки, лінійки прокручування, меню, кнопки та контролери інших типів. Кожен з цих об'єктів відповідає на повідомлення користувача, які той надсилає, натискаючи на певні клавіші на клавіатурі чи фіксує мишу. Але у відповідь відбуватимуться різні дії, залежно від того, який об'єкт було вибрано (який з них одержав повідомлення). Наприклад, якщо курсор миші вказував на кнопку розгортання вікна, то у відповідь на фіксацію вікно займе весь екран, якщо ж, можливо, курсор вказував на команду "Save" меню редактора текстів, то поточний редагований файл буде записано на диск.

Отже, надсилання повідомлень об'єктам невідомих типів є потужною технологією програмування. Саме це й означає термін *Об'єктно-орієнтоване програмування*.

Базовими поняттями ООП є інкапсуляція, поліморфізм та успадкування. Об'єкти, розташовані на робочому столі ОС Windows, відомі як *поліморфні*. Це означає, що вони мають певні спільні характеристики, такі, як здатність бути поміщеними в одну колекцію (наприклад, desktop) і здатність відповідати на однакові повідомлення. Поліморфізм безпосередньо пов'язаний з успадкуванням. За допомогою *успадкування* можна визначати нові об'єкти, вказуючи лише, чим вони відрізняються від уже визначених.

Об'єкти *інкапсулюють* дані та процедури в єдину цілісність. Дані містять інформацію про об'єкт, процедури описують правила його поведінки (зокрема, правила доступу до цієї інформації). Об'єднання в одному понятті даних про деякий реальний об'єкт-прототип і операцій над ним робить об'єкт програми замкнутою самодостатньою сутністю, що містить усі необхідні знання про конкретний елемент прикладної галузі.

Об'єкти, як було сказано, інкапсулюють дані та процедури, але об'єктно-орієнтовану програму краще розуміти не як набір процедур, а як множину

об'єктів, які обмінюються повідомленнями (рис. 1). Наголос на об'єктах, а не на діях, приводить до іншої структури програми, ніж у процедурно орієнтованих мовах.

3.2. Структура програми

Технологія покрокового структурного програмування зверху-вниз є вже сталою, добре відомою і часто використовуваною. Спуск від поставленої задачі до процедур в операторах мови програмування є інтуїтивно зрозумілим навіть для початківців, і здебільшого навчання програмуванню приводить до засвоєння такої технології розробки алгоритмів. Кожен крок дає змогу перейти від більш абстрактної процедури до більш конкретної. Така *ієрархічна декомпозиція* приводить до процедурної (деревовидної) структури програми.



Рис. 1. Система складається з об'єктів, пов'язаних між собою передаванням повідомлень. Це зменшує кількість з'єднань між компонентами системи, а тому збільшує її модульність

Об'єктно-орієнтований підхід використовує два основоположні поняття згаданої технології: підхід до програмування зверху вниз і те, що програма є серією процедур, або подій, уведених одна за одною. Об'єкти – початкові структурні одиниці програми. Причому об'єкти реального світу (з галузі прикладної проблеми) більш безпосередньо проектуються у вигляді об'єктів програми, ніж у вигляді процедур. Об'єктно-орієнтовані програми часто називають *моделюванням*, оскільки об'єкти програми відтворюють поведінку своїх реальних прототипів. Об'єкти можна також трактувати як процеси чи завдання. Справді, розробники Smalltalk'у назвали об'єкти *завданнями* і *діяльністю*. Процеси відбуваються всередині об'єктів, а інформація передається

між ними. Одні об'єкти можуть запускати процеси в інших шляхом надсилання *повідомлень*. Об'єктно-орієнтована програма є набором об'єктів, що спілкуються.

При дотриманні процедурного стилю програмування всі дані, в ідеалі, повинні б бути максимально приховані в процедурах, а обмін інформацією відбуватися через інтерфейси. Однак насправді дуже часто спільні дані не можна приховати, бо є забагато процедур, що їх потребують, або ці дані завеликі. Тоді такі дані стають глобальними, що є джерелом помилок, бо доступ до глобальних даних може здійснюватись з усіх точок програми (звертання до даних не локалізовані).

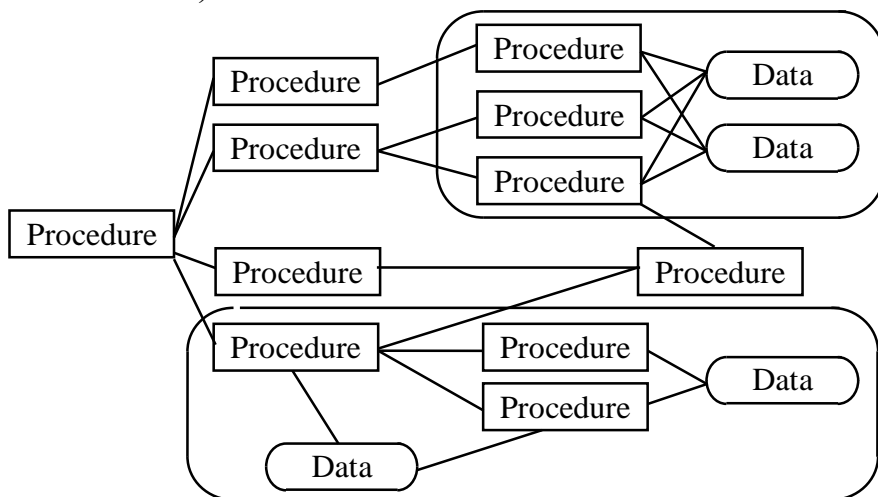


Рис. 2. Процедури можуть використовувати спільні глобальні дані. Такі дані та процедури приховуються в окремих модулях

Концепції приховування та абстрагування даних введено у зв'язку з проблемою необмеженого доступу до глобальних даних. На рис. 2 схематично зображено технологію об'єднання груп процедур та глобальних даних, з якими вони оперують. Це можна робити і без спеціальних засобів мови програмування, але

ефективною технологія буде лише тоді, коли такі засоби є.

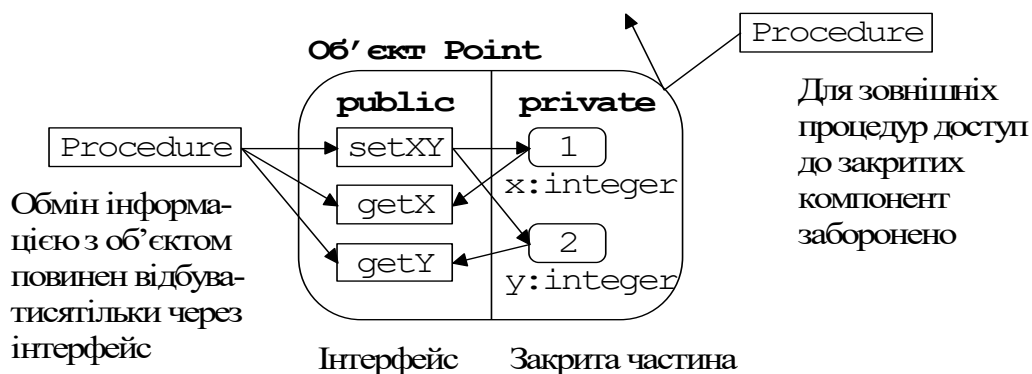


Рис. 3. Об'єкт Point (точка) ілюструє приховування даних. Об'єкти характеризуються своїм інтерфейсом, який зазначає, що вони вміють робити, але не дає інформації, як це реалізовано

Структурним компонентом, який об'єднує дані і процедури, є об'єкт. Він поділений на дві частини: доступну (*public*) і закриту від доступу (*private*). Доступ до закритих даних дозволено лише процедурам усередині об'єкта. Це означає, що доступ до *private* даних можливий тільки через *public* процедури. Таке пов'язування процедур з відповідними даними називається *інкапсуляцією*.

Процедури об'єкта називають також *методами*. Вони “утворюють оболонку”, чи “капсулу”, навколо даних, захищаючи їх від зовнішніх втручань (рис. 3).

3.3. Об'єкти

Як уже зазначено, об'єкти є початковими структурними одиницями в ООП. Вони можуть перебувати в певному *стані*. Свій стан об'єкти пам'ятають за допомогою власних компонент-даних. Ці компоненти називають також *полями*. Імена полям та процедурам всередині об'єкта дають так, щоб полегшити розуміння представлення та призначення об'єктів. На рис. 3 показано приклад того, як виглядає об'єкт. Тут зображено широко вживаний у багатовіконних системах, графічних редакторах та іншому об'єкт Point (точка).

На схемі показано лише три методи, які дають змогу отримувати та задавати числові значення полів. Справжній об'єкт повинен би містити набагато більше методів, проте така спрощена структура дасть змогу легше зрозуміти структуру та поведінку типового об'єкта. Нижче наведено оголошення типу Point мовою Pascal:

```

type Point = object;
  private x,y: integer; {поля даних закриті від доступу}
  public           {методи забезпечують взаємодію з об'єктом}
  procedure setXY(a,b: integer); {задати значення x, y}
  function getX: integer;        {отримати значення x}
  function getY: integer;        {отримати значення y}
  end; {Point}
procedure Point.setXY;           {тіло методу в Borland Pascal}
  begin x:=a; y:=b end;         {оголошується окремо після}
function Point.getX;           {оголошення об'єктового типу}
  begin getX:=x end;
function Point.getY;
  begin getY:=y end;

```

Це оголошення описує внутрішні числові та процедурні компоненти об'єкта. У цій реалізації об'єкт точка містить дві цілочислові компоненти *x* та *y*, які відображають її декартові координати. Їх значення дозволено змінювати лише процедурі *setXY*. Функції *getX*, *getY* використовуються для отримання відповідних значень.

Як і процедури, об'єкти можуть обробляти інформацію. Однак від процедур їх особливо відрізняє те, що об'єкти мають *життєвий цикл*. Вони можуть бути створені і знищені. Поки об'єкт існує, ви можете використовувати будь-які його *public* елементи. Після того, як об'єкт знищено, відведена йому пам'ять звільняється, і надалі неможливо є спілкуватися (зв'язуватися) з ним.

3.4. Класи

Необхідною властивістю об'єктно-орієнтованої мови є можливість оголошувати користувачеві нові типи даних. На відміну від таких мов, як FORTRAN чи Algol ("прародича" Pascal'ю), які давали змогу оперувати з даними лише стандартних типів, Pascal дає змогу будувати на базі стандартних нові, структуровані типи – *типи, оголошені користувачем*. Одним з таких типів є *об'єктний тип*, або *клас*. Клас містить опис структури об'єкта: імена і типи його полів, оголошення доступних і закритих методів. Доступні методи утворюють *інтерфейс* класу.

Оголошення класу не створює ніяких об'єктів, але оголошений клас можна використати для такого створення. Об'єкти ще називають *екземплярами* класу. Створення екземпляра ще не задає значень його полів, їх треба ініціалізувати одразу після створення.

Процес програмування в об'єктно-орієнтованій мові передбачає такі етапи:

- оголошення класів, які визначають структуру та поведінку об'єктів;
- створення об'єктів – екземплярів оголошених класів, задання їхніх полів;
- організація комунікації між об'єктами у вигляді послідовності повідомлень.

Екземпляри створюють за шаблоном класу, вони містять власні копії полів, визначених в оголошенні класу. Часто їх називають змінними екземпляра. Вони існують всередині екземпляра (не в класі).

Процедури, навпаки, насправді існують поза екземпляром. Немає сенсу копіювати їх знову і знову, бо вони однакові для всіх екземплярів класу. Але доступ до цих процедур завжди здійснюється через об'єкти класу, в якому вони оголошені.

3.5. Взаємодія

Об'єкти взаємодіють між собою, надсилаючи *повідомлення*. Кожному повідомленню відповідає певний метод. Повідомлення та методи є двома сторонами одного поняття. *Методи* є тими процедурами, які активізуються, коли об'єкт отримує повідомлення. Крім виконання самого виклику, повідомлення також передає від об'єкта до об'єкта інформацію, як вхідний параметр процедури чи вихідний результат функції.

За термінологією традиційного програмування повідомлення є *викликом* однієї процедури іншою, метод – код процедури, що викликається. Спочатку зміна в термінології сприймається як щось очевидне, але тут прихована причина того, чому традиційні мови не підтримують поліморфізм і скерування повідомлень, отриманих об'єктом, до відповідних методів під час виконання програми. Така термінологія полегшує також проектування систем в термінах, близьких до реального життя.

Повідомлення та методи дозволяють виконати розподіл праці: один об'єкт лише надсилає повідомлення, інший – застосовує відповідні методи для їхньої обробки. Кожен діє у своїх межах.

3.6. Виконання

Виконання безпосередньо пов'язане з методами. Іноді методи називають член-функціями, оскільки вони є членами об'єкта. Методи можуть бути `public` (для забезпечення інтерфейсу) і `private` (для виконання всієї "чорної роботи"). Як звичайно, один `public`-метод викликає одного або кілька `private`-методів.

Повідомлення спонукають до дії методи об'єкта, а ті передають назовні нові повідомлення. Повідомлення об'єкта-передавача і відповідні методи об'єктів-отримувачів мають однакові імена, тому іноді важко розрізнити повідомлення і методи. Проте, коли поліморфізм починає діяти, одне повідомлення може викликати багато різних методів, які мають однакові імена і набори параметрів.

Вузкий інтерфейс об'єкта не дає лініям зв'язку в складній системі заплутуватися і перетинатися. Якщо *об'єкт* бере керування, щоб виконати певне повідомлення, то це означає, що на його відповідальності є виконання відповідного методу. Зовнішні об'єкти не можуть знати, які процеси відбуваються всередині.

Підсумовуючи сказане, можна дати таке означення об'єктно-орієнтованого програмування: *ООП – це методологія програмування, яка ґрунтується на представленні програми у вигляді сукупності об'єктів, кожен з яких є реалізацією певного класу, а класи утворюють ієрархію за принципами успадкування.*

4. Успадкування, поліморфізм і ООП

Як було зазначено, об'єкти інкапсулюють дані та дії і передають повідомлення один одному. Класи використовують для створення об'єктів. Проте застосування самих лише об'єктів і класів не вичерпує всього об'єктно-орієнтованого програмування. Разом з інкапсуляцією ООП залучає успадкування та поліморфізм. *Успадкування* класифікує об'єкти за їх спільними властивостями. *Поліморфізм* дає змогу надсилати повідомлення об'єктам невідомого типу.

4.1. Поліморфні структури даних

Ми визначили ООП як програмування з надсиланням повідомлень об'єктам невідомого типу. Насправді тут є деяка неточність. Хоча справжній тип об'єкта може бути невідомий, проте програміст, який надсилає ці поліморфні повідомлення, знає, що всі об'єкти належать до деякої поліморфної структури даних (колекції) і мають конкретний набір спільних властивостей, щонайменше однаковий перелік імен методів у своїх інтерфейсних частинах. Такий перелік імен називають *протоколом взаємодії* об'єктів.

