Чернігівський національний педагогічний університет
імені Т.Г. Шевченка

Горошко Ю.В., Костюченко А.О.

ТЕОРІЯ І МЕТОДИКА РОЗРОБКИ ПЕДАГОГІЧНИХ ПРОГРАМНИХ ЗАСОБІВ

Рекомендовано до друку
вченою радою Чернігівського
національного педагогічного
університету ім. Т.Г.Шевченка
від 29 грудня 2010 р.,
протокол № 5

Чернігів, 2011

УДК 37.016:004.415

Горошко Ю.В., Костюченко А.О.

ТЕОРІЯ І МЕТОДИКА РОЗРОБКИ ПЕДАГОГІЧНИХ ПРОГРАМНИХ ЗАСОБІВ. – Ч.: Єрмоленко О.М., 2011, - 144 с.

В посібнику викладені теоретичні засади об’єктно-орієнтованого програмування та методичні елементи створення педагогічних програмних засобів.

Викладення матеріалу ведеться на прикладах, вже, розроблених і впроваджених педагогічних програмних засобів, що дає студенту змогу не тільки вивчити особливості об’єктно-орієнтованого програмування, але і знайомить його з внутрішньою будовою робочих проектів, ієрархією даних, особливостями реалізації методів обробки математичних даних.

Посібник призначений для студентів педагогічних спеціальностей, вчителів які розпочинають вивчення об’єктно-орієнтованого програмування і прагнуть навчитися створювати власні педагогічні програмні засоби.

Рецензенти:

Цибко Ганна Юхимівна - кандидат педагогічних наук, завідувач кафедри інформатики та ОТ, доцент Чернігівського національного педагогічного університету ім. Т.Г.Шевченка.

Гур'єв Володимир Іванович - кандидат технічних наук, завідувач кафедри економічної кібернетики та інформатики, доцент Чернігівського державного інституту економіки та управління.

Рекомендовано до друку вченою радою Чернігівського національного педагогічного університету ім. Т.Г.Шевченка, від 29 грудня 2010 р., протокол № 5

Вступ

Майже одночасно з появою в школі комп'ютерів почали створюватися програми, призначені для навчання школярів програмуванню. Потім з'явилися програми для навчання іншим предметам. За такими програмами та інструментальними засобами закріпився термін „педагогічні програмні засоби".

Педагогічні програмні засоби (ППЗ) – це сукупність комп'ютерних програм, призначених для досягнення конкретної цілі навчання.[5] ППЗ є головною частиною комп'ютерного програмно-методичного комплексу, який включає в себе, крім ППЗ, методичний та дидактичний супровід даних програм.

Проте основною проблемою використання ППЗ в навчанні є те, що дуже часто програмні засоби спочатку створюються, а потім розпочинаються способи знайти їм місце в навчальному процесі. Ці програми створюють частіше за все не так, як це потрібно учню та вчителю, а так, як це зручно і зрозуміло розробнику. Так наприклад, чи не першими ППЗ стали численні „автоматизовані навчаючі системи" – просто тому, що подібні програмні засоби використовувалися на підприємствах, і в момент появи комп'ютерів в масовій школі стали актуальними задачі автоматизації та створення „автоматизованих робочих місць". Зручність і простота розробки викликали появу великої кількості електронних задачників – текстів з викладеним навчальним матеріалом, які супроводжувалися в кращому випадку графічними ілюстраціями; багато контролюючих програм, в яких зазвичай потрібно з декількох запропонованих відповідей вибрати вірну.

В зв'язку з цим можна вказати на системи, інколи дуже потужні, призначені для розв'язування навчальних задач. Основний їх недолік в тому, що в таких системах незрозуміло, навіщо взагалі потрібен учень – майже все за нього робить програма.

В іншому випадку, коли розробкою ППЗ займається вчитель, мало обізнаний в особливостях і можливостях мов програмування, програми виявляються безпорадними з точки зору можливостей вибраного середовища

програмування. Вони використовують можливості комп'ютера лише частково, проте найчастіше в їх основу покладена грамотна методика, і програми досить часто дидактично обгрунтовані.

Разом з тим методика викладання кожного навчального предмету в свою чергу враховує особливості відповідної науки, тому правомірно говорити про методичні вимоги до ППЗ, які передбачать специфіку конкретної науки і відповідного їй навчального предмету. Визначаючи педагогічні вимоги, які ставляться перед ППЗ, необхідно враховувати також особливості вибору тематики ППЗ, аргументоване певними методичними цілями, і забезпечувати перевірку педагогічної ефективності використання ППЗ. Більш детально про вимоги до ППЗ розглянуто в наступному параграфі.

Зважаючи на вказане, можна з впевненістю сказати, що лише невелика частина ППЗ можуть ефективно використовуватися при викладанні в школі.

Тому є необхідність навчати студентів педагогічних вузів, відповідних спеціальностей, не просто програмуванню, а вчити створювати ППЗ.

Проте, з іншого боку, перш ніж створювати ППЗ, студент повинен вміти програмувати. Як писав Чарльз Уезерелл [11], викладання програмування – справа майже безнадійна, його вивчення – непосильна праця. Викладач може по-різному займатися зі студентами, читати лекції, робити критичні зауваження, направляти вірним шляхом. Студент може все ретельно записувати, запам'ятовувати, читати, здавати заліки, дискутувати. Проте всі зусилля марні, якщо студент не буде практикуватися в написанні програм, оскільки навички програмування набуваються тілки у практиці. Більше того, вчитися потрібно на "справжніх" програмах, а не на дуже спрощених прикладах, якими переповнені підручники з програмування. Особливо актуальними є ці думки стосовно вивчення об'єктно-орієнтовного програмування (ООП), оскільки ООП є інстументом, перш за все, для створення великих програм. Тільки в процесі написання великої програми студент може по-справжньому навчитися моделювати об'єкти предметної області, встановлювати зв'язки між ними, будувати відповідне дерево класів та програмувати методи цих класів.

Розробка програмного забезпечення – складний і трудомісткий процес. При традиційному, не об'єктно-орієнтованому підході, відбувається лавиноподібне нарощування складності розробки програми [3]. Подолати ці проблеми можна шляхом правильного використання об'єктно-орієнтованого підходу до програмування.

Вбачається два шляхи створення сучасних ППЗ. У першому випадку програмуванням займається професійний програміст (колектив програмістів) при активній участі педагогів з тієї галузі, до якої відноситься ППЗ. В цій ситуації володіння педагогом основ ООП дозволить підвищити ефективність взаємодії його з програмістом, прискорити процес розробки ППЗ. У другому випадку сам педагог, що оволодів компетентностями в області ООП, розробляє ППЗ.

Зрозуміло, що на даному етапі досить серйозне вивчення програмування можливо тільки на тих спеціальностях у педвузі, де інформатика є основною спеціальністю, або спеціалізацією. Але тут постає питання, а як же ефективно вивчати основи ООП у педвузі, щоб майбутні педагоги могли плідно брати участь у розробці ППЗ? Нам вбачається, що вивчення повинно базуватися на прикладах вже розроблених успішних ППЗ. В цьому випадку студенти не тільки вивчать основи ООП, але і познайомляться з внутрішньою будовою таких проектів, ієрархією класів, особливостями реалізації методів обробки математичних даних. З іншого боку, недоцільно повністю відкривати програмний код. Це пов'язано з багатьма причинами. За час вивчення такий код неможливо збагнути пересічному студенту. Велика кількість нюансів у реальних класах відволіче студентів від основної ідеї. Є проблеми і з авторським правом. Тому дуже бажано, якщо це можливо, на основі реального ППЗ підготувати завдання, що пов'язані з навчальними ієрархіями класів, реалізаціями методів, які посильні для опанування студентами, але дозволять показати принципи і особливості використання ООП при розробці ППЗ. Розробляючи такі ієрархії класів, студенти зможуть набути навички створення реальних проектів, освоять особливості обробки математичних даних.

1. Поняття парадигми та технології програмування

Наведемо спочатку цитату з тлумачного словника. Парадигма – набір теорій, стандартів та методів, які спільно представляють собою спосіб організації наукового знання. Або іншими словами, спосіб бачення світу.

Багато різних авторів дають дещо відмінні тлумачення поняттю «парадигма програмування». Так наприклад:

- Діомідіс Спінелліс дає таке означення «Слово парадигма використовується в програмуванні для визначення сімейства позначень (нотацій), які розділяють загальний спосіб (методику) реалізації програми». [18]
- Деніел Бобров визначає парадигму як «стиль програмування, як опис намірів програміста». [14]
- Брюс Шрайвер визначає парадигму програмування як «модель чи підхід до розв'язання проблеми». [17]
- Лінда Фрідман – як «підхід до розв'язання проблем програмування» [16]
- Пітер Вегнер парадигму визначає, як «правила класифікації мов програмування в відповідності з деякими умовами, які можуть бути перевірені». [19]
- Тімоті Бадд пропонує розуміти термін «парадигма», як «спосіб концептуалізації того, що означає "виконувати обчислення", і як задачі, які підлягають розв'язанню на комп'ютері, мають бути структуровані і організовані». [15]

Виходячи з вищесказаного, поняття парадигми програмування можна сформулювати наступним чином. *Парадигми програмування* – це сукупність ідей та понять, які визначають стиль написання програми.

Парадигма, в першу чергу, визначається базовою програмною одиницею. В сучасній індустрії програмування дуже часто парадигма

програмування визначається набором інструментів програміста, а точніше, мовою програмування і бібліотеками, що використовуються.

Відомо декілька основних парадигм програмування, найважливішими з яких на даний момент є парадигма імперативного, функціонального, логічного, об'єктно-орієнтованого програмування.

Імперативне програмування – це парадигма програмування, яка описує процес обчислення у вигляді інструкцій, які змінюють стан програми. Тобто це послідовність команд, які має виконати комп'ютер.

Функціональне програмування – це парадигма програмування, в якій процес обчислення трактується як обчислення значень функцій в математичному розумінні останніх (на відмінність від функцій як підпрограм в процедурному програмуванні). На практиці відмінність математичної функції від поняття «функції» в імперативному програмуванні полягає в тому, що імперативні функції взаємодіють і змінюють вже визначені дані. Таким чином, в імперативному програмуванні, при виклику однієї і тієї ж функції з однаковими параметрами можна отримати різні дані на виході, через вплив на функцію зовнішніх факторів. А в функціональній мові при виклику функції з одними і тими ж аргументами ми завжди отримаємо однаковий результат.

Логічне програмування – це парадигма програмування, яка базується на теорії і апараті математичної логіки з використанням математичних принципів резолюції. Логічні мови програмування, такі як Prolog, зазвичай визначають, що потрібно обчислити, а не як.

Об'єктно-орієнтовне програмування – це парадигма програмування, в якій основними концепціями є поняття об'єктів і класів, які взаємодіють між собою за допомогою повідомлень.

Парадигма програмування визначає, в яких термінах програміст описує логіку програми. Наприклад, в імперативному програмуванні програма описується, як послідовність дій, а в функціональному

програмуванні подається у вигляді виразу і множини визначень функцій. В популярному об'єктно-орієнтованому програмуванні програму прийнято розглядати як набір взаємодіючих об'єктів. ООП є по суті імперативне програмування, яке доповнене принципом інкапсуляції даних і методів в об'єкт.

Важливо відмітити, що парадигма програмування не визначає однозначно мову програмування. Багато сучасних мов програмування є мультипарадигменними, тобто допускають використання різних парадигм. Так мовою C++, яка є об'єктно-орієнтованою, можна писати чисто імперативні програми без використання об'єктів, а на Ruby, в основу якого в значній мірі покладена об'єктно-орієнтовна парадигма, можна писати згідно стилю функціонального програмування.

Крім парадигми програмування важливе місце займає технологія програмування.

Технології програмування – це система методів, способів і прийомів розробки і налагодження програм. Тобто під технологією програмування ми будемо розуміти технологію розробки програмних засобів, включаючи всі процеси, починаючи з моменту зародження ідеї програмного засобу. Гарна технологія програмування дає можливість отримати суттєвий ріст продуктивності праці програмістів та підвищує якість програмного продукту.

Серед технологій програмування можна виділити структурну, модульну та об'єктно-орієнтовну.

В основі *структурної технології програмування* лежить подання програми у вигляді ієрархічної структури блоків. У відповідності з даною методологією:

- будь-яка програма представляє собою структуру, побудовану з трьох типів базових конструкцій (послідовне виконання, розгалудження, цикл);

- фрагменти програми, що повторюються, можуть оформлюватися у вигляді так званих підпрограм;
- розробка програми відбувається покроково, методом «зверху вниз».

Модульна технологія програмування полягає в розбитті програми на логічні частини, які називають програмними модулями. Програмний модуль – це будь-який фрагмент опису процесу, оформлений як самостійний програмний продукт, який можна використовувати в описах процесу.