Наприклад, протокол взаємодії поліморфних об'єктів Трикутник, Коло, Прямокутник може містити імена методів рисування, стирання, задання центра та переміщення. При цьому методи з однаковими іменами можна реалізувати по-різному для різних об'єктів. Зокрема, різної реалізації потребують методи рисування та стирання. Кожен з об'єктів може містити також унікальні, властиві лише йому, методи, відмінні від усіх методів інших об'єктів.

Поліморфізм експлуатує засоби, спільні для множини об'єктів. Успадкування забезпечує засоби вираження цих властивостей тобто засоби оголошення протоколу взаємодії. Поліморфізм гарантує, що всі об'єкти відповідають на згадані повідомлення своїм власним способом.

Ще один приклад поліморфної колекції - вікна різних типів і форм на робочому столі ОС Windows. Легко бачити, що спільними властивостями для них є наявність рамки, заголовку, прямокутна форма та ін. Це *зовнішні* властивості. Їх можна виявити, не взаємодіючи з системою. Інша група властивостей визначає поведінку вікон. Наприклад, іконка (піктограма) теж є формою вікна, хоч і має інший зовнішній вигляд. Адже іконки мають такі ж *поведінкові* властивості, як і вікна: здатність займати певну позицію на екрані, відповідати на фіксацію (клацання) або подвійну фіксацію миші та ін. Об'єкти-вікна повинні "знати", як змінювати свій розмір та розташування, підтримувати певний загальний спосіб керування перемиканнями між вікнами різних програм. Кожне з вікон (у тому числі й іконки) повинне вміти обробляти повідомлення, які надходять до нього у результаті вибору команд із системного меню. Але робити це вони можуть по-різному: в системному меню іконок команда згортання є недоступною, а для розгорнутого на весь екран вікна не доступна команда розгортання.

ООП є стилем програмування, яким передбачено мінімізувати складність програмної системи за допомогою зменшення числа зв'язків між компонентами системи. Для цього визначають прості протоколи взаємодії, які працюють в межах вузьких інтерфейсів об'єктів-компонент.

4.2. Класифікація

Щоб взаємодія була простою, вона має бути загальною. Спеціальні (особливі) повідомлення тимчасово відкидають. Об'єкти зі спільними властивостями об'єднують у класи. Ці спільні властивості визначають як абстрактний *базовий об'єкт* – екземпляр базового класу. Властивості базового класу успадковують від нього всіма породжені класи. Після успадкування кожен об'єкт визначає, чим він відрізняється від інших. Спеціальні випадки обробляють створенням підкласів.

Базовий клас визначає загальні властивості, підкласи, або *породжені* класи, визначають спеціальні властивості.

Класифікація належить до першого етапу створення об'єктно-орієнтованої програми (етап оголошення класів). Його завданням є визначити певний спосіб впорядкування групи об'єктів і задати протокол взаємодії.

Нехай потрібно класифікувати групу об'єктів – геометричних фігур (кіл, прямокутників і трикутників) різного кольору та розміру. Для цього можна використати різні їхні властивості. Якщо програма маніпулюватиме об'єктами залежно від їхнього розміру, то в базовому класі слід оголосити поле і методи для підтримки розміру:

```

type Shape = object {Фігура}
  private size: integer;          {...}
  public procedure setSize(s: integer);
    function getSize: integer; {...}
end; {Shape}

```

Усі породжені класи будуть їх успадковувати.

Якщо ж розмір є суттєвим не для всіх фігур, а лише, наприклад, для трикутників, то згадані поле і методи треба оголошувати не в базовому, а у відповідному породженому класі.

Розглянемо інші можливості. Ви, наприклад, можете потребувати звертатися до підмножини об'єктів певного кольору. У цьому випадку базовий клас повинен містити поле *color* методи *getColor*, *setColor* для отримання та задання кольору.

Допустимими є також інші критерії класифікації. Наприклад, цілком можлива ситуація, коли потрібно організувати об'єкти відповідно до їхньої форми. У цьому випадку постає певна проблема, адже під час оголошення класу *Shape* ще не відомі типи всіх породжених з нього фігур. За аналогією з попередніми випадками спробуємо вирішити її за допомогою імітації: оголосимо в базовому типі цілочисельне поле, яке буде містити номер типу фігури.

Звичайно, така організація можлива, однак насправді програма, що використовує інформацію про тип об'єкта, буде дуже ненадійною, якщо ця інформація зберігається лише всередині самого об'єкта. Нагадаємо, що причиною класифікації є спрощення протоколу передавання повідомлень. Тобто для виконання однакових дій з різними об'єктами використовуються однакові повідомлення. Наприклад, нам потрібно переміщувати фігури на екрані. Реалізація методів переміщення кіл, прямокутників і трикутників повинна бути різною, а імена методів, для зручності, – однаковими. Можна, звичайно, спробувати організувати виклик потрібного методу в класі *Shape* за допомогою перемикача (*case <поле-тип> of...*). Проте, по-перше, ми ще не знаємо імен усіх підкласів, а, по-друге, така система не зможе розширюватись. Реальні ж системи мають кілька базових типів і десятки породжених.

Справжня об'єктно-орієнтована програма вирішує поставлену проблему скерування поліморфних повідомлень до відповідних методів, яку ми будемо називати *проблемою диспетчеризації*, іншим способом. Об'єктно-орієнтовані мови підтримують спеціальний засіб, відомий, як *пізнє* чи *динамічне зв'язування* – зв'язування з кодом потрібного методу під час виконання програми. Пізнє зв'язування використовує інформацію про фактичний тип поліморфного об'єкта, не потребуючи звертання до його полів.

4.3. Успадкування

Успадкування і поліморфізм є засобами для організації ієрархії та спрощення взаємодії. Успадкування дає змогу формально визначати спільні риси множини об'єктів, а поліморфізм – спільний протокол для взаємодії з об'єктами однієї ієрархії.

Спільні засоби та характеристики визначаються в базовому класі, підкласи є його спеціалізаціями. Вони *успадковують* все визначене в ньому. Створення підкласів з успадкуванням полів та методів називається *програмуванням з успадкуванням*. Породжені класи в свою чергу можуть бути базою для породження підкласів. Ієрархія типів звичайно має вигляд дерева із більш загальними властивостями в корені і більш спеціальними в листках. Таке дерево називають *ієрархією класів*, або *ієрархією типів*.

4.4. Проектування ієрархії класів

Розглянемо розвиток ієрархії типів на прикладі класифікації множини геометричних фігур під час розв'язування такої задачі.

ЗАДАЧА Створити програму мовою *Borland Pascal*, яка зображає на екрані дисплея групу заданих кіл, прямокутників та трикутників і обертає їх навколо заданої точки (наприклад, навколо початку системи координат). Оголосити і використати в програмі поліморфні об'єкти – геометричні фігури. Перенести написану *DOS*-програму в середовище *Microsoft Windows*.

Трикутники вважаються рівнобедреними. У початковий момент основи трикутників та прямокутників паралельні до осі абсцис.

Щоб зробити програму максимально незалежною від різних компіляторів та графічних бібліотек, використаємо абстрактний клас *tScreen*, який реалізуватиме графічний дисплей з простим але достатнім набором можливостей. Передбачимо такі його властивості:

- екран графічного дисплея має розмір 100×100 пікселів;
- з екраном пов'язана прямокутна система координат з початком у лівому нижньому куті;
- протокол взаємодії класу *tScreen* містить функції ініціалізації та завершення роботи з графікою, очищення екрана, задання кольорів фону і зображення, рисування прямих ліній та кіл;
- програма буде використовувати єдиний екземпляр *screen* класу *tScreen*.

Ми побачимо, що клас *tScreen* є добрим прикладом того, як інкапсуляція функцій і даних може надати програмістові найбільшу свободу дій. У даному випадку ми не будемо залежати від конкретної реалізації графічного екрана. З іншого боку, незалежно від користувачів класу *tScreen*, його розробники можуть змінювати внутрішнє представлення класу, пристосовуючи його до конкретної апаратури та операційної системи.

Для обговорення успадкування на даний момент деталі визначення класу *tScreen* є несуттєвими, тому ми подамо їх пізніше, а призначення методів класу буде зрозумілим з їхніх назв. Ми покажемо процес створення ієрархії класів геометричних фігур. Він, звичайно, є спрощеним прикладом роботи із зображуваними на екрані об'єктами, що мають спільні властивості, але обговорювані в ньому принципи застосовують і для створення складніших систем, таких як Microsoft Windows або робочий стіл Smalltalk.

Ми поставили завдання написати програму обертання групи геометричних фігур навколо точки (0,0). Цими фігурами будуть коло, прямокутник, трикутник. Реалізуємо їх як екземпляри однієї ієрархії класів *tShape*. Відповідні класи фігур назовемо *tCircle*, *tRectangle*, *tTriangle*.

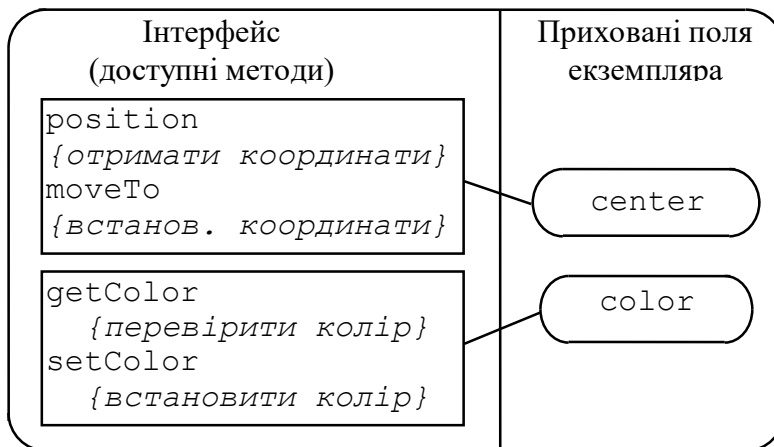


Рис. 4. Базовий клас *tShape* (фігура)

Наступний крок – встановлення спільних властивостей об'єктів у контексті зображення їх на екрані дисплея. Очевидно, що кожна з перелічених фігур повинна займати певну позицію на екрані і може мати свій колір. Перше наближення базового класу схематично зображене на рис. 4.

Реалізація класу *tShape* мовою Borland Pascal може мати такий вигляд:

```

type Colors = 0..15 {Black..White};
      tPoint = record x,y:integer end;
type tShape = object
      function getColor: Colors; {методи} { | }
      procedure setColor(c: Colors); { |протокол }
      procedure position(var p: tPoint); { |взаємодії}
      procedure moveTo(p: tPoint); { | }
      private
      center: tPoint; {поля}
      color: Colors; end; {tShape}
      function tShape.getColor; { | }
      begin getColor:=color end; { | }
      procedure tShape.setColor; { | }
      begin color:=c end; { |реалізація}
      procedure tShape.position; { | методів }
      begin p:=center end; { | }
      procedure Shape.moveTo; { | }
      begin center:=p end; { | }

```

Екземпляри класу *tShape* не є вживаними самі по собі, бо не вміють нічого іншого, як запам'ятовувати та повідомляти інформацію про координати центру та колір фігури. Натомість об'єкти класів, породжених з *tShape*, зможуть мати цікавішу поведінку.

Схема на рис. 5 зображає структуру класу *tCircle*, який будемо використовувати для створення об'єктів-кіл. У її верхній частині показано успадковані елементи, а в нижній – специфічні, ті, які підтримують властивості, притаманні тільки колам.

Клас *tCircle* успадковує з батьківського *tShape* всі поля і методи. Крім того він повинен містити поле для зберігання радіуса кола та методи доступу до нього. Крім них, на схемі показані також методи рисування та стирання кола. Оскільки коло є реальною, а не абстрактною фігурою, то об'єкт повинен містити методи рисування та стирання (*draw* і *hide*). Для доступу до полів *center* і *color* базового типу вони використовують відповідні методи з інтерфейсної частини *tShape*.

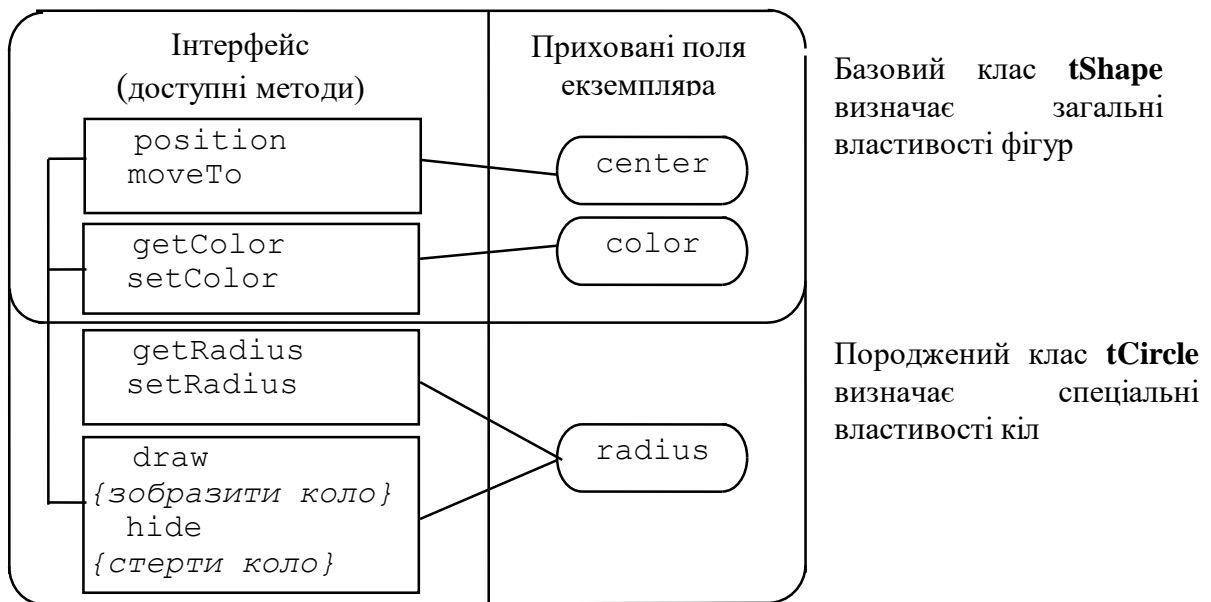


Рис. 5. Клас (*tCircle*) успадковує спільні властивості та визначає власні

Підтримка радіуса є специфічною особливістю класу *tCircle*, оскільки інші фігури (прямокутник, трикутник) характеризуються іншими геометричними величинами. Методи рисування та стирання, очевидно, є різними для різних фігур. Оголошення класу *tCircle* може мати такий вигляд:

```

type tCircle = object(tShape)
    function getRadius: ineger;
    procedure setRadius(r: integer);
    procedure draw;
    procedure hide;
    private
    radius: integer;
    end; {tCircle}
function tCircle.getRadius;
    begin getRadius:=radius end;
procedure tCircle.setRadius;
    begin radius:=r end;
procedure tCircle.draw;
    var saveColor: integer; p: Point;
    begin position(p);    {p-центр кола}

```

```

    {зберегти поточний колір рисування екрана}
    saveColor:=screen.foreground;
    {встановити для рисування колір об'єкта-кола}
    screen.setForeground(getColor);
    {зобразити коло, використовуючи метод класу tScreen}
    with screen do Circle(p,radius);
    {відновити колір рисування екрана}
    screen.setForeground(saveColor)
end; {draw}
procedure tCircle.hide;
var saveColor: integer; p: Point;
begin position(p); {p-центр кола}
    {зберегти поточний колір рисування екрана}
    saveColor:=screen.foreground;
    {встановити для рисування колір фону}
    screen.setForeground(screen.background);
    {стерти коло, використовуючи метод класу tScreen}
    with screen do Circle(p,radius);
    {відновити колір рисування екрана}
    screen.setForeground(saveColor)
end; {hide}

```

Клас *tRectangle*, як і *tCircle*, успадковує спільні властивості безпосередньо з базового класу (верхня частина схеми на рис. 6) і містить специфічні, притаманні лише йому поля та методи (нижня частина схеми).