Об'єктно-орієнтовна технологія програмування полягає в поданні програми у вигляді сукупності об'єктів, кожен з яких є екземпляром певного класу (класи утворюють ієрархію наслідування) і має інтерфейс у вигляді набору методів для взаємодії один з одним.

2. Семантика мов програмування

Семантика – система правил визначення поводження окремих мовних конструкцій. Семантика визначає смислове значення інструкцій алгоритмічних мов.

Існує декілька підходів до визначення семантики мов програмування. Найбільш широко розповсюджені такі три різновиди: операційний, денотаційний (математичний) і деріваційний (аксіоматичний).

При описі семантики в рамках *операційного* підходу зазвичай виконання конструкцій мов програмування інтерпретується за допомогою деякої уявної ЕОМ. *Деріваційна* семантика описує наслідки виконання конструкцій мови за допомогою мови логіки і задання перед- і післяумов. *Денотаційна* семантика оперує поняттями, типовими для математики – множини, відповідності, судження та твердження.

3. Вимоги до педагогічних програмних засобів

Загальновідомо, що розробка ПЗ, які використовуються в навчальних цілях, являє собою дуже складний процес, що вимагає колективної праці не тільки вчителів, методистів, програмістів, але й психологів, гігієністів, дизайнерів. У зв'язку із цим правомірно висувати комплекс вимог до створюваних ППЗ, щоб їх використання не викликало б негативних (у психолого-педагогічному або фізіолого-гігієнічному змісті) наслідків, а служило б цілям інтенсифікації навчального процесу, розвитку особистості учня.

Виходячи з цього, можна перерахувати основні вимоги, які ставляться до ППЗ:

педагогічні вимоги (дидактичні, методичні, обґрунтування вибору тематики);

технічні вимоги;

ергономічні вимоги;

фізіологічно-гігієнічні вимоги;

естетичні вимоги;

вимоги до оформлення документації.

Зупинимося більш детально на розкритті сутності педагогічних вимог, які висуваються до розроблювальних ППЗ.

Дидактичні вимоги до ППЗ. Вимога забезпечення науковості змісту ППЗ передбачає подання засобами програми науково достовірних відомостей (по можливості методами досліджуваної науки). При цьому можливість моделювання, імітації досліджуваних об'єктів, явищ, процесів (як реальних, так і "віртуальних") може забезпечити проведення експериментально-дослідницької діяльності, що ініціює самостійне "відкриття" закономірностей досліджуваних процесів, і разом з тим наближає шкільний експеримент до сучасних наукових методів дослідження.

Вимога забезпечення *доступності* означає, що навчальний матеріал запропонований програмою, форми й методи організації навчальної діяльності повинні відповідати рівню підготовки учнів і їхнім віковим особливостям. Встановлення того, чи доступний розумінню учня запропонований за допомогою ППЗ навчальний матеріал, чи відповідає він раніше набутим знанням, умінням і навичкам, відбувається за допомогою тестування. Від отриманих результатів залежить подальший хід навчання з використанням ППЗ.

Вимога *адаптивності* (пристосованості ППЗ до індивідуальних можливостей учня) передбачає реалізацію індивідуального підходу до учня, врахування індивідуальних можливостей сприйняти навчального матеріалу. Реалізація адаптивності може забезпечуватися різними засобами наочності, декількома рівнями диференціації при поданні навчального матеріалу за складністю, обсягом, змістом.

Вимога забезпечення *систематичності й послідовності* навчання з використанням ППЗ передбачає необхідність засвоєння учнем системи понять, фактів і способів діяльності в їхньому логічному зв'язку з метою забезпечення послідовності й наступності в оволодінні знаннями, уміннями й навичками.

Вимога забезпечення *комп'ютерної візуалізації навчальної інформації* запропонованим ППЗ, передбачає реалізацію можливостей сучасних засобів візуалізації (наприклад, засобів комп'ютерної графіки, технології мультимедіа) об'єктів, процесів, явищ (як реальних, так і "віртуальних"), а також їхніх моделей, подання їх динаміки розвитку, в часовому та просторовому русі, зі збереженням можливості діалогового спілкування із програмою.

Вимога забезпечення *свідомості* навчання, *самостійності й активізації діяльності* учнів передбачає забезпечення засобами програми самостійних дій по здобуттю навчальної інформації при чіткому розумінні

конкретних цілей і завдань навчальної діяльності. Активізація діяльності учня може забезпечуватися можливістю самостійного керування ситуацією на екрані, вибору режиму навчальної діяльності; варіативності дій у випадку ухвалення самостійного рішення; створення позитивних стимулів, що спонукують до навчальної діяльності, що підвищують мотивацію навчання (наприклад, часткове застосування ігрових ситуацій, гумор, доброзичливість при спілкуванні, використання різних засобів візуалізації).

Вимога забезпечення *міцності засвоєння результатів навчання* передбачає забезпечення усвідомленого засвоєння учнем змісту, внутрішньої логіки й структури навчального матеріалу, що надаються засобами ППЗ. Ця вимога досягається здійсненням самоконтролю й самокорекції; забезпеченням контролю на основі зворотного зв'язку, з діагностикою помилок за результатами навчання і оцінкою результатів навчальної діяльності, поясненням сутності допущеної помилки; тестуванням, що констатує просування в навчанні.

Вимога забезпечення *інтерактивного діалогу* передбачає необхідність його організації за умови забезпечення можливості вибору варіантів змісту досліджуваного матеріалу, а також режиму навчальної діяльності можливостями ППЗ.

Вимога розвитку *інтелектуального потенціалу* учня передбачає забезпечення: розвитку мислення (наприклад, алгоритмічного, програмістського стилю мислення, наочно-образного, теоретичного); формування вміння приймати оптимальні рішення або варіативні рішення в складній ситуації; формування вмінь з обробки інформації (наприклад, на основі використання систем обробки даних, інформаційно-пошукових систем, баз даних).

Методичні вимоги до ППЗ передбачають необхідність: ураховувати своєрідність і особливості конкретного навчального предмета; передбачати специфіку відповідної науки, її понятійного апарату, особливості методів

дослідження її закономірностей; реалізації сучасних методів обробки інформації.

Обґрунтування вибору теми навчального предмета (курсу) при розробці ППЗ необхідно аргументувати педагогічною доцільністю його використання.

Ергономічні вимоги до змісту й оформлення ППЗ обумовлюють необхідність:

- ураховувати вікові й індивідуальні особливості учнів, різні типи організації нервової діяльності, різні типи мислення, закономірності відновлення інтелектуальної й емоційної працездатності;
- забезпечувати підвищення рівня мотивації навчання, позитивні стимули при взаємодії учня з ППЗ (доброзичлива й тактовна форма звертання до учня, можливість кількаразового звертання до програми у випадку невдалої спроби, можливість використання в програмі ігрових ситуацій);
- встановлювати вимоги до зображення інформації (кольорова гама, розбірливість, чіткість зображення), до ефективності зчитування зображення, до розташування тексту на екрані ("віконне", табличне, у вигляді тексту, що заповнює весь екран, і т.д.), до режимів роботи з ППЗ.

Естетичні вимоги до ППЗ встановлюють: відповідність естетичного оформлення функціональному призначенню; відповідність кольорового колориту призначенню ППЗ і ергономічним вимогам; упорядкованість і виразність графічних і образотворчих елементів ППЗ.

Окремі елементи ергономічних та естетичних вимог обумовлюють створення зручного та зрозумілого інтерфейсу користувача.

Програмно-технічні вимоги до ППЗ передбачають вимоги по забезпеченню: стійкості до помилкових і некоректних дій користувача;

мінімізації часу на дії користувача; ефективного використання технічних ресурсів (у тому числі й зовнішньої пам'яті); відновлення системної області перед завершенням роботи програми; захисту від несанкціонованих дій користувача; відповідності функціонування ППЗ опису в експлуатаційній документації.

Вимоги до оформлення документації на розробку й використання ППЗ встановлюють єдиний порядок побудови й оформлення основних документів на розробку й використання ППЗ, які створюються в установах і організаціях незалежно від їхньої відомчої приналежності.

4. Критерії якості ППЗ

Для ППЗ, як і будь-якого іншого ПЗ можна висувати такі критерії щодо його якості:

- функціональність
- надійність
- легкість застосування
- ефективність
- мобільність

Функціональність – це здатність ПЗ виконувати набір функцій, які задовольняють завдання чи можливі потреби користувачів, в межах області застосування ПЗ.

Надійність ПЗ – це його здатність безвідмовно виконувати визначені функції при заданих умовах протягом заданого періоду часу з достатньо великою ймовірністю. При цьому під відмовою в ПЗ розуміють виникнення в ньому помилок. Таким чином, надійний ПЗ не виключає наявність в ньому помилок – важливо лише, щоб ці помилки при практичному використанні даного ПЗ в заданих умовах проявлялись достатньо рідко. Впевнитися, що

ПЗ відповідає даній вимозі, можна при його використанні шляхом тестування, а також при практичному застосуванні.

Легкість застосування – це характеристика ПЗ, яка дозволяє мінімізувати зусилля користувача по підготовці вхідних даних, використанню ПЗ і аналізу отриманих результатів, а також викликати позитивні емоції пересічного користувача.

Ефективність – це відношення рівня послуг, які надає ПЗ користувачу при певних умовах до об'єму використаних ресурсів.

Мобільність – це здатність ПЗ бути переміщеним з однієї операційної системи в іншу без особливих зусиль.

Зрозуміло, що до критеріїв якості потрібно віднести виконання специфічних вимог, що стосуються ППЗ даного класу.

5. Особливості об'єкто-орієнтовного програмування в середовищі Lazarus

Перш ніж розпочати розгляд основ створення ППЗ, необхідно розглянути особливості мови, середовища програмування, в межах якого будуть створюватися майбутні ППЗ. Зокрема ми зупинимося на середовищі Lazarus, яке базується на об'єкто-орієнтовній мові Object Free Pascal і перш за все визначимо деякі поняття.

Клас – це визначений користувачем тип даних, який має стан (його подання: поля чи властивості), низку операцій (його поведінка: методи) та подій, на які він може реагувати.

Об'єкт – це екземпляр класу чи змінна типу даних, визначеного цим класом.

Атрибут класу – це поля і методи, які характеризують клас.

Атрибути об'єкту – це значення полів, які характеризують об'єкт в його класі. Атрибути об'єкту можуть бути постійні і змінні, в процесі життя об'єкту.

5.1. Концепції ООП

Будь-яку об'єктно-орієнтовану мову програмування характеризують три основні властивості, а саме: *інкапсуляція*, *наслідування* і *поліморфізм*.

Перед розглядом вказаних концепцій розглянемо не таку важливу, проте необхідну концепцію абстрагування.

Абстрагування виділяє важливі характеристики деякого об'єкту, які відрізняють його від всіх інших видів об'єктів і таким чином чітко визначає його концептуальні межі з точки зору предметної області.

Абстрагування фокусується на суттєвих з точки зору предметної області характеристиках об'єкту.

Розглянемо поняття автомобіля і в якості приклада візьмемо таку важливу його частину, як двигун внутрішнього згорання. Для кожної сфери використання автомобіля існує свій набір представлень і асоціацій, пов'язаних з цим поняттям. Це об'єкт, який у взаємодії з іншими об'єктами (системою живлення, карбюратором, коробкою передач) забезпечує певну функціональність. Ця функціональність полягає в здатності створення обертального моменту валу, який передається коробці передач. При цьому двигун використовує в роботі інші системи: систему запалювання і систему подачі палива. В нашому випадку двигун є абстракцією, виділеною на основі своєї найбільш значущої характеристики, а точніше, здатності перетворювати хімічну енергію палива, в механічну енергію обертання розподільчого валу. Продовжуючи аналіз і декомпозицію автомобіля, можна виділити набір абстракцій: двигун, коробка передач, система запалення і т.д. Кожна з цих абстракцій виділяється на основі свого інтерфейсу і функціональності.

Інтерфейс відображає зовнішнє поводження абстракції. Внутрішня реалізація описує представлення цієї абстракції і механізми досягнення бажаного поводження об’єкту.

Інкапсуляція (encapsulation) – це механізм об’єднання даних та коду, який маніпулює цими даними, а також захисту того і іншого від зовнішнього втручання, або невірного використання.

Концепція інкапсуляції часто позначається „Чорним ящиком”. Нема необхідності турбуватися про його вміст, необхідно лише знати, як взаємодіяти з цим ящиком, не розбираючись у внутрішній структурі. Розділ „Як ним користуватися” називається інтерфейсом класу.