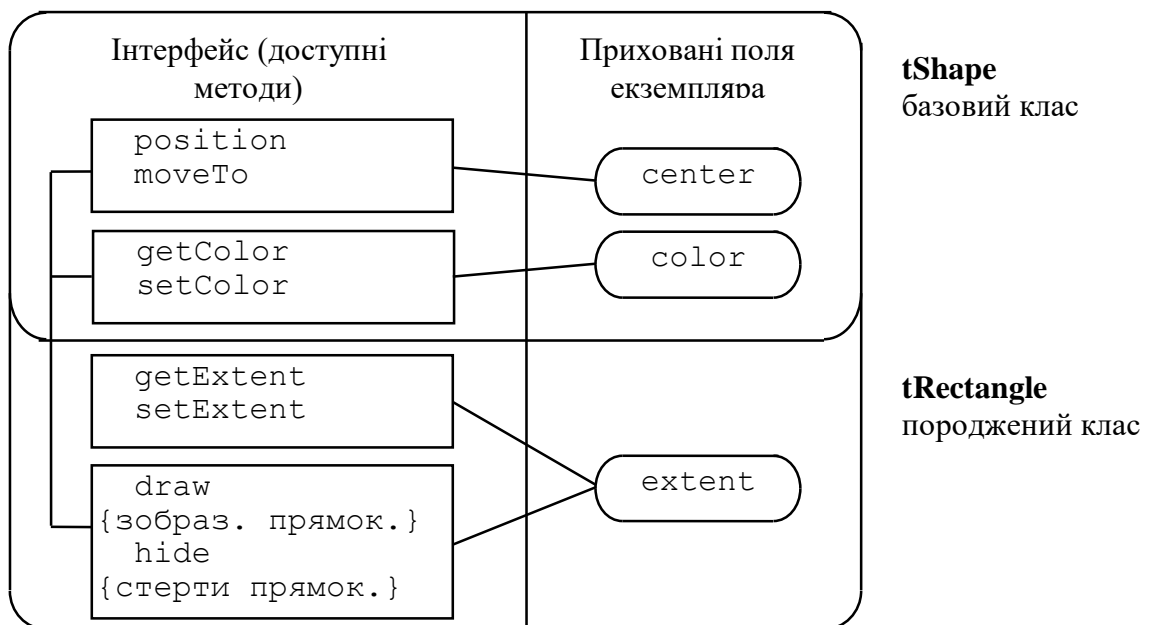


Рис. 6. Клас *tRectangle* (прямокутник) – інший нащадок базового класу

Щоб зобразити прямокутник, потрібно знати його ширину і висоту (не радіус, як для кола). Вони будуть міститися в полі *extent*. Сторони прямокутника вважаються паралельними до осей координат. Потрібні також методи доступу до нового поля і спеціальні методи рисування і стирання прямокутника. Оголошення класу *tRectangle* матиме наступний вигляд (для економії місця наведемо текст реалізації тільки для методу *draw*):

```

type tRectangle = object(tShape)
  procedure getExtent(var e: Point);
  procedure setExtent(e: Point);
  procedure draw;
  procedure hide;
  private
    extent: Point;
  end; {tRectangle}
procedure tRectangle.draw;
  var saveColor: integer; p: Point;
  a,b,c,d: Point;
begin {отримати координати прямокутника}
  position(p);
  {обчислити координати його вершин}
  with extent do
    begin a.x:=p.x-x div 2; a.y:=p.y-x div 2;
          c.x:=a.x+x;   c.y:=a.y+y;
          b.x:=c.x;     b.y:=a.y;
          d.x:=a.x;     d.y:=c.y
    end; {with}
  {зберегти поточний колір рисування екрана}
  saveColor:=screen.foreground;
  {встановити для рисування колір об'єкта}
  screen.setForeground(getColor);
  {зобразити прямокутник методами класу tScreen}
  with screen do
    begin moveTo(a);.lineTo(b);.lineTo(c);
          lineTo(d);lineTo(a)
    end; {with}
  {відновити колір рисування екрана}
  screen.setForeground(saveColor)
end; {draw}

```

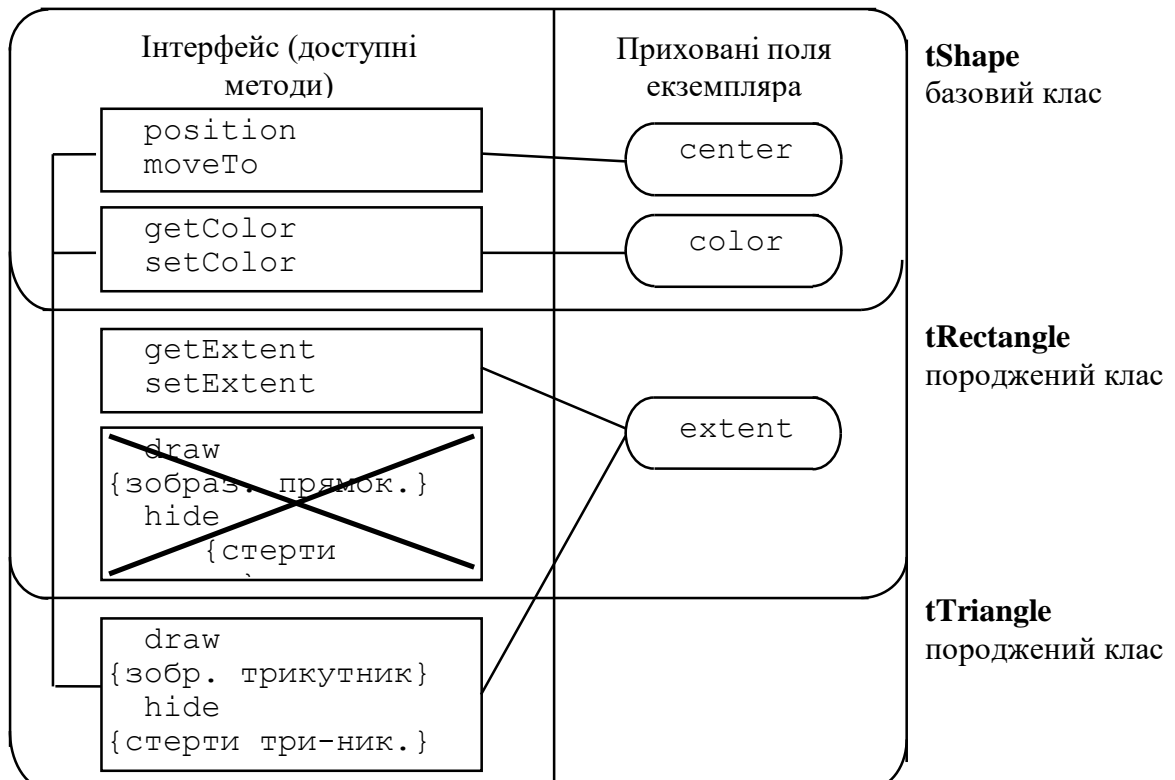


Рис. 7. Клас tTriangle (трикутник) має суттєві спільні властивості з класом tRectangle

Схема на рис. 7 демонструє, що клас *tTriangle* породжено з *tRectangle*. Але чому? Адже прямокутник і трикутник зовні зовсім не схожі. Насправді помилкою було б класифікувати об'єкти лише за їхнім зовнішнім виглядом. Наприклад, еліпс може здатися дуже подібним за формою до кола. Однак щоб справді визначити спільні властивості в ієрархії типів, потрібно проаналізувати, як об'єкти будуть *використовуватись*. Еліпс потребує більше інформації, ніж коло, бо він задається центром і двома півсями (а не одним радіусом). Оскільки трикутник є рівнобедренний, з основою, паралельною до осі абсцис, то його так само, як і прямокутник, можна задати висотою і шириною. Тобто, прямокутник і трикутник мають суттєві спільні властивості, хоч і виглядають по-різному.

Як і для прямокутника, ширина і висота трикутника зберігаються в полі *extent* типу *tPoint*, тому клас *tTriangle* породжено безпосередньо з *tRectangle*. Це означає, що *tTriangle* успадковує всі властивості з обох класів: *tShape* і *tRectangle*. Проте для рисування та стирання трикутника потрібні методи не такі, як для прямокутника. Тому успадковані з *tRectangle* методи *draw* і *hide* в класі *tTriangle* *перевизначаються*.

Реалізація класу *tTriangle* може мати такий вигляд (як і раніше повністю наведемо текст тільки для методу *draw*):

```

type tTriangle = object (tRectangle)
  procedure draw;
  procedure hide;
  end; {tTriangle}
procedure tTriangle.draw;
  var saveColor,w,h,g:integer; a,b,c,p:Point;
begin {отримати координати трикутника}
  position(p);
  {обчислити координати його вершин}
  with extent do begin w:=x div 2; h:=y div 2 end;
  with p do
    begin a.x:=x; a.y:=y+h;
      b.x:=x+w; b.y:=y-h;
      c.x:=x-w; c.y:=y-h
    end; {with}
  {зберегти поточний колір рисування екрана}
  saveColor:=screen.foreground;
  {встановити для рисування колір об'єкта}
  screen.setForeground(getColor);
  {зобразити трикутник методами класу tScreen}
  with screen do
    begin moveTo(a);
      lineTo(b); lineTo(c); lineTo(a)
    end; {with}
  {відновити колір рисування екрана}
  screen.setForeground(saveColor)
end; {draw}

```

4.5. Тестовий приклад

Описані класи можна використовувати для створення об'єктів-фігур та роботи з цими об'єктами. Нижче наведено приклад простої тестової програми.

```

program Tests; uses CRT, ScrClass, Shapes;
  {ScrClass - модуль, що містить оголошення об'єкта,}
  { який реалізує графічний екран. }
  { Shapes - модуль з наведеними вище оголошеннями }
  { класів TShape, tCircle, tRectangle, tTriangle}
procedure testPattern;
begin with screen do {позначити межі екрана}
  begin moveTo(0,0);
    lineTo(0,100); lineTo(100,100);
    lineTo(100,0); lineTo(0,0) end;
  with screen do {позначити центр екрана}
  begin moveTo(0,0); lineTo(100,100);
    moveTo(100,0); lineTo(0,100) end
end; {testPattern}
  {$X+}
procedure testCircle; {тестування об'єкта-кола}
var aCircle:tCircle;
begin with aCircle do
  begin moveTo(50,50); setColor(Red);
    setRadius(4); draw
  end; ReadKey;
  aCircle.hide; ReadKey;
  aCircle.moveTo(60,60); aCircle.draw
end; {testCircle}
procedure testRectangle; {тестування об'єкта-прямокутника}
var aRect:tRectangle;
begin with aRect do
  begin moveTo(50,50); setColor(Red);
    setExtent(8,8); draw
  end; ReadKey;
  aRect.hide; ReadKey;
  aRect.moveTo(60,60); aRect.draw
end; {testRectangle}
procedure testTriangle; {тестування об'єкта-трикутника}
var aTrian:tTriangle;
begin with aTrian do
  begin moveTo(50,50); setColor(Red);
    setExtent(8,8); draw
  end; ReadKey;
  aTrian.hide; ReadKey;
  aTrian.moveTo(60,60); aTrian.draw
end; {testTriangle}
begin {main program}
  screen.Initialize; screen.background(Cyan);
  testPattern; testCircle; ReadKey;
  screen.clear;
  testPattern; testRectangle; ReadKey;
  screen.clear;
  testPattern; testTriangle; ReadKey;
  screen.cleanUp
end.

```

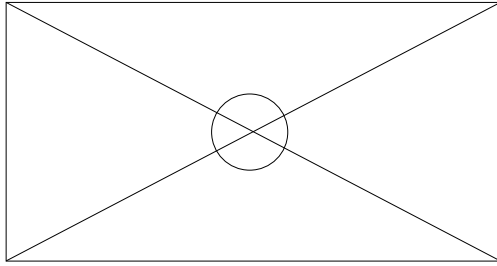


Рис. 8. Приклад використання об'єктів

Після виконання процедур *testPattern* та *testCircle* екран дисплея матиме вигляд, як на рис. 8.

5. Поліморфізм у дії

Здатність поміщати об'єкти однієї ієрархії типів, побудованої на основі спільного базового типу, в одну структуру даних і використовувати для взаємодії з ними єдиний протокол, оголошений для базового класу, відома як поліморфізм. Термін *поліморфізм* у біології означає буквально “багато форм”. Біолог визначає поліморфізм як варіацію форм і функцій, які трапляються в окремих членів однієї популяції.

Поліморфізм спрощує завдання програміста, узагальнюючи синтаксис взаємодії. Таке узагальнення дає змогу трактувати об'єкти різних типів подібним чином. Для вираження спільності протоколу взаємодії та надсилання повідомлень об'єктам подібних (але не точно тих самих) типів поліморфізм використовує успадкування. Поліморфізм – це здатність подібних об'єктів відповідати на одне і те саме повідомлення по-різному. Для цього поліморфні типи містять оголошення методів з однаковими заголовками та різними реалізаціями – відповідними для різних типів ієрархії. Вибір відповідного методу виконується за допомогою *узагальненого механізму диспетчеризації*. Диспетчеризацією називається скерування поліморфного повідомлення до конкретного методу: одного з багатьох однойменних.