Наприклад, по мірі розвитку програмного продукту, розробники можуть прийняти рішення змінити внутрішній устрій певних об’єктів для економії пам’яті чи покращення продуктивності. При цьому перевагою використання інкапсуляції є можливість внесення змін в об’єкт, не змінюючи інтерфейсу, без впливу на інші об’єкти.

Якщо повернутися до розглядуваного автомобіля, то можна продемонструвати принцип інкапсуляції на прикладі реалізації акселератора. В одному випадку механізм зв’язку між педаллю газу і заслінкою карбюратора може бути виповнений за допомогою складної системи важелів і тросів, а в іншому – за допомогою датчика сигналу, з якого подається на електронний прилад управління карбюратором. В будь-якому випадку інтерфейсом для водія залишається натиснення на педаль акселератора.

Наслідування (inheritance) – можливість будувати класи-нащадки на основі вже створеного класу, які наслідують властивості (як поля, так і методи) свого „батька”, а крім того можуть містити нові дані або методи. Набір класів, пов’язаних наслідуванням, називається ієрархією класів.

Поліморфізм (polymorphism) (від грецького polymorphos) - це властивість, що дозволяє те саме ім'я використовувати для розв’язування

схожих, але технічно різних завдань, які відносяться до різних класів, пов'язаних між собою.

Ефект поліморфізму об'єктів полягає в тому, що одне і те саме повідомлення, будучи відісланим різним об'єктам, може призвести до виконання різних дій (виклику різних методів) в залежності від того, який конкретний об'єкт на етапі виконання програми є отримувачем цього повідомлення.

При поліморфізмі деякі частини (методи) батьківського класу замінюються новими, які реалізують специфічні дії для даного нащадка. Виконання кожної конкретної дії буде визначатися типом даних.

З поняттям поліморфізм тісно пов'язане поняття „Пізнього зв'язування".

5.2. Обробка виключних ситуацій

Важливо, щоб програма, яка створюється, була надійною. Надійність програм полягає в тому, що вони повинні обробити помилки, що виникають, у спосіб, відповідний ситуації, а потім, якщо це можливо, повернутися до виконання коду програми, а в протилежному випадку коректно припинити роботу. У мові Object Pascal стан помилок програми відображається у виключеннях. Виключення - це об'єкти, що містять дані про тип помилки, яка виникла, і місце її виникнення. Обробка виключних ситуацій може полягати у виконанні коду завершення, або в корекції стану, що викликав виключення. Виконання коду завершення є простішим випадком і полягає в гарантованому виконанні певного коду. Як правило, цей різновид реакції програми на виключення, що виникло, використовується для гарантії того, що програма звільнить ті ресурси, які вона розподілила, не звертаючи уваги на стан помилки. У програмі це може виглядати так:

```
 {розподіл ресурсу}
try
 {операції, в яких можуть виникнути виключні
    ситуації}
finally
 {звільнення ресурсу}
End;
```

Особливо гостро питання надійності постають у програмах, пов'язаних з обробкою математичних об'єктів, таких, як наприклад функції, оскільки існує проблема області визначення, наявність розривів тощо, що призводить до помилок у обчисленнях з плаваючою точкою. Якісна програма повинна обробити такі помилки, присвоївши відповідній змінній певне значення або, якщо це необхідно, повідомивши про виникнення помилки користувача. У програмі Gran1 при виникненні помилки обчислення відповіді присвоюється певне значення, яке далі можна аналізувати при побудові графіка, обчисленні інтегралу тощо. Обробити таку ситуацію дозволяє наступна синтаксична конструкція:

```
try
 {оператори, в яких може виникнути помилка
(виключення)}
except
 {оператори обробки виключення}
End;
```

Оператори в розділі except виконуються, тільки якщо виключення виникає під час виконання операторів у розділі try (або у довільній процедурі, що викликана з розділу try).

При використанні виключень можна за короткий проміжок часу написати "правильні" оператори алгоритму, а потім дописати обробку виключних ситуацій, якщо це потрібно.

5.3. Модуль даних та його структура

Зважаючи на те, що програмування в Lazarus базується на модульному принципі, тобто групи споріднених класів, які мають розроблятися і змінюватися разом, а також дані, яки ними використовуються, описуються в окремих модулях. В результаті головна програма залишається простою і більш зрозумілою. Крім того, зауважимо, що кожен модуль можна компілювати окремо від інших.

Кожен модуль має жорстку структуру, яка автоматично генерується середовищем при його створенні. Модуль складається з чотирьох частин, будь-яка з яких може бути порожньою: інтерфейсної частини, частини реалізації, частини ініціалізації і частини завершення (останні дві частини є необов'язковими). Спочатку вказується заголовок модуля – ключове слово unit та ім'я модуля (ідентифікатор), яке має співпадати з назвою файлу, в якому зберігається модуль.

```
unit Unit1;
interface              //Інтерфейсна частина
  uses
implementation  //Частина реалізації
  uses
initialization  //Частина ініціалізації
finalization    //Частина завершення
end.
```

Інтерфейсна частина завжди йде першою і починається з ключового слова interface. В цій частині міститься оголошення всіх глобальних елементів модуля (типів, констант, змінних і підпрограм), які мають бути доступними основній програмі та іншим модулям (до яких буде під'єднано даний). *Частина реалізації* починається з ключового слова implementation і містить визначення підпрограм, які було оголошені в інтерфейсній частині. Також в ній можуть бути оголошені локальні для модуля елементи –

додаткові типи, константи, змінні і підпрограми. Елементи, описані в частині реалізації, доступні лише в даному модулі.

Таке розділення модуля на частини дозволяє створювати і розповсюджувати модулі у відкомпільованому вигляді (з розширенням dcu), додаючи до них лише опис інтерфейсної частини, щоб програмісти могли використовувати цей модуль, не маючи його вихідних кодів. При цьому внести зміни в такий модуль неможливо. Такий підхід по-перше дозволяє повторно використовувати раніше написані для інших програм модулі, по-друге розмежовує доступ до модуля його автора і програмістів, що його використовують, по-третє дозволяє розбити програму на набір логічно незалежних модулів.

Після заголовків перших двох частин (інтерфейсної і реалізації) можна додатково вказати модулі, які підключаються до даного модулю, за допомогою ключового слова uses, за яким має йти список модулів через кому з завершальною крапкою з комою. Модулі, які підключені в інтерфейсній частині, доступні з будь-якої частини модуля, а модулі, підключені в частині реалізації, доступні всюди, крім інтерфейсної частини.

Частина ініціалізації починається з ключового слова initialization, а *частина завершення* починається з ключового слова finalization. Вказані в частині ініціалізації оператори виконуються до передачі управління основній програмі і найчастіше використовуються для підготовки роботи модуля (ініціалізація змінних, відкриття необхідних файлів). В частині завершення вказуються оператори, які виконуються після завершення роботи основної програми (звільнюються виділені в модулі ресурси, закриваються файли).

Якщо модулів в програмі декілька, то послідовність виконання їх частин ініціалізації відповідає порядку їх опису в модулі, де вони підключаються за допомогою ключового слова uses, а послідовність виконання частин завершення їй протилежна.

В кінці модуля ставиться end з крапкою.

5.4. Створенная нових класів

Опис нового класу має наступну структуру

```
Type
  <ім'я класу>=class(<базовий клас>)
          {опис полів і методів}
  End;
```

Якщо базовий клас не вказано, то ним автоматично буде клас TObject, який є коренем дерева класів Object Pascal.

Огляд можливостей середовища Lazarus по роботі з класами будемо здійснювати на конкретному прикладі: описати клас точка, що має дві цілочисельні координати і може повідомити значення цих координат в текстовому вигляді. Для спрощення пояснення та розуміння координатами будемо вважати координати графічної області. Оскільки це не буде потребувати переведення координат прямокутної декартової системи в координати графічної області для відображення об‘єкту. Таке переведення буде розглянуте пізніше на прикладі конкретних ПЗ.

Опис такого класу може виглядати так:

```
TDot=class(TObject)
  x, y: Integer;
  function Coord: String;
End;
```

Згідно з існуючими в Lazarus правилами запису в якості префіксу кожного створюваного класу і інших типів використовується літера „T”. Проте це лише порада і для компілятора ніякої різниці немає, яку назву має клас.

Поля класу є змінними, які визначені всередині класу. Вони призначені для зберігання даних під час роботи екземпляра класу (об’єкту). Обмежень на типи полів в класі не передбачено. В описі класу поля мають передувати методам і властивостям. Зазвичай поля слугують для обміну даними між

методами відповідного класу. Вони аналогічні полям запису та унікальні для кожного створеного в програмі екземпляра класу.

Методи – це процедури та функції, які оголошені всередині класу і часто використовуються для операцій над його полями. Від звичайної процедури і функції метод відрізняються тим, що йому передається вказівник на той об'єкт, який його викликав. Тому опрацьовуватися будуть поля того об'єкту, який викликав метод. Кожен метод має один неявний аргумент – об'єкт, до якого він застосовується, цей аргумент доступний як зарезервована змінна *Self*.

Методи характеризуються: іменем, параметрами і типом результату (методи-функції).

Метод в нашому класі оголошено, проте необхідно визначити його реалізацію, яка буде мати наступний вигляд.

```
function TDot.Coord: String;
Begin
  Coord:='['+IntToStr(x)+'; '+IntToStr(y)+']';
End;
```

Як і в більшості сучасних ООП мов (java, c++), в Object Pascal змінна типу клас не є сховищем об'єкту, вона є вказівником (посиланням) на об'єкт, який розміщений в пам'яті. Даний підхід полягає в тому, що при описі змінної ви не створюєте об'єкт в пам'яті, ви лише резервуєте місце в пам'яті для посилання на об'єкт.

Отже, перед тим як використовувати об'єкт, для нього необхідно виділити пам'ять. Таке виділення пам'яті, ініціалізація об'єкту відбувається за допомогою *конструктору*, який за замовчуванням має назву Create. Забігаючи наперед, зазначимо, що створений екземпляр знищується іншим методом *деструктором* (`Destroy`). Проте знищення об'єкту рекомендують виконувати методом Free, який спочатку перевіряє вказівник (чи не рівний він nil), а тільки потім викликає Destroy.

Тоді реалізація роботи з визначеним класом може бути такою.

```
Var MyDot: TDot;
...
Begin
  ...
  MyDot:=TDot.Create;
  ...
  MyDot.Free;
End;
```

В розглянутому прикладі ми не описували власного конструктора, а користувалися батьківським. Під час виклику даного конструктору відбувається власне створення об'єкту – під нього виділяється пам'ять, яка пов'язується із змінною-вказівником на об'єкт. А надання певних значень полям можна реалізувати в основній програмі. Проте часто надання значень полям вносять до конструктора, або передаються до нього як параметри. Для реалізації такої дії необхідно описати власний конструктор.

Перед іменем конструктора ставиться ключове слово constructor. Незважаючи на те, що конструктору можна надати будь-яке ім'я, краще притримуватися стандартного імені Create.

Разом з описом власного конструктора опишемо ще декілька полів та методів, які нам знадобляться в подальшому:

- метод, який буде повертати відстань від нашої точки до початку відліку графічної області, тобто його верхнього лівого кута. Для ефективності реалізації визначимо поле дійсного типу Distance і метод `SetDistance`, в якому будемо змінювати значення цього поля;
- методи відображення (`Show`) та приховування (`Hide`) точки в графічній області, та поле, яке буде містити покажчик стану точки (`Visible : Boolean`);

- метод переміщення точки в нове положення (`MoveTo(NewX, NewY : Integer)`), який полягає в приховуванні точки зі старими координатами, встановлення нових координат точки та відображення її.

В результаті отримаємо таке визначення класу:

```
Type
  TDot=class(TObject)
    x, y: Integer;
    Visible: Boolean;
    Distance: Real;
    constructor create(Ax, Ay: Integer);
    procedure SetDistance;
    function Coord: String;
    procedure Show;
    procedure Hide;
    procedure MoveTo(NewX, NewY: Integer);
  End;
```

і додати необхідні описи:

```
Constructor TDot.Create(Ax, Ay: Integer);
Begin
  x:=Ax;
  y:=Ay;
  Show;
  SetDistance;
End;

Procedure TDot.Show;
Begin
  Visible:=True;
  Pixels[X, Y]:=GetColor;
End;
```

```
Procedure TDot.Hide;
Begin
  Visible:=False;
  Pixels[X, Y]:=GetColor;
End;

Function TDot.MoveTo(NewX, NewY: Integer);
Begin
  Hide;
  X:=NewX;
  Y:=NewY;
  Show;
  SetDistance;
End;

procedure TDot.SetDistance;
Begin
  Distance:=sqrt(sqr(x)+sqr(y))
End;
```

І тепер виконання команди `MyDot:=TDot.Create(10, 15);` призведе не лише до виділення пам'яті під наш об'єкт, а і надасть введені значення його полям.