5.1. Чим поліморфізм відрізняється від успадкування

Успадкування, по суті, є просто засобом, що дає змогу повторно використовувати код. Поліморфізм застосовується тільки до методів, успадкованих з базового класу. Успадкування є більш загальним засобом, ніж поліморфізм. Його використовують для модифікації поведінки класу, який має властивості близькі до потрібних, але трохи інші. Наприклад, щоб забезпечити підтримку миші, якої нема у класа *tRectangle*, треба просто створити на його базі новий клас, який успадкує всі наявні можливості, і додати поля та методи, потрібні для підтримки миші.

Найбільш базове використання успадкування – розширити або звужити можливості існуючих класів. Поліморфізм використовує успадкування з більш спеціальною метою: щоб виразити спільність. Це означає, що поліморфізм використовує успадкування тільки для побудови поліморфних типових ієрархій. Вони подібні на ієрархії звичайних типів, але мають свою особливість: протокол взаємодії для всіх типів у поліморфній ієрархії задається в базовому класі.

Поліморфізм потребує визначення базового класу, який задає множину операцій чи процедур, що їх використовуватимуть усі підкласи для взаємодії.

5.2. Поліморфізм узагальнює диспетчеризацію

Ближчий розгляд процедур *testCircle*, *testRectangle* і *testTriangle* виявляє, що їх тексти практично однакові. Якщо не брати до уваги різні імена і типи об'єктів, то процедури відрізняються лише одним рядком:

```

procedure testCircle;
... aCircle.setRadius(4) {лише для кола} ...
end;
procedure testRectangle;
... aRect.setExtent(8,8) {лише для прямокутників}...
end;
procedure testTriangle;
... aTrian.setExtent(8,8) {лише для трикутників}...
end;

```

Зауважимо, що схожість між прямокутниками і трикутниками стає більш очевидною при розгляді тексту програми, ніж при розгляді самих фігур на екрані.

Подібність процедур викликає бажання спростити програму, замінивши їх однією (процедурою *testShape*), яка б виконувала такі ж дії зі своїм аргументом – вказівником на об'єкт типу *tShape*. Тоді самі об'єкти створювалися б у головній програмі і передавалися б їй як фактичний аргумент:

```

program Tests; uses CRT, ScrClass, Shapes;
{$X+}
type ShapePtr=^tShape;
procedure testShape(p:ShapePtr);
begin with p^ do
    begin moveTo(50,50); setColor(Red); draw
    end; ReadKey;
    p^.hide; ReadKey;
    p^.moveTo(60,60); p^.draw; ReadKey;
    screen.clear
end; {testShape}
var aShape: ShapePtr;           {вказівник на фігуру}
begin {main program}
    screen.Initialize; screen.background(Cyan);
    aShape:=New(tCircle);       {створили коло           }
    aShape^.setRadius(4);       { задали його радіус}
    testShape(aShape);
    aShape:=New(tRectangle);    {створили прямокутник}
    aShape^.setExtent(8,8);     {задали його розміри }
    testShape(aShape);
    aShape:=New(tTriangle);     {створили трикутник }
    aShape^.setExtent(8,8);     {задали його розміри}
    testShape(aShape);
    screen.cleanUp
end.

```

На жаль, ця програма не працюватиме. Методи *setRadius* і *setExtent* не визначені в базовому класі *tShape*, тому під час спроби їх викликати трапиться помилка етапу компіляції. Щоб уникнути цієї ситуації, треба використати *конструктори* – методи спеціального вигляду, які застосовують для ініціалізації полів новостворених об'єктів. У випадку програмування мовою Pascal ці методи прийнято називати іменем *Init*. Отже, інтерфейси оголошених раніше класів слід доповнити конструкторами:

```

type tShape = object ...
constructor Init(p: Point);
    ... end; {tShape}
    tCircle = object(tShape) ...
constructor Init(p: Point; r: integer);
    ... end; {tCircle}
    tRectangle = object(tShape) ...
constructor Init(p,e: Point); ...
    ... end; {tRectangle}
    tTriangle = object(tRectangle)
constructor Init(p,e: Point); ...
    ... end; {tTriangle}

constructor tShape.Init;
    begin center:=p; color:=Black end;
constructor tCircle.Init;
    begin tShape.Init(p); radius:=r end;
constructor tRectangle.Init;
    begin tShape.Init(p); extent:=e end;
constructor tTriangle.Init;
    begin tShape.Init(p); extent:=e end;

```

Конструктор треба викликати під час створення об'єкта, бо в іншому випадку можливе виникнення помилок при використанні неініціалізованого об'єкта. Синтаксис Pascal'ю дає змогу зазначати ім'я конструктора під час звертання до функції *New*:

```

.....
const c:Point =(x:8;y:8);
        e:Point =(x:50;y:50);
begin {main program}
    screen.Initialize; screen.background(White);
    aShape:=New(tCircle,Init(c,4));
    testShape(aShape);
    aShape:=New(tRectangle,Init(c,e));
    testShape(aShape);
    aShape:=New(tTriangle,Init(c,e));
    testShape(aShape);
    screen.cleanUp
end.

```

Проте, проблема все ще є. Як викликати відповідний метод *draw* у процедурі *testShape*? Нагадаємо, що кожен клас визначає свій власний особливий

метод зображення фігури. Потрібно було б викликати цей метод залежно від типу об'єкта-аргумента процедури *testShape*. Розглянемо попереднє визначення цієї процедури:

```
procedure testShape (p:ShapePtr);
begin ... p^.draw; ... end; {testShape}
```

Змінна p^{\wedge} є об'єктом класу *tShape*, але метод *draw* для нього не визначений. Це невелика трудність: можна просто додати оголошення *draw* до інтерфейсу класу *tShape*. Але як визначити тіло методу? В ідеалі *tShape.draw* мав би нічого не робити (у всякому разі, нічого не рисувати), а особливості рисування конкретних фігур мали б бути визначені в породжених класах:

```
type tShape = object ...
    procedure draw; ... end; {tShape}
procedure tShape.draw;
begin end;
```

Чи буде це вирішенням проблеми? Адже кожен нормальний компілятор встановить, що p є вказівником на *tShape* і зв'яже p^{\wedge} з методом *tShape.draw*, який нічого не робить. Справді, ми не використовуємо для зв'язку з поліморфними об'єктами переваг узагальненого механізму диспетчеризації. У випадку узагальнення сильно спрощується використання поліморфної ієрархії типів у прикладних програмах, але втрачається інформація про особливості методів для об'єктів різних типів. Одним з вирішень могло б бути збереження інформації про тип об'єкта всередині нього самого:

```
type kind=(circle, rectangle, triangle);
    tShape=object ...
    procedure setShapeType (st:kind);
    procedure getShapeType (var st:kind);
    private ... shapeType:kind;
    end; {tShape}
```

Цю інформацію слід було б задавати під час створення об'єкта.

Далі є дві можливості забезпечити спрямування повідомлення *draw* до відповідного методу. Диспетчеризацію можна організувати в процедурі *testShape*, наприклад, так:

```
procedure testShape (p:ShapePtr);
var k:kind;
begin ... {рисування фігури}
    p^.getShapeType (k);
    case k of
        circle: tCircle(p^).draw;
        rectangle: tRectangle(p^).draw;
        triangle: tTriangle(p^).draw;
    end; {case} ...
end;
```

Але ж метою узагальнення протоколу взаємодії було спрощення програми, а з наведеного прикладу цього зовсім не видно!

Іншою альтернативою є організація вибору потрібного методу всередині класу *tShape*. Такий підхід задовільнив би принцип приховування деталей реалізації так глибоко, як тільки можливо:

```

procedure tShape.draw;
begin      case shapeType of
    circle: tCircle.draw;
  rectangle: tRectangle.draw;
    triangle: tTriangle.draw end {case}
end; {draw}

```

Бачимо, що такий варіант програми теж непростий.

Описані два підходи поставили важке питання: **хто має турбуватися про диспетчеризацію повідомлень до відповідних методів** – користувач ієрархії класів *tShape* (це він пише процедуру *testShape*), чи розробник цієї ієрархії (метод *tShape.draw*)? З одного боку, користувач мав би отримати готовий продукт і зайнятися його розширенням, породженням підкласів, створенням екземплярів, їх використанням тощо – наступними, вищими рівнями реалізації прикладної програми. Адже саме такий підхід є характерним для ООП. З іншого боку, розробник не в змозі передбачити всієї різноманітності фігур, у яких може виникнути потреба. Крім того, хто б з них не займався реалізацією вибору потрібних методів за допомогою перемикачів, було б дуже важко виконати навіть невелику модифікацію ієрархії типів. Адже кількість операторів варіанту була б дуже великою навіть для простого протоколу взаємодії.

Об'єктно-орієнтовані мови програмування підтримують спеціальний засіб, який дає змогу легко вирішити описану проблему. Диспетчеризація поліморфних повідомлень реалізується за допомогою *пізнього*, чи *динамічного* зв'язування. Воно виконується, на відміну від *раннього*, не на етапі компіляції програми, а під час її виконання, і використовує інформацію про справжній тип поліморфного об'єкта.

5.3. Динамічне (пізнє) зв'язування

Розширення базового класу для породження нового підкласу не потребує перевизначення і перекомпіляції першого. Всі породжені класи використовують протокол, визначений у базовому класі. Відсилання до відповідних спеціальних методів (методів підкласу) відбувається під час виконання програми. Це ще одне розширення на додаток до ієрархії поліморфних типів, яке не потребує перекомпіляції та перемикачів-диспетчерів. Воно називається *пізнім* або *динамічним зв'язуванням*.

Щоб забезпечити пізнє зв'язування, екземпляри поліморфних класів містять інформацію про свій тип. Ця інформація та процес її використання невидимі для програміста, і він може нічого про це не знати.

Функції, зв'язування з якими виконується динамічно, називаються віртуальними. Компілятор пам'ятає таблицю адрес розташування віртуальних функцій, щоб мати змогу під час виконання програми передати керування відповідній функції. Таку таблицю часто називають *таблицею віртуальних методів* (VTM). Динамічно зв'язувані методи оголошуються в базовому класі з ключовим словом **virtual**. Методи, що перекривають їх у всіх породжених класах, теж повинні бути віртуальними. Всі підкласи деякого базового класу з віртуальними методами мусять мати власну реалізацію цих методів.

Отже, оголошення проектованої ієрархії типів матиме вигляд:

```

type tShape = object
  constructor Init(p: Point);
  function getColor: Colors;
  procedure setColor(c: Colors);
  procedure position(var p: Point);
  procedure moveTo(p: Point);
  procedure draw; virtual;
  procedure hide; virtual;
  private
  center: Point;
  color: Colors; end; {tShape}
  tCircle = object(tShape)
constructor Init(p: Point; r: integer);
  function getRadius: integer;
  procedure setRadius(r: integer);
  procedure draw; virtual;
  procedure hide; virtual;
  private
  radius: integer; end; {tCircle}
  tRectangle = object(tShape)
constructor Init(p,e: Point);
  procedure getExtent(var e: Point);
  procedure setExtent(e: Point);
  procedure draw; virtual;
  procedure hide; virtual;
  private
  extent: Point; end; {tRectangle}
  tTriangle = object(tRectangle)
constructor Init(p,e: Point);
  procedure draw; virtual;
  procedure hide; virtual; end; {tTriangle}

```

Компілятор автоматично вставить у код виконуваної програми таблиці віртуальних методів, і процедура *testShape*, викликатиме щоразу метод *draw* відповідного об'єкта – фактичного аргумента: *tCircle.draw*, *tRectangle.draw*, чи *tTriangle.draw*, не потребуючи ніяких операторів-перемикачів. І розробник, і користувач класу *tShape* тепер можуть турбуватись про інші проблеми: розробник – про нові підкласи ієрархії, а користувач – про вищі рівні реалізації прикладної програми. Перший виклик процедури *testShape* створить на екрані таке ж зображення, як показано на рис. 8 (на стор. 21).

Розглянемо докладніше, як відбувається динамічне зв'язування. Для кожного класу з віртуальними методами компілятор створює в сегменті даних таблицю, яка містить імена та адреси розташування в сегменті коду цих методів (*draw* і *hide* в нашому прикладі), тобто таблицю віртуальних методів. Виконання конструктора *Init*, крім ініціалізації полів об'єкта, пов'язує його з VTM його класу. Таке зв'язування відбувається автоматично у відповідь на ключове слово **constructor**, вказане в заголовку методу *Init*. Тепер виклик $p^{\wedge}.draw$ у процедурі *testShape* виконується в наступній послідовності: p – вказівник на об'єкт- \langle фактичний аргумент процедури \rangle ; p^{\wedge} – власне цей об'єкт (коло, прямокутник чи трикутник); VTM відповідного класу (її адреса записана в невидимому додатковому полі об'єкта); код потрібного методу. Взаємозв'язки між об'єктами, VTM і методами схематично зображені на рис. 9.

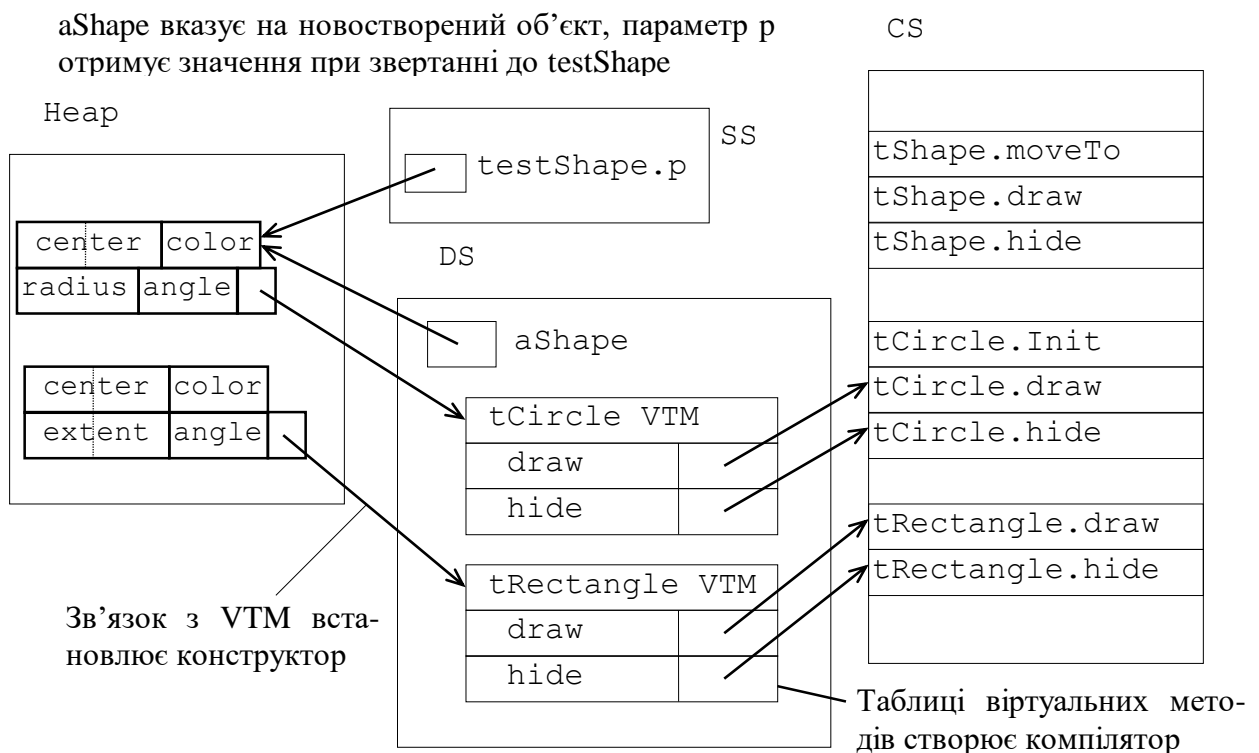


Рис. 9. Виклик методу *draw* у процедурі *testShape* здійснюється за таким ланцюжком:

$p - \wedge - \text{VTM} - \text{draw}$

Із всього сказаного можна зробити такі важливі висновки:

- класи з віртуальними методами обов'язково повинні містити один спеціальний метод – конструктор;
- виклик конструктора необхідно виконати перед першим звертанням до об'єкта, найкраще – під час створення об'єкта; виклик віртуального методу об'єкта, неініціалізованого конструктором, незмінно призводить до зависання комп'ютера;
- конструктор не може бути віртуальним.