Але, нажаль, підвищення ефективності призводить до втрати надійності, тому що нам необхідно синхронізувати зміну координат точки з перерахунком її відстані (значення `Distance` має бути синхронізоване з значеннями полів x та y). Для цього в об'єктно-орієнтовних мовах вводиться поняття властивостей, які ми розглянемо дещо пізніше.

5.5. Присвоєння об'єктів

Як зазначалося раніше, Object Pascal базується на *методі об'єктних посилань (object reference model)*. Ідея цього методу полягає в тому, що змінна типу клас не містить цей об'єкт в якості значення, вона лише містить вказівник (Pointer), який вказує, в якому місці пам'яті знаходиться об'єкт.

Тоді наступний запис:

```
Var Dot1,Dot2: TDot;
Begin
  Dot1:=TDot.Create(10,10);
  Dot2:=Dot1;
End;
```

лише копіює адресу пам'яті об'єкту `Dot1` в змінну `Dot2`, що не призводить до копіювання даних одного об'єкту в інший. Тобто вказані змінні будуть посилатися на одну частину пам'яті, і внесення змін через будь-яку змінну є взаємозамінними.

Якщо ж дійсно є необхідність перенести всі дані з одного об'єкту в інший, то необхідно копіювати кожне поле одного об'єкту в інший. Деякі класи мають метод `Assign`, який виконує цю операцію.

Об'єкт передається в підпрограму завжди як посилання і відповідно слова var вживати не потрібно, на відміну від змінних не об'єктного типу. Відповідно, якщо внести зміни в об'єкт в межах підпрограми, такі зміни будуть збережені після завершення роботи підпрограми.

5.6. Наслідування типів

Наслідування (inheritance) – це механізм мови ООП, який дозволяє використовувати властивості вже існуючого класу при описі нового класу і описувати новий клас не цілком, а лише відмінності від батьківського класу.

Клас, від якого відбувається наслідування, називається батьківським класом, суперкласом, класом-предком.

Клас, який є результатом наслідування, називається класом-нащадком, похідним класом, дочірнім класом.

Наявність механізму наслідування прискорює розробку нових програм і робить ОО програми гнучкими і легко модернізованими.

Тобто таким чином будується ієрархічна залежність класів, і в кожному новому класі-нащадку вносяться нові характеристики, або змінюється реалізація вже існуючих.

Всі об'єкти деякого класу можуть мати доступ до методів, реалізованих в батьківських класах.

Ми розглянули клас точка, на основі нього побудуємо похідний клас коло (`TCircle`), в якому вводиться поле, що містить радіус кола, а також додамо метод обчислення його площі.

Отримаємо наступний опис класу:

```
TCircle=class(TDot)
  Rad: Integer;
  constructor create(Ax, Ay, ARad: Integer);
  function Area: Real;
  procedure Show;
  procedure Hide;
  procedure PutCirc(Ax, Ay, ARad: Integer; Color:
TColor);
End;
```

Створений клас отримає від свого батьківського класу поля та методи, описані в класі `TDot`. Щоб правильно ініціалізувати в створюваному об'єкті поля, які відносяться до батьківського класу, необхідно відразу ж при вході в конструктор викликати конструктор батьківського класу за допомогою зарезервованого слова inherited.

Визначення методів класу `TCircle` матиме вигляд:

```
constructor TCircle.create(Ax,Ay,ARad: Integer);
Begin
  inherited create(Ax,Ay);
  Rad:=ARad;
End;

function TCircle.Area: Real;
Begin
  Area:=Pi*Sqr(Rad);
End;

procedure TCircle.Show;
Begin
  Visible:=True;
  PutCirc(X, Y, Rad, GetColor);
End;

procedure TCircle.Hide;
Begin
  Visible:=False;
  PutCirc(X, Y, Rad, GetBkColor);
End;

procedure PutCirc(Ax, Ay, ARad: Integer; Color:
TColor);
var TempColor: Word;
Begin
  TempColor:=Pen.Color;
  Pen.Color:=Color;
  Ellipse(Ax - ARad, Ay - ARad, Ax + ARad, Ay +
ARad);
  Pen.Color:=TempColor;
End;
```

5.7. Підміна, перевизначення та перезавантаження методів

При оголошенні методів можуть бути використані такі ключі (службові слова): `virtual`, `dynamic`, `override`, `overload`, `abstract`.

В залежності від їх використання розрізняють декілька видів методів:

Статичний метод – це метод, оголошений без вказування додаткових ключів (virtual, dynamic, override).

Віртуальний чи динамічний метод – це метод, оголошений з вказуванням ключа (virtual, dynamic, override). В подальшому просто віртуальний метод. Ключі virtual, dynamic використовуються при першому описі такого методу. Якщо вони підміняються у нащадках, то в описі таких методів використовується override.

Як зазначалося раніше, при створенні ієрархії класів всі методи, оголошені в батьківському класі, будуть мати місце і в класі-нащадку. Пороте буває необхідність змінити в нащадку поведінку методу, визначеного в його суперкласі. Для цього в класі-нащадку оголошується метод тієї ж структури (з такою ж назвою, параметрами, типом результату), що і в суперкласі. В залежності від його оголошення даний метод може бути перевизначеним, або підміненим.

Перевизначення методу відбувається, якщо замість деякого методу батьківського класу відбувається оголошення в класі нащадку статичного чи віртуального методу тієї ж структури.

Підмінити метод можливо лише в тому випадку, якщо в батьківському класі він був оголошений як віртуальний, чи вже був підміненим.

Метод, оголошений як статичний, залишається статичним в усіх похідних класах, тільки якщо він не буде „схований” за допомогою нового віртуального методу тієї ж структури. Метод, оголошений як віртуальний

(динамічний), залишається методом, який підтримує пізнє зв'язування в усіх похідних класах, тільки якщо ви не сховаєте його за допомогою нового статичного методу, який нічого не виконує.

Перевизначення чи підміна методу може перслідувати одну з наступних цілей:

- *розширення*: новий метод розширює успадковану операцію, з огляду на вплив атрибутів підкласу;
- *обмеження*: новий метод обмежується виконанням лише частини дій успадкованого методу, зважаючи на специфіку нового підкласу;
- *оптимізація*: використання специфіки підкласу дозволяє спростити і прискорити відповідний метод (наприклад, перевизначення методу пошуку максимального в класі TIntegerSet і його підкласі TSortIntegerSet, ми можемо спростити його, використовуючи специфіку упорядкованих множин);
- зручність.