Тепер можна спростити написану раніше програму. Розглянемо уважніше реалізацію методів *draw* у різних класах. Легко бачити, що значна частина коду повторюється. Справді, ці методи відрізняються лише оператором **with** screen

do Усі методи на вході зберігають поточний колір рисування і відновлюють його перед виходом. Методи задають також колір рисування фігури. У всіх нових породжених підкласах нам доведеться програмувати такі самі дії, що буде зайвою роботою. Було б добре, якби підкласи містили тільки код, специфічний для конкретної фігури. Цього легко досягти, оголосивши метод *draw* базового класу наступним чином:

```

procedure tShape.draw;
  var saveColor: integer;
  begin saveColor:=screen.foreground;
        screen.setForeground(color);
        drawShape; {рисування фігури}
        screen.setForeground(saveColor)
  end; {draw}

```

де метод *drawShape* оголошений в класі *tShape* як віртуальний, з порожнім тілом. У всіх породжених класах він повинен перевизначатись, реалізуючи рисування відповідного об'єкта. Оскільки *drawShape*, фактично, є методом “внутрішнього використання”, то його доцільно оголошувати в прихованій частині об'єкта. Текст програми з описаними змінами буде наведено нижче.

5.4. Обертання поліморфних об'єктів-фігур

Нашою головною метою було обернути об'єкти навколо початку системи координат. Наведена нижче програма виконує це завдання, але спочатку вкажемо на деякі її особливості. По-перше, в оголошенні класу *tShape* додатково вказані поле і методи для підтримки інформації про поточний кут повороту даного об'єкта. Усі класи використовують процедуру *rotateAboutOrigin* для обчислення нових координат даної точки після повороту на даний кут. (Процедура оголошена в частині реалізації модуля *Shapes*) Методи *drawShape* всіх породжених класів модифіковані так, щоб враховувати кут повороту фігури. Вся ієрархія типів *tShape* розташована в окремому модулі – *Shapes*.

```

unit Shapes;      interface uses ScrClass;
type ShapePtr = ^tShape;
      tShape = object
  constructor Init(p: tPoint);           { |           }
    function getColor: Colors;           { |           }
    procedure setColor(c: Colors);       { |           }
    function getAngle: integer;          { | протокол }
    procedure setAngle(a: integer);      { | взаємодії }
    procedure position(var p: tPoint);  { | класу     }
    procedure moveTo(p: tPoint);        { |           }
    procedure draw;                       { |           }
    procedure hide;                       { |           }
    procedure rotate(a: integer);        { |           }
      private
  center: Point; color: Colors; angle: integer;
  procedure drawShape; virtual;

```

```

        end; {tShape}
{-----}
CirclePtr = ^tCircle;
    tCircle = object(tShape)
    constructor Init(p: tPoint; r: integer);
    destructor Done; virtual;
        private
radius: integer;
    procedure drawShape; virtual;
        end; {tCircle}
{-----}
    RectPtr = ^tRectangle;
    tRectangle = object(tShape)
    constructor Init(c,e: tPoint);
    destructor Done; virtual;
        private
extent: tPoint;
    procedure drawShape; virtual;
        end; {tRectangle}
{-----}
    TriPtr = ^tTriangle;
    tTriangle = object(tRectangle)
    constructor Init(c,e: tPoint);
    destructor Done; virtual;
        private
    procedure drawShape; virtual;
        end; {tTriangle}
{=====} implementation
procedure rotateAboutOrigin(p:tPoint;a:integer;var newP:Point);
var theta,cos_theta,sin_theta:real;
begin theta:=a*PI/180;
    cos_theta:=cos(theta); sin_theta:=sin(theta);
    with p do
    begin newP.x:=round(x*cos_theta-y*sin_theta);
        newP.y:=round(x*sin_theta+y*cos_theta)
    end
end;
{=====}
constructor tShape.Init(p: tPoint);
    begin center:=p; angle:=0;
        color:=screen.foreground end;
function tShape.getColor: Colors;
    begin getColor:=color end;
procedure tShape.setColor(c: Colors);
    begin color:=c end;
function tShape.getAngle: integer;
    begin getAngle:=angle end;
procedure tShape.setAngle(a: integer);
    begin angle:=a end;
procedure tShape.Position(var p: tPoint);
    begin p:=center end;
procedure tShape.moveTo(p: tPoint);
    begin center:=p end;
procedure tShape.draw;

```

```

    var saveColor: Colors;
    begin with screen do
        begin saveColor:=foreground; setForeground(color);
            drawShape; setForeground(saveColor) end
    end;
procedure tShape.hide;
    var saveColor: Colors;
    begin with screen do
        begin saveColor:=foreground; setForeground(background);
            drawShape; setForeground(saveColor) end
    end;
procedure tShape.Rotate(a: integer);
    begin setAngle(angle+a); draw end;
procedure tShape.drawShape;
    begin end;
{-----}
constructor tCircle.Init(p: tPoint; r: integer);
    begin tShape.Init(p); radius:=r end;
destructor tCircle.Done;
    begin end;
procedure tCircle.drawShape;
    var ctr: tPoint;
    begin position(ctr); rotateAboutOrigin(ctr,getAngle,ctr);
        screen.tCircle(ctr,radius)
    end;
{-----}
constructor tRectangle.Init(c,e: tPoint);
    begin tShape.Init(c); extent:=e end;
destructor tRectangle.Done;
    begin end;
procedure tRectangle.drawShape;
    var a,b,c,d,p:tPoint; g:integer;
    begin g:=getAngle; position(p);
        with extent do
            begin a.x:=p.x-x div 2; a.y:=p.y-x div 2;
                c.x:=a.x+x;    c.y:=a.y+y;
                b.x:=c.x;    b.y:=a.y;
                d.x:=a.x;    d.y:=c.y end;
            rotateAboutOrigin(a,g,a); rotateAboutOrigin(b,g,b);
            rotateAboutOrigin(c,g,c); rotateAboutOrigin(d,g,d);
            with screen do
                begin moveTo(a);.lineTo(b);.lineTo(c);
                    lineTo(d); lineTo(a) end
            end;
        end;
{-----}
constructor tTriangle.Init(c,e: tPoint);
    begin tRectangle.Init(c,e) end;
destructor tTriangle.Done;
    begin end;
procedure tTriangle.drawShape;
    var w,h,g:integer; a,b,c,p:tPoint;
    begin with extent do
        begin w:=x div 2; h:=y div 2 end;
        g:=getAngle; Position(p);

```

```

with p do
begin a.x:=x;   a.y:=y+h;
      b.x:=x+w; b.y:=y-h;
      c.x:=x-w; c.y:=y-h end;
rotateAboutOrigin(a,g,a); rotateAboutOrigin(b,g,b);
rotateAboutOrigin(c,g,c);
with screen do begin moveTo(a);
                    lineTo(b); lineTo(c); lineTo(a) end
end;
end. {Shapes}

```

Щоб продемонструвати поліморфізм у дії, використано поліморфний масив, який містить зображувані об'єкти. Наступні оператори оголошують масив і заносять у нього три нові об'єкти:

```

const cpos:Point=(x:60;y:6); rpos:Point=(x:70;y:6);
      tpos:Point=(x:80;y:6); ext:Point=(x:8;y:8);
type ShapePtrMas=array[1..10]of ShapePtr;
var shapeArray:ShapePtrMas;
begin shapeArray[1]:=New(CirclePtr,Init(cpos,4));
      shapeArray[2]:=New(RectPtr,Init(rpos,ext));
      shapeArray[3]:=New(TriPtr,Init(tpos,ext)); ...

```

Масив *shapeArray* є справжнім поліморфним масивом. У нього можна заносити будь-які елементи, якщо тільки вони є вказівниками на об'єкти ієрархії типів *tShape*. Тепер, щоб обернути кожен елемент масиву, достатньо виконати наступний оператор циклу:

```

for j:=1 to size do {обертає size елементів масиву}
begin shapeArray[j]^draw; {початкове розташування}
      for i:=1 to 8 do {обернути фігуру 8 раз}
        shapeArray[j]^rotate(10) {по 10 градусів}
      end; {відрахованих від поточного значення кута}

```

Кожна фігура зображається у своєму початковому положенні методом *tShape.draw*, а тоді вісім раз повертається за допомогою методу *tShape.rotate*. Він щоразу збільшує значення поля *angle* об'єкта на десять градусів і викликає метод *draw*, щоб зобразити об'єкт у новій позиції. Зауважимо, що методи *draw* і *rotate* є статичними і одними і тими ж для всієї ієрархії. Зв'язування з ними виконується під час компіляції програми. Методи ж *drawShape* (які викликаються з *draw*) є віртуальними, специфічними для кожного з класів ієрархії. Зв'язок з *drawShape* налагоджується під час виконання програми за допомогою VTM того об'єкта, на який вказує *shapeArray[j]*. У цьому і полягає узагальнений механізм диспетчеризації (відсилання) повідомлень до відповідних методів.

Із самого початку ми поставили за мету перенесення DOS-програми в інше операційне середовище. Щоб зробити цей процес набагато менш болісним, доцільно розділити програму на частини, які різною мірою потребуватимуть змін. Створення об'єктів-фігур та їх обертання, очевидно, є найбільш стабільною

частиною програми, незалежною від конкретної платформи, тому ці дії відокремлені в самостійний модуль.

```

unit Rotation; interface
  procedure RotateShapes;
implementation uses Shapes, ScrClass;
  const cpos:Point=(x:60;y:6); rpos:Point=(x:70;y:6);
    tpos:Point=(x:80;y:6); ext:Point=(x:8;y:8);
    nShapes=10;
  type range=1..nShapes;
    ShapePtrMas=array[range]of ShapePtr;
procedure RotateArray
  (var a:ShapePtrMas; size:range; count:byte);
  var i:byte; j:range;
  begin for j:=1 to size do
    with a[j]^ do
      begin draw;
        for i:=1 to count do rotate(10)
      end
    end; {RotateArray}
procedure RotateShapes;
  var shapeArray:ShapePtrMas;
  begin shapeArray[1]:=New(CirclePtr, Init(cpos, 4));
    shapeArray[1]^.setColor(Red);
    shapeArray[2]:=New(RectPtr, Init(rpos, ext));
    shapeArray[2]^.setColor(Blue);
    shapeArray[3]:=New(TriPtr, Init(tpos, ext));
    shapeArray[3]^.setColor(LightMagenta);
    RotateArray(shapeArray, 3, 9);
  end; {RotateShapes}
end. {Rotation}

```

6. Розробка і реалізація класу tScreen

Інкапсуляція вберігає користувачів класів від можливих змін. Можна сказати, що між розробниками і користувачами діє така угода: користувач взаємодіє з об'єктами лише через інтерфейс, розробник гарантує певну поведінку об'єктів за цих умов. Дотримання специфікацій на ранніх стадіях розробки задає розподіл інтересів. Як тільки досягнуто угоди, розробники і користувачі можуть займатися власною частиною проекту: доки вони дотримуватимуться угоди щодо інтерфейсу класу, доти можуть не цікавитись роботою один одного. Написавши “заглушки” замість ще не готових методів, користувач може навіть компілювати свою програму: компілятор перевірить правильність дотримання протоколу. Розробник має свободу для експериментів з метою розробки найкращої за всіма параметрами реалізації класу.

Побачимо, як такий поділ праці діє на практиці, проектуючи згаданий раніше клас *tScreen*. Нашою метою буде запустити програму обертання фігур з

мінімальними змінами в різних графічних середовищах: Borland's BGI (Borland Graphics Interface) та Microsoft Windows.

Обговорений раніше клас *tScreen* підтримує небагато методів маніпулювання з графічним екраном. Завданням розробника є так реалізувати систему і визначити інтерфейс, щоб мінімізувати зміни в кодї замовника у випадку перенесення програми на іншу платформу. Завдання замовника – точно сформулювати вимоги до функціональних можливостей, необхідних для підтримки його програми.

Екран використовує прямокутну систему координат з початком у лівому нижньому кутку. Незалежно від комп'ютера і операційної системи, максимальні значення абсциси й ординати рівні 100. Екран використовує уявну ручку для рисування ліній. Її можна переміщувати, рисувати пряму, коло, змінювати колір фону і рисунку, очищати екран. Усі графічні системи потребують задання початкових параметрів, яке виконуватиметься методом *initialize*. На завершення багато систем потребують відновлювальних операцій, які виконуватиме метод *cleanUp*. У доступній частині класу треба оголосити також кольори. Вони будуть доступні і для розробника, і для користувача.

Після обговорення вимог до інтерфейсу розробник може зайнятися реалізацією класу. В приватній частині потрібно оголосити внутрішні змінні, необхідні для перетворення віртуальної системи координат екрана *tScreen* в реальні координати системи, яку підтримує конкретна графічна бібліотека.

Читачеві, знайомому з основними прийомами використання модуля *Graph.tpu* нескладно буде освоїти реалізацію методів, наведену нижче:

```

unit ScrClass;           interface uses Graph;
const {максимальні значення віртуальних координат}
    virtual_maxX=100; virtual_maxY=100;
    {константи кольорів оголошено для того, щоб не було
    потреби підключати до головної програми модуль Graph}
    Black=0;      Blue=1;      Green=2;      Cyan=3;
    Red=4;        Magenta=5;    Brown=6;    LightGray=7;
    DarkGray=8;   LightBlue=9; LightGreen=10; LightCyan=11;
    LightRed=12;  LightMagenta=13; Yellow=14;   White=15;
type Colors = Black..White;
    tPoint = record x,y:integer end;
    {клас tScreen реалізує графічний екран}
    tScreen = object
procedure initialize; {переводить екран у графічний режим}
procedure cleanUp;    {завершує використання графіки}
procedure moveTo(p:tPoint); {переміщує ручку в точку p}
procedure lineTo(p:tPoint); {рисує лінію до точки p}
procedure circle(p:tPoint; r:integer); {рисує коло}
procedure setBackground(c^Colors); {задає колір фону}
    function background:byte; {повертає поточний колір фону}
procedure setForeground(c:Colors); {задає колір рисунка}
    function foreground:byte; {повертає поточн. колір рисув.}
procedure clear;      {очищає графічний екран}
    private
    maxX, maxY: word; {роздільна здатність конкретного екрана}

```

```

procedure translate(p:tPoint; var newP:tPoint);
  function translateX(x:integer):integer;      {перетворення}
  function translateY(y:integer):integer;      {віртуальних}
end; {tScreen}                               {координат у реальні}
{програма використовуватиме один екземпляр класу tScreen}
var screen: tScreen;
      implementation
procedure tScreen.translate(p:tPoint; var newP:tPoint);
  var x,y^integer;
  begin with p do begin newP.x:=translateX(x);
                        newP.y:=translateY(y) end

  end;
function tScreen.translateX(x:integer):integer;
  begin ttranslateX:=round(maxX*x/virtual_maxX) end;
function tScreen.translateY(y:integer):integer;
  begin translateY:=maxY-round(maxY*y/virtual_maxY) end;

```

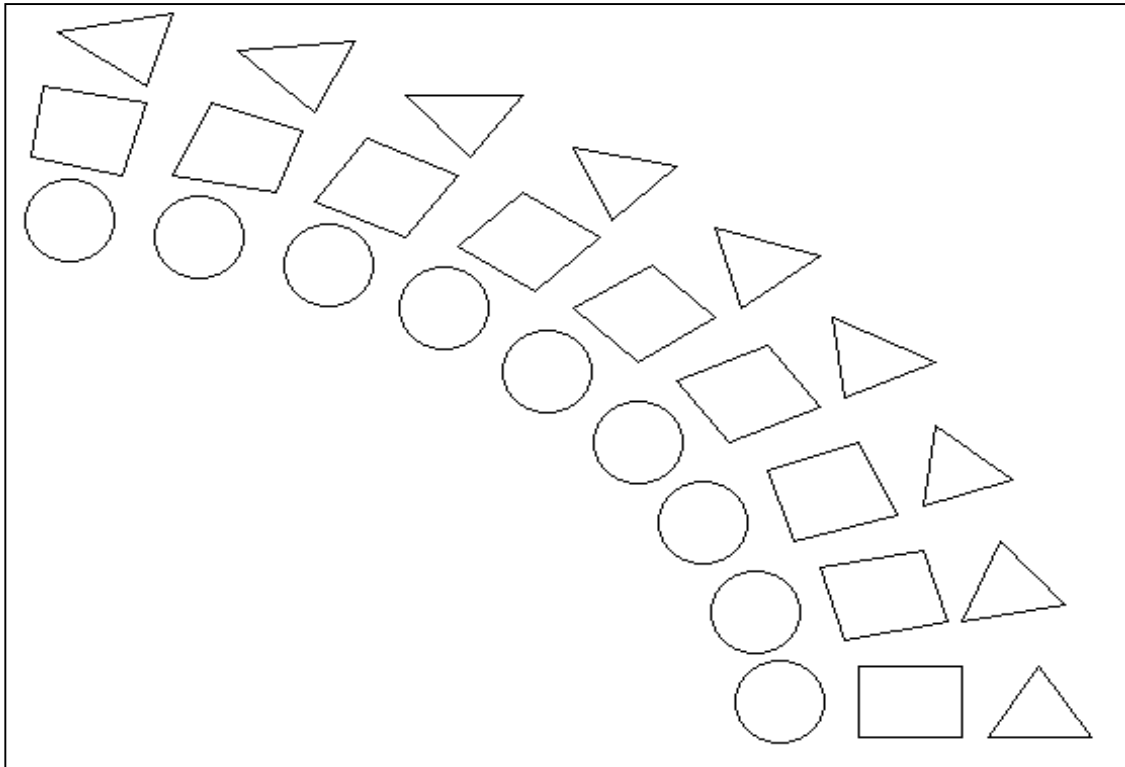


Рис. 10. Поліморфізм у дії: різні об'єкти отримують однакові повідомлення

```

procedure tScreen.initialize;
  var grDriver, qrMode, ErrCode: Integer;
  begin grDriver:=Detect; InitGraph(grDriver,qrMode,'');
        ErrCode:=GraphResult;
        if ErrCode<>grOk then begin
          writeln('Graphics error:', GraphErrorMsg(ErrCode));
          Halt(1) end; {if}
        maxX:=Graph.GetMaxX; maxY:=Graph.GetMaxY;
        setForeground(White); setBackground(Blue)
  end;

```

```

procedure tScreen.cleanUp;
  begin Graph.CloseGraph end;
procedure tScreen.moveTo(p:tPoint);
  begin translate(p,p);
    with p do Graph.moveTo(x,y) end;
procedure tScreen.lineTo(p:tPoint);
  begin translate(p,p);
    with p do Graph.lineTo(x,y) end;
procedure tScreen.circle(p:tPoint; r:integer);
  var xr,yr:integer;
  begin translate(p,p):
    xr:=round(maxX*r/virtual_maxX);
    yr:=round(maxY*r/virtual_maxY);
    r:=(xr+yr) div 2;
    with p do Graph.circle(x,y,r) end;
procedure tScreen.setBackground(c:Colors);
  begin Graph.setBkColor(c) end;
function tScreen.background:byte;
  begin background:=Graph.getBkColor end;
procedure tScreen.setForeground(c:Colors);
  begin Graph.setColor(c) end;
function tScreen.foreground:byte;
  begin foreground:=Graph.getColor end;
procedure tScreen.clear;
  begin Graph.clearDevice end;
end.

```

Тепер головну програму можна написати зовсім просто:

```

program Main;
uses Shapes,ScrClass,Rotation;
begin screen.Initialize;
  screen.setBackground(White);
  RotateShapes; readln;
  screen.cleanUp
end.

```

Після виклику *RotateShapes* екран матиме вигляд, показаний на рис. 10.

Отже, перша половина поставленої задачі повністю розв'язана: спроектовано поліморфну ієрархію типів *tShape* (повний текст модуля *Shapes* на ст. 28–30); розроблено клас *tScreen*, який реалізує графічний екран у середовищі BGI (ст. 33–35); модуль *Rotation* містить процедури створення й обертання об'єктів-фігур (ст. 31–32). Тепер перенесемо програму в середовище Microsoft Windows.

7. Програмування для Microsoft Windows

Програма обертання фігур для Microsoft Windows буде складнішою, ніж BGI версія. Однак не тому, що клас *tScreen* складніше реалізувати для ОС

Windows, ніж для MS DOS. Основна трудність полягає в тому, щоб організувати загальну структуру Windows-програми. Перш ніж розглядати відмінності у реалізації класу *tScreen*, добре було б з'ясувати, які частини нашої програми зазнають змін. Отже, потрібна нова реалізація модуля *ScrClass*. Крім того, зовсім по-іншому виглядатиме головна програма. А модулі *Rotation* і *Shapes* можуть залишатися такими ж, як описано вище.

Зміни головної програми зумовлені тим, що Microsoft Windows справді є зовсім іншою операційною системою, ніж MS DOS. Програміст повинен передбачити початкову ініціалізацію Windows-програми, все виведення керувати до функцій Windows API (Application Program Interface). Щоб зробити це, потрібно розуміти основи функціонування операційного середовища Windows.

7.1. Графічний інтерфейс користувача

ОС Windows підтримує специфікацію Common User Access (CUA) стандарту System Application Architecture (SAA) фірми IBM. Його використання спрощує створення інтерфейсів прикладних програм завдяки наявності визначеного набору доступних розробникові функцій. Крім того, використання CUA спрощує роботу користувача з програмами за рахунок подібності їхніх інтерфейсів.

Основою користувацького інтерфейсу Microsoft Windows є *вікно*. Вікно – це прямокутна ділянка екрана, що виділяється прикладній програмі для виконання операцій уведення/виведення. Більшість інших елементів інтерфейсу є частковими випадками вікна. Вікно може містити низку компонент: заголовок, рамку, рядкове меню, кнопки та ін.

7.2. Керування подіями

Архітектура Windows реалізована за принципом керування потоком подій. У відповідь на будь-яку подію (натискання на клавішу, переміщення чи фіксація миші, зміна значення таймера) ядро Windows надсилає прикладній програмі *повідомлення* та його розгорнутий опис. Наприклад, у випадку натискання клавіші надсилається повідомлення *wm_KeyDown*, а у випадку вибору команди з рядового меню – *wm_Command*. Повідомлення складається з трьох частин: цілочислового ідентифікатора і двох параметрів, відповідно, типу *word* та *longint*. Ідентифікатор дає змогу однозначно зрозуміти призначення повідомлення, а параметри (які називаються *wParam* і *lParam*) містять інформацію, що розкриває суть події. Усі події поміщаються в чергу подій, яку в міру потреби обробляє прикладна програма.

7.3. Апаратно незалежна графіка

Середовище Windows надає програмістові широкий набір графічних операцій. Можна відображати текст (використовуючи різноманітні шрифти),

геометричні фігури, растрові графічні зображення. Для виведення інформації Windows-програма використовує функції GDI — інтерфейсу графічного пристрою. Ці функції взаємодіють з драйверами пристроїв і є апаратно-незалежними, тобто можуть використовуватись для виведення на будь-який пристрій: екран, принтер чи плотер. Зображення у всіх випадках буде однаковим.

Для виведення графічних зображень у середовищі є низка визначених засобів, зокрема пензлі, олівці, шрифти. Їх використовують відповідно для заповнення фону, побудови графічних примітивів, виведення тексту. Одночасно можна застосовувати лише один олівець, один пензель і один тип шрифту.

7.4. Багатозадачність

Windows дає змогу виконувати кілька задач одночасно. У цьому випадку, як звичайно, одна з них є активною (з нею взаємодіє користувач), а інші або просто залишаються в оперативній пам'яті, або виконуються у фоновому режимі (наприклад, друкування документа, антивірусні програми тощо). Програми, що працюють одночасно, повинні розділяти доступ до ресурсів комп'ютера: центрального процесора, пам'яті, дисплея. На екрані кожній програмі виділяється спеціальна прямокутна область – *вікно*.

7.5. Гнучке керування пам'яттю

Можливість одночасної роботи кількох програм потребує гнучкої схеми керування пам'яттю. У цьому випадку використовують динамічний розподіл пам'яті. Замість вказівників у Windows застосовують *посилання*, які фактично є вказівниками на вказівники. Посилання – це індекс у таблиці вказівників, яку підтримує і використовує ядро Windows.

Однією з переваг керування пам'яттю в ОС Windows є змога використовувати один і той самий код різними програмами. Також можливим є динамічне завантаження бібліотек, функції яких тоді можуть використати одразу кілька програм. Такі *бібліотеки* називаються *динамічно завантажуваними* (DLL).

7.6. Структура Windows-програми

Microsoft Windows дає змогу запустити одночасно кілька екземплярів деякої програми, тому ця ОС розрізняє *аплікацію* (Windows-програму) і *екземпляр* програми. Незалежно від того, скільки екземплярів програми запущено, працює тільки одна аплікація програми.

Наприклад, працюючи з редактором текстів Microsoft Word, ви можете використовувати два екземпляри програми: в одному з них готуєте статтю, а в іншому – електронний лист. Ці екземпляри будуть потребувати різних областей пам'яті для зберігання текстів, але можуть використовувати один і той самий код (код самого власне редактора текстів), що розташований в одному місці пам'яті.

Windows-програма. Спільний код і дані, які використовують різні екземпляри програми, називають *аплікацією*. Є певна група даних, яку треба ініціалізувати під час першого запуску аплікації і не потрібно переініціалізувати у всіх наступних. Як звичайно, це виконує функція “InitApplication”.

Windows-екземпляр. Кожен екземпляр програми оперує з певними внутрішніми даними (наприклад, ділянка оперативної пам’яті, виділена редакторові текстів). У них, крім всього іншого, потрібно зберігати інформацію про вікно, виділене для цього екземпляра: вигляд вікна, його розмір, розташування та ін. Такі дані треба ініціалізувати під час запуску кожного екземпляра. Ініціалізацію виконує функція, яку, як звичайно, називають “InitInstance”.

Процедура WinMain. Аналогом головної частини DOS-програми у середовищі Windows є процедура WinMain. Вона першою отримує керування у Windows-аплікації, перевіряє, чи виконуються в конкретний момент ще якісь екземпляри аплікації. Якщо ні, то вона викликає функцію InitApplication. Далі процедура ініціалізує новий екземпляр, викликаючи InitInstance, і переходить у *цикл обробки повідомлень*. Нагадаємо, що Windows є системою, *керованою подіями*. Кожна подія поміщає повідомлення в чергу. Головний (системний) цикл повідомлень вибирає їх звідти і виконує *диспетчеризацію* – відсилає їх до відповідної аплікації. Псевдокод цього циклу може виглядати так:

```
while GetMessage
    DispatchMessage
```

Кожна аплікація містить подібний цикл. Вона працює доти, доки не отримає повідомлення *wm_Quit*, що, звичайно, стається, коли користувач вибирає команду *Close* в системному меню вікна.

7.7. Обробка повідомлень

Для обробки повідомлень, які отримує аплікація, програміст повинен написати спеціальну функцію. Її можна назвати довільно, але типово її називають *MainWndProc* – віконна процедура. Віконна процедура (функція) відповідає за відбір тих повідомлень, на які реагуватиме ця аплікація. Аплікація сама ніколи не викликає своєї віконної функції: це робить ядро Windows. Такі функції називають *функціями опосередкованого виклику* (callback function). Віконну функцію аплікації вказують в InitApplication. Кожного разу, коли надходять повідомлення до Windows-програми, її віконна функція перевіряє, чи повідомлення є специфічним для цієї аплікації. Тоді вона обробляє його спеціальними засобами аплікації. В іншому випадку *MainWndProc* викликає уповноваженого обробника повідомлень і передає йому отримане повідомлення. Таким обробником є стандартна функція *DefWindowProc*.

Усі функції опосередкованого виклику відповідають на повідомлення *wm_Destroy*. Це повідомлення надсилається аплікації, коли користувач вибирає

“Close” в системному меню. Більшість callback функцій обробляють його, відправляючи quit-повідомлення менеджерів вікон.

Насправді в середовищі Windows є кілька черг, які працюють у режимі реального часу. *Системна черга* містить усі вхідні події всієї системи. Менеджер вікон перевіряє системну чергу, визначає, яка аплікація повинна обробляти чергову подію, і поміщає повідомлення про неї у чергу визначеної аплікації. Кожен екземпляр має свою власну чергу подій. Його віконна функція обробляє ці події. Отримавши повідомлення `wm_Destroy`, вона виконує оператор `PostQuitMessage(0)`, який надсилає повідомлення `wm_Quit` у чергу аплікації. Отримання цього повідомлення завершує роботу циклу обробки повідомлень процедури `WinMain`. Екземпляр припиняє існування, вікно закривається.

Отже, кожна аплікація повинна містити:

- *WinMain* — точка входу в програму і цикл обробки повідомлень;
- *InitApplication* – ініціалізує дані, які використовуватимуть усі екземпляри, зокрема, реєструє в операційній системі клас вікон аплікації;
- *InitInstance* – створює новий екземпляр та ініціалізує його дані;
- *MainWndProc* – обробляє повідомлення аплікації.

Така структура програми є сталою, і програмісти здебільшого використовують шаблони чи заготовки для названих процедур. Найбільше відмінностей між аплікаціями є у їхніх віконних функціях, тому можна сказати, що все програмування Windows-аплікації зосереджене у віконній функції. Програміст повинен визначити повідомлення, на які аплікація відповідатиме своїм власним чином і, можливо, написати процедури, які викликатиме `MainWndProc`.

Будь-яка віконна функція повинна обробляти щонайменше одне повідомлення – `wm_Destroy`. Такий фрагмент демонструє оголошення цієї функції:

```
function MainWndProc (Wnd:hWnd; message,wParam:word;
    lParam: longint)^longint; export;
begin MainWndProc:=0; case message of
    wm_Destroy: begin PostQuitMessage(0); Exit
                end end; {case}
    MainWndProc:=DefWindowProc (message,wParam,lParam)
end; {MainWndProc}
```

Директива **export** спонукає компілятора згенерувати для функції опосередкованого виклику `MainWndProc` спеціальний *пролог*. Пролог пов’язує код функції із сегментом даних екземпляра програми. Завдяки такому зв’язку функція має доступ, зокрема, до змінних і констант програми.

7.8. Використання ресурсів

Ресурси в контексті Windows — це описи елементів інтерфейсу Windows-програм: меню, панелей діалогу, курсорів, іконок, растрових зображень, рядків повідомлень і командних клавіш. Ресурси створюють за допомогою спеціальних

редакторів і потім приєднують до виконуваної програми. Для зменшення обсягу пам'яті, який займає програма під час виконання, ресурси завантажуються в пам'ять лише за необхідністю. Розділення прикладної програми на код та інтерфейсну частину дає змогу змінювати останню, не переробляючи код.

Наша програма, яку наведемо нижче, використовує такі ресурси: рядкове меню "RotateMenu", панель діалогу "AboutBox", курсор `idc_Arrow` та іконку `idc_Application`. Такі курсор та іконка є стандартними (визначеними в ОС Windows) і не потребують ніяких додаткових оголошень, а меню і панель визначені за допомогою спеціальної мови опису ресурсів так:

```

ROTATEMENU MENU
BEGIN
    POPUP "&Help"
    BEGIN
        MENUITEM "&AboutRotate...", 101
    END
END
ABOUTBOX DIALOG 22, 17, 104, 75
STYLE DS_MODALFRAME | VS_OVERLAPPED | VS_CAPTION
    | VS_SYSMENU
CAPTION "About Rotate"
BEGIN
CONTROL "Rotate Shapes", -1, "STATIC", SS_CENTER
    | WS_CHILD | WS_VISIBLE | WS_GROUP, 0, 5, 104, 8
CONTROL "Abstract Screen Class", -1, "STATIC", SS_CENTER
    | WS_CHILD | WS_VISIBLE | WS_GROUP, 0, 20, 104, 8
CONTROL "adapted for", -1, "STATIC", SS_CENTER
    | WS_CHILD | WS_VISIBLE | WS_GROUP, 7 0, 29, 104, 8
CONTROL "Microsoft Windows", -1, "STATIC", SS_CENTER
    | WS_CHILD | WS_VISIBLE | WS_GROUP, 0, 38, 104, 8
CONTROL "OK", 1, "BUTTON", BS_DEFPUSHBUTTON | WS_CHILD
    | WS_VISIBLE | WS_TABSTOP, 35, 55, 32, 14
END

```

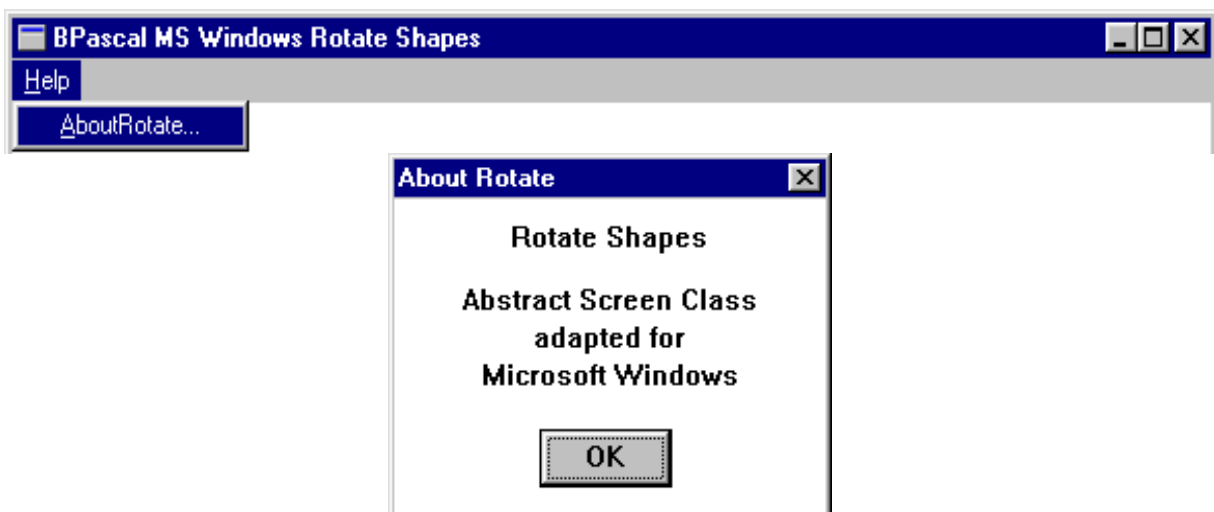


Рис. 11. Використання ресурсів у Windows-аплікаціях

У середовищі Borland Pascal for Windows ресурси створюються за допомогою спеціальної програми – редактора ресурсів Resource Workshop. Визначення ресурсів зберігаються в окремому файлі і приєднуються до програми директивою {\$R <ім'я файла>.res}. Зауважимо, що файли ресурсів стандартно мають розширення “res”.

Меню RotateMenu є зовсім простим: воно складається лише з одного розділу (“Help”), який містить лише одну команду (“About Rotate...”). Знак “&” перед літерою означає, що вона буде виділена. Кожна з команд меню повинна мати власний числовий ідентифікатор. У випадку вибору команди меню ядро Windows надсилає повідомлення *wm_Command* відповідній віконній функції. Параметр *wParam* цього повідомлення містить ідентифікатор команди меню. Віконна функція повинна виконувати всі дії, пов'язані з командою меню. У нашому випадку після вибору користувачем команди “About Rotate...” буде виводитись на екран панель діалогу *AboutBox* і викликатись функція *About*, яка керуватиме цим діалогом. Описані меню і панель діалогу показані на рис. 11.

7.9. Посилання на вікно, екземпляр програми, контекст дисплея

У розумінні Windows *посилання* – це 16-розрядне число, яке однозначно ідентифікує деякий об'єкт середовища. Це число є індексом у таблиці вказівників на певні об'єкти (вікна, олівці, пензлі тощо), яку підтримує та використовує ядро Windows.

Найбільш важливими і часто вживаними є посилання на вікно, екземпляр програми і контекст пристрою (дисплея). Зокрема, щоб викликати панель діалогу, програміст повинен передати як параметри процедурі створення панелі посилання на вікно і посилання на екземпляр.

Посилання на вікно дає змогу однозначно визначати кожне вікно, що використовується в системі. Більшість функцій Microsoft Windows отримують посилання на вікно як перший параметр. Маючи посилання на вікно, ви можете переміщувати його, змінювати його розмір, перерисовувати вміст вікна тощо. *Посилання на екземпляр* дає програмістові доступ до даних конкретного екземпляра програми, що працює. Після створення нового екземпляра програми посилання на нього присвоюється глобальній змінній ОС Windows.

Перед виведенням тексту чи рисуванням графіки всередині вікна потрібно задати його *дисплейний контекст*. Дисплейний контекст автоматично переводить відносні віконні координати в абсолютні дисплейні. Він зберігає також дані про поточний шрифт, колір тексту, олівця, стиль лінії та ін. Як тільки віконна функція отримує повідомлення *wm_Paint*, вона повинна створити новий контекст, викликавши функцію *BeginPaint* ОС Windows. *BeginPaint* повертає постання на контекст дисплея. Далі програма може виводити текст чи графіку у визначене вікно. Коли виведення закінчено, програма повинна повідомити про це викликом функції *EndPaint*. Для більшості функцій виведення Windows (виведення тексту рисування еліпсів, відрізків та ін.) посилання на контекст дисплея потрібно задавати як перший параметр.

8. Програма обертання об'єктів-фігур

8.1. Модифікація класу *tScreen* для *Microsoft Windows*

Щоб зобразити наші об'єкти-фігури в середовищі Windows, потрібно модифікувати методи класу *tScreen*, що відповідають за графічні побудови. Нагадаємо, що вони називаються *lineTo* і *circle*. Замість функцій бібліотеки BGI використаємо GDI функції Windows: *LineTo* та *Ellipse*.

У попередньому пункті йшлося про те, як часто використовують посилання на екземпляр програми, вікно та контекст дисплея під час звертання до API та GDI функцій. Тому доцільно буде доповнити клас *tScreen* полями *hCurrentInstance*, *hWindow*, *hDeviceContext* для зберігання відповідних посилань. Інтерфейс класу потрібно доповнити також методами отримання та збереження посилання на екземпляр програми.

Як уже було сказано, графічні примітиви в середовищі Windows будуються з використанням олівця певного кольору. Наперед визначеними є чорний і білий олівці: *black_Pen* і *white_Pen*. Посилання на них можна отримати за допомогою функції *GetStockObject*. Олівці інших кольорів потрібно створити за допомогою функції *CreatePen*. Після закінчення використання всі створені олівці потрібно знищити за допомогою *DeleteObject*.

У закриту частину класу *tScreen* введено поле *forColor* для зберігання кольору поточного олівця і масив *pens* посилань на олівці (шістнадцяти кольорів), два з яких є визначеними в системі, а інші створюють під час виконання методу *Initialize*. Метод *cleanUp* знищує створені в програмі олівці. Встановлення поточного олівця виконують процедурою *SelectObject*, виклик якої є в тілі методу *setForeground*.

Нижче наведена реалізація класу *tScreen* для середовища Microsoft Windows. Зауважимо тільки, що вона не підтримує різних кольорів фону. Він завжди залишається білим, оскільки використовується стандартний пензель *white_Brush*.

```

unit ScrClass;
interface uses WinTypes, WinProcs;
const virtual_maxX=110; virtual_maxY=110;
        Black=0;           Blue=1;           Green=2;           Cyan=3;
        Red=4;             Magenta=5;        Brown=6;          LightGray=7;
        DarkGray=8;        LightBlue=9;    LightGreen=10;    LightCyan=11;
        LightRed=12;      LightMagenta=13; Yellow=14;        White=15;
type Colors = Black..White;
        Point = record x,y:integer end;
        tScreen = object
constructor Init;
        procedure getInstance(var h: THANDLE);
        procedure setInstance(h: THANDLE);
        procedure initialize(hWind: HWND; hDevCont: HDC);

```

```

procedure cleanUp;
procedure moveTo(p: tPoint);
procedure lineTo(p: tPoint);
procedure circle(p: tPoint; r: integer);
procedure setBackground(c:Colors);
    function background:byte;
procedure setForeground(c:Colors);
    function foreground:byte;
procedure clear;
    private
        hWindow: HWND; hDeviceContext: HDC;
        hCurrentInstance: THANDLE;
        maxX, maxY: word; forcolor,backcolor: Colors;
        pens: array[Colors]of HPen;
procedure translate(p:tPoint; var newP:tPoint);
    function translateX(x:integer):integer;
    function translateY(y:integer):integer;
        end; {tScreen}
var screen: tScreen;
{=====} implementation
constructor tScreen.Init;
    begin hWindow:=0; hDeviceContext:=0;
        hCurrentInstance:=0; maxX:=0; maxY:=0;
        forcolor:=Black; backcolor:=White
    end;
procedure tScreen.getInstance;
    begin h:=hCurrentInstance end;
procedure tScreen.setInstance;
    begin hCurrentInstance:=h end;
procedure tScreen.translate(p:tPoint; var newP:tPoint);
    var x,y:integer;
    begin with p do
        begin newP.x:=translateX(x);
            newP.y:=translateY(y)
        end
    end;
function tScreen.translateX(x:integer):integer;
    begin translateX:=round(maxX*x/virtual_maxX) end;
function tScreen.translateY(y:integer):integer;
    begin translateY:=maxY-round(maxY*y/virtual_maxY) end;
procedure tScreen.initialize;
    var aRect: TRECT;
    begin hWindow:=hWind; hDeviceContext:=hDevCont;
        getClientRect(hWind,aRect); {помістити в aRect розміри}
        maxX:=aRect.right; maxY:=aRect.bottom; {внутр. обл. вікна}
        pens[Black]:=getStockObject(Black_Pen);
        pens[Blue]:=CreatePen(ps_Solid,0,RGB(0,0,$8F)); {RGB задає}
        pens[Green]:=CreatePen(ps_Solid,0,RGB(0,$8F,0)); { вагові}
        pens[Cyan]:=CreatePen(ps_Solid,0,RGB(0,$8F,$8F)); { частки}
        pens[Red]:=CreatePen(ps_Solid,0,RGB($8F,0,0)); {червоного,}
        pens[Magenta]:=CreatePen(ps_Solid,0,RGB($8F,0,$8F)); {зеле-}
        pens[Brown]:=CreatePen(ps_Solid,0,RGB($8F,$8F,0)); { ного,}
        pens[LightGray]:=CreatePen(ps_Solid,0,RGB($8F,$8F,$8F));
        pens[DarkGray]:=CreatePen(ps_Solid,0,RGB($F,$F,$F)); { си-}

```

```

pens[LightBlue]:=CreatePen(ps_Solid,0,RGB(0,0,$FF)); {НЬОГО}
pens[LightGreen]:=CreatePen(ps_Solid,0,RGB(0,$FF,0));
pens[LightCyan]:=CreatePen(ps_Solid,0,RGB(0,$FF,$FF));
pens[LightRed]:=CreatePen(ps_Solid,0,RGB($FF,0,0));
pens[LightMagenta]:=CreatePen(ps_Solid,0,RGB($FF,0,$FF));
pens[Yellow]:=CreatePen(ps_Solid,0,RGB($FF,$FF,0));
pens[White]:=getStockObject(White_Pen);
end;
procedure tScreen.cleanup;
  var c: Colors;
  begin hWindow:=0; hDeviceContext:=0;
    maxX:=0; maxY:=0;
    for c:=Blue to Yellow do
      DeleteObject(pens[c])
    end;
procedure tScreen.moveTo(p:tPoint);
  begin if hWindow=0 then Exit;
    translate(p,p);
    with p do WinProcs.moveTo(hDeviceContext,x,y)
  end;
procedure tScreen.lineTo(p:tPoint);
  begin if hWindow=0 then Exit;
    translate(p,p);
    with p do WinProcs.lineTo(hDeviceContext,x,y)
  end;
procedure tScreen.Circle(p:tPoint; r:integer);
  var xr,yr:integer;
  begin if hWindow=0 then Exit;
    translate(p,p);
    xr:=round(maxX*r/virtual_maxX);
    yr:=round(maxY*r/virtual_maxY);
    r:=(xr+yr) div 2;
    with p do WinProcs.Ellipse(hDeviceContext,x-r,y-r,x+r,y+r)
  end;
procedure tScreen.setBackground(c:Colors);
  begin {do nothing} end;
function tScreen.background:byte;
  begin background:=backcolor end;
procedure tScreen.setForeground(c:Colors);
  begin WinProcs.SelectObject(hDeviceContext,pens[c]);
    forcolor:=c end;
function tScreen.foreground:byte;
  begin foreground:=forcolor end;
procedure tScreen.clear;
  begin {do nothing} end;
end. {UNIT}

```

8.2. Головна програма

Структура Windows-програми детально обговорена, раніше. Нижче наведено текст нашої програми обертання об'єктів-фігур. У ній оголошено процедуру *WinMain* (точка входу програми), функцію реєстрації класу вікон *InitApplication*, функцію створення екземпляра *InitInstance*, віконну функцію

MainWndProc, а також, крім перерахованих “традиційних”, діалогову функцію *About*.

Діалогова функція обробляє повідомлення, що надсилаються панелі діалогу. Панелі діалогу можуть бути двох типів – модальні і немодальні. Програма використовує *модальну панель*. Така панель не дає змоги користувачеві взаємодіяти з іншими вікнами програми доти, доки він не закриє її. Описана вище модальна панель діалогу “*AboutBox*” закриється, якщо натиснути на кнопку “*Ok*”, чи вибрати команду *Close* системного меню.

Діалогова функція також є функцією опосередкованого виклику. Своєю структурою і призначенням вона нагадує віконну функцію програми. Відмінністю є те, що діалогова функція не містить виклику вповноваженого обробника повідомлень, яким для неї є функція *DefDlgProc*. Ядро Windows викликає її при потребі автоматично, підтримуючи роботу панелі діалогу.

Додаткові пояснення будуть наведені після тексту програми

```

program Main; {$R menu.res}
  uses WinTypes, WinProcs, Strings, ScrClass, Rotation;
  const AppName='RotateWClass';
          idm_About=101;
function About(hDlg:HWND; Message, Wparam:word;
               Lparam:LongInt):LongBool; export;
begin About:=true;
      case message of
        wm_InitDialog: Exit;
        wm_Command: if (wParam=idOk) or (wParam=idCancel) then
          begin EndDialog(hDlg,0); Exit end
      end; About:=false
end; {About}
function MainWndProc(hWind:HWND; Message, Wparam:word;
                    Lparam:LongInt):LongInt; export;
var lpProc:TFarProc; ps:TPaintStruct; hDevCont:HDC;
     hInst:THandle;
begin MainWndProc:=0;
      case Message of
        wm_Command: case wParam of
          idm_About:begin screen.GetInstance(hInst);
                        lpProc:=MakeProcInstance(@About,hInst);
                        DialogBox(hInst, 'AboutBox', hWind, lpProc);
                        FreeProcInstance(lpProc); Exit end;
          else
            MainWndProc:=DefWindowProc(hWind,Message,Wparam,Lparam)
          end; {case wParam}
        wm_Paint: begin hDevCont:= BeginPaint(hWind,ps);
                       TextOut(hDevCont,10, 4, 'Well-designed class interfaces',30);
                       TextOut(hDevCont,10,20, 'produce modular objects and',27);
                       TextOut(hDevCont,10,36, 'replaceable system components.',30);
                       screen.Initialize(hWind,hDevCont); RotateShapes;
                       screen.CleanUp; EndPaint(hWind,ps); Exit end;
        wm_Destroy: begin PostQuitMessage(0); Exit end
      end; {case Message}
      MainWndProc:=DefWindowProc(hWind,Message,Wparam,Lparam)

```

```

end; {WindowProc}
function InitApplication:BOOL;
var wcl: TWndClass;      {задати загальні властивості класу вікон}
begin wcl.style:=cs_HRedraw or cs_VRedraw;  {стиль вікон класу}
      wcl.lpfWndProc:=@MainWndProc;  {адреса фіконної функції}
      wcl.cbClsExtra:=0; wcl.cbWndExtra:=0; {додаткова пам'ять}
      wcl.hInstance:=HInstance;        {посилання на екземпляр}
      wcl.hIcon:=LoadIcon(0,idi_Application); {посил.на іконку}
      wcl.hCursor:=LoadCursor(0,idc_Arrow); {посилан.на курсор}
      wcl.hbrBackGround:=GetStockObject(white_Brush); {фон}
      wcl.lpszMenuName:='RotateMenu';      {ім'я ресурсу "меню"}
      wcl.lpszClassName:=AppName;         {ім'я класу вікон}
      InitApplication:=RegisterClass(wcl) {Зареєструвати клас!}
end; {InitApplication}
function InitInstance:BOOL;
var hWind:HWND;          {запам'ятати посилання на даний}
begin screen.SetInstance(hInstance);      {екземпляр програми}
      hWind:=CreateWindow(AppName,'BPascal MS Windows Rotate Shapes',
                          ws_OverlappedWindow,cw_UseDefault,cw_UseDefault,
                          cw_UseDefault,cw_UseDefault,0,0,HInstance,nil);
      if hWind=0 then InitInstance:=false
      else          {показати вікно, перерисувати його вміст}
      begin ShowWindow(hWind,CmdShow); UpdateWindow(hWind);
            InvalidateRect(hWind,nil,true); InitInstance:=true end
end; {InitInstance}
procedure WinMain;
var Message:TMsg; {якщо працюючих екземплярів програми ще нема,}
begin if HPrevInst=0 then {визначити і зареєструвати клас вікон}
      if not InitApplication then Halt(255);
      if not InitInstance then Halt(255); {створити новий
            екземпляр і розпочати цикл обробки повідомлень}
      while GetMessage(Message,0,0,0) {вибирає повідомлення}
      do {з черги аплікації в структуру даних типу TMsg}
      begin TranslateMessage(Message);
            {перетворює повідомлення від віртуальних клавіш у
            повідомлення про натискання фізичних клавіш}
            DispatchMessage(Message)
            {передає повідомлення віконній функції}
      end; Halt(Message.wParam)
end;
BEGIN WinMain END.

```

InitApplication. Вікно завжди створюють на основі класу. Властивості класу задають структурою даних комбінованого типу *tWndClass*. Призначення її полів вказано в тексті програми у вигляді коментарів. Стиль *cs_HRedraw or cs_VRedraw* свідчить про те, що вміст вікна буде перерисовуватись у кожному випадку зміни його горизонтального чи вертикального розміру. Можна вказувати також інші стилі: *cs_NoClose* – із системного меню вилучається команда *Close*; *cs_OwnDC* – вікна мають власний дисплейний контекст; *cs_SaveBits* – вміст вікна зберігається як растрове зображення, що дає змогу швидко його відновлювати; інші стилі.

Посилання на *іконку* повертає функція *LoadIcon*. Клас вікон може використовувати стандартні іконки або створені користувачем за допомогою редактора ресурсів. У першому випадку параметрами *LoadIcon* є нуль і один з таких ідентифікаторів: *idi_Application* – прикладна програма; *idi_Asterisk* – інформаційне повідомлення; *idi_Exclamation* – попередження; *idi_Hand* – повідомлення про помилку; *idi_Question* – введення інформації. У другому випадку функції *LoadIcon* передається посилання на екземпляр програми та ім'я ресурсу-іконки.

Тип *курсора* задають викликом функції *LoadCursor*. Призначення її параметрів таке ж, як і *LoadIcon*. Є ряд стандартних курсорів, серед яких *idc_Arrow*, *idc_Cross*, *idc_IBeam*, *idc_Wait* та ін.

Після того, як усім полям структури *tWndClass* присвоєно потрібні значення (визначені атрибути вікон), виконують реєстрацію класу в системі за допомогою *RegisterClass*. Реєстрацію потрібно виконувати тільки для першого екземпляра аплікації.

InitInstance. Кожен екземпляр нашої програми використовує один об'єкт класу *tScreen*. Він містить поле *hCurrentInstance* для зберігання посилання на екземпляр. Це посилання заноситься в нього методом *setInstance*.

Зареєстрований клас вікон визначає їхні основні характеристики. Для створення конкретного вікна даного класу необхідно викликати функцію *CreateWindow*, у параметрах якої можна вказати індивідуальні характеристики вікна:

function *CreateWindow* (*ClassName* {ім'я класу вікон}, *WindowName*: *PChar* {ім'я конкретного вікна, яке виводиться у його заголовку}; *style*: *longint* {стиль вікна}; *x*, *y* {координати лівого верхнього кута}, *width*, *height*: *integer* {розміри}; *wndParent*: *hWnd*{посилання на батьківське вікно}; *menu*: *hMenu* {посилання на власне меню, якщо воно є}; *instance*: *tHandle* {посилання на екземпляр програми}; *param*: *pointer* {додаткова інформація}): *hWnd*;

Заданий у програмі стиль *ws_OverlappedWindow* є комбінованим. Він складається з наступних стилів: *ws_Overlapped* – головне вікно програми; *ws_Caption* – вікно має заголовок; *ws_SystemMenu* – системне меню; *ws_ThickFrame* – розмір вікна можна змінювати, розтягаючи за допомогою миші його рамку; *ws_MinimizeBox* – вікно має кнопку згортання; *ws_MaximizeBox* – вікно має кнопку розгортання.

Віконна функція *MainWndProc* обробляє три повідомлення: *wm_Destroy*, *wm_Paint*, *wm_Command*. Перше з них є завершальним і про нього вже йшлося раніше. Повідомлення *wm_Paint* програма отримує у випадку створення вікна та кожної зміни його розміру. Вікно – це область дисплея, куди програма виводить текстову та графічну інформацію. Перш ніж починати виведення, прикладна програма повинна отримати контекст дисплея. Посилання на контекст повертає функція *BeginPaint*. Оскільки контекст займає досить багато пам'яті, після завершення виведення програма повинна звільнити його за допомогою *EndPaint*.

Крім обертання об'єктів-фігур процедурою *Rotation.RotateShapes*, наша Windows-програма виводить пояснювальний текст за допомогою *TextOut*.

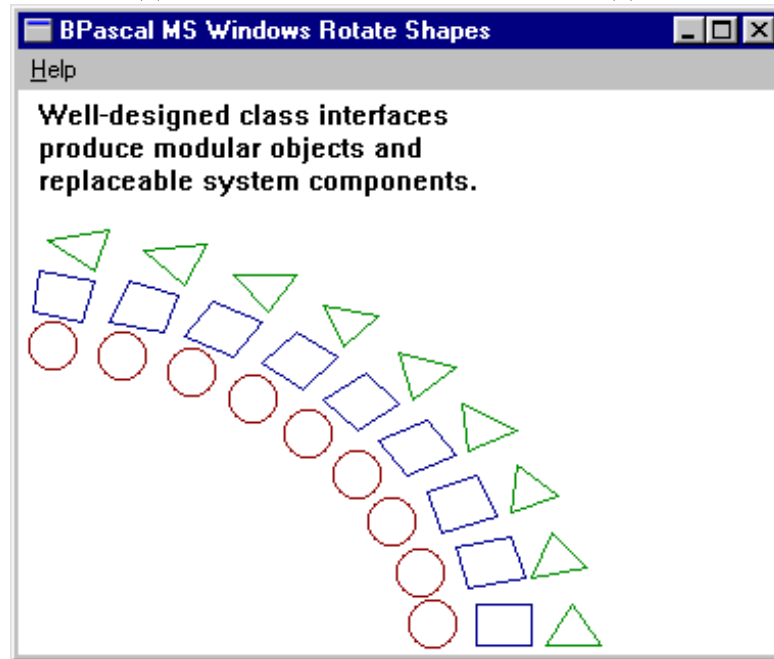


Рис. 12. Використання об'єктів-фігур у середовищі Microsoft Windows.

Повідомлення *wmCommand* програма отримує під час вибору команд рядкового меню. У нашому випадку вибір “About Rotate...” має викликати панель діалогу *AboutBox*. Для цього потрібно завантажити в пам'ять екземпляр діалогової функції за допомогою *MakeProcInstance* та відобразити модальну панель діалогу, викликавши *DialogBox*. Після виклику процедури *DialogBox* ядро Windows передає керування діалоговій функції опосередкованого виклику *About*. Завершення діалогу виконує процедура *EndDialog*. Після цього слід звільнити пам'ять, зайняту функцією *About*, викликавши *FreeProcInstance*.

Тепер завдання перенесення програми в середовище MS Windows виконано. Класи ієрархії *tShape* (ст. 28-30) і модуль *Rotation* (ст. 31-32) використані в ній без змін. Нова реалізація класу *tScreen* (ст. 42-44) скерує виведення графіки програмою до функцій Windows GDI. Виконання головної програми (ст. 44-46) створить на екрані дисплея вікно, зображене на рис. 12.

Детальнішу інформацію про основні прийоми програмування в середовищі Borland Pascal for Windows можна отримати з [5].

9. Список літератури

1. *Greg Voss. Object-Oriented Programming: an Introduction.* – Osborne McGraw-Hill, Berceley, California, USA, 1991.
2. *Гради Буч. Объектно-ориентированное проектирование с примерами применения.* М., 1992.
3. *Зуев Е. А. Язык программирования Turbo Pascal 6.0.* М., 1992.
4. *Музичук А. О., Сибіль Ю. М. Елементи об'єктно-орієнтованого програмування в середовищі Turbo Pascal. Методичні вказівки для студентів факультету прикладної математики.* Львів, ЛДУ, 1994.
5. *Федоров А., Рогаткин Дм. Borland Pascal в среде Wmdows.* К., 1993.