Перезавантаження методів. Інколи виникає необхідність описати методи (чи звичайні процедури, функції), які будуть виконувати схожі задачі, і будуть відрізнятися кількістю, або типом вхідних параметрів. Наприклад, необхідно обчислити площу трикутника, але відповідний трикутник може задаватися або довжинами своїх сторін, або координатами вершин. Звичайно, можна ці процедури описати з різними іменами, проте існує можливість використати одне ім'я. Необхідно буде лише їх позначити ключовим словом overload (перезавантаження). Перезавантаження означає, що допускається існування принаймні двох методів з однаковими іменами, відмічених ключовим словом overload і тим, що списки параметрів цих методів відрізняються. Вибір необхідного методу відбувається компілятором шляхом перевірки параметрів. Наприклад:

```
Function S(a,b,c: Real): Real; overload;
Function S(ax,ay,bx,by,cx,cy: Real): Real; overload;
```

5.8. Наслідування статичних методів

Як можна бачити, методи, описані в класах, є статичними, і це викликає деяку проблему. Так наприклад, якщо `TCircle` викликає метод `MoveTo` класу `TDot`, то на екрані буде переміщуватися не коло, а точка. Це відбувається тому, що компілятор вніс у тіло методу `TDot.MoveTo` виклики методів `TDot.Hide` і `TDot.Show`. Якщо тепер в об'єкті типу `TCircle` викликати унаслідуваний метод `MoveTo`, то в ньому будуть викликатися методи `TDot.Hide` і `TDot.Show` і по екрану буде переміщуватися не коло, а точка.

Якщо метод `TDot.MoveTo` є статичним, то виходом із такої ситуації може бути його дублювання у класі `TCircle`. Зрозуміло, що це не кращий вихід, оскільки програмний код даремно збільшується та витрачається час програміста.

5.9. Віртуальні методи і поліморфізм

Як можна бачити, таке використання статичних методів не завжди є оптимальним способом розв'язування задач. Вихід з ситуації, яка була розглянута в попередньому пункті, полягає у віртуалізації методів `TDot.Hide` і `TDot.Show`. Віртуальні методи реалізують досить вагомий засіб для узагальнення, який називається поліморфізмом. Його суть полягає в наданні методам з однаковим іменем, яке спільно використовується в ієрархії об'єктів, різної реалізації у кожному нащадку. Може здатися, що це те ж саме, що і наслідування статичних методів, але це не так, і ми побачимо це трохи далі.

Проста ієрархія графічних фігур, про які було вказано раніше, є гарним прикладом поліморфізму в дії, реалізованого засобами віртуальних

методів.

Кожен об'єкт класу в нашій ієрархії представляє окремий тип фігури на екрані: крапку або коло. Пізніше, якщо нам доведеться визначати класи для представлення інших фігур, наприклад, ліній, квадратів, дуг і тому подібне, ми зможемо переписати специфічні для кожного класу методи, що зображують відповідну фігуру на екрані та ховають її (методи `Show` та `Hide`).

Назва «поліморфізм» (грец. багатоформність) пішла від того, що один метод, наприклад `Show`, використовується для показу (в буквальному розумінні) багатьох форм.

5.10. Раннє та пізнє зв‘язування

Процес, в результаті якого виклики статичних методів однозначно вирішуються компілятором під час компіляції, носить назву *раннє зв'язування*. Якраз раннє зв'язування не дозволяло використовувати `TDot.MoveTo` для руху кола. При *пізньому зв'язуванні* метод, який викликає, і метод, який викликають, не можуть бути зв'язані під час компіляції, тобто їх зв'язування відбувається пізніше, коли виклик виконується. Робиться це наступним чином: при компіляції для кожного класу створюється таблиця віртуальних методів, в яку заносяться адреси всіх віртуальних методів цього класу. А в коді замість конкретних адрес методів записуються посилання на відповідні клітинки таблиці віртуальних методів.

Тому віртуалізуємо наші методи `Show` та `Hide`.

В результаті отримаємо такий опис класів:

```
Type
  TDot=class(TObject)
    x, y: Integer;
    Visible: Boolean;
```

```
    Distance: Real;
    constructor create(Ax, Ay: Integer);
    procedure SetDistance;
    function Coord: String;
    procedure Show; virtual;
    procedure Hide; virtual;
    procedure MoveTo(NewX, NewY: Integer);
  End;

  TCircle=class(TDot)
    Rad: Integer;
    constructor create(Ax, Ay, ARad: Integer);
    function Area: Real;
    procedure Show; override;
    procedure Hide; override;
    procedure PutCirc(Ax, Ay, ARad: Integer; Color:
TColor);
  End;
```

Визначення всіх зазначених методів залишиться незмінним. Що ж це нам дало? Тепер, в методі `TDot.MoveTo` немає прямих викликів методів `TDot.Hide` і `TDot.Show`, а записано посилання на таблицю віртуальних методів. Оскільки метод `TDot.MoveTo` викликається з об'єкту класу `TCircle`, в таблиці віртуальних методів записані адреси методів `TCircle.Hide` і `TCircle.Show` і викликатися будуть якраз вони, що і дозволяє методу `TDot.MoveTo` рухати по екрану коло.

5.11. Абстрактні методи

Раніше був розглянутий клас `TDot` (точка) та `TCircle` (коло) і в класі «коло» був визначений метод `Area` (обчислення площі кола). Спираючись на правило сумісності типів (завжди можна використовувати об'єкт класу

нащадка, коли очікується об’єкт класу предка, але не можна використовувати об’єкт класу предка, коли очікується об’єкт класу нащадка), можна змінній типу `TDot` надати значення типу `TCircle`. Але в такому випадку звернутися до методу Area буде неможливо, оскільки в класі `TDot` даного методу немає, тому що не існує поняття площі точки.

З такої ситуації можна вийти приведенням змінної типу `TDot` до типу TCircle за допомогою оператора «as».

Наприклад:

```
Var d : TDot;
Begin
  d:=TCircle.Create(100,100,10);
  TextOut(10,10,FloatToStr((d as TCircle).Area));
  d.Free;
End;
```

Але можна і в `TDot` оголосити метод `Area`, який в ньому визначений не буде, а буде визначеним у класах-нащадках (наприклад `TCircle`). Такий метод називають абстрактним. При описі його необхідно позначити ключовим словом abstract.

Таким чином, *абстрактним* називається метод, який оголошений в класі, проте визначення якого відсутнє, він ніколи не викликається і обов’язково має бути підмінений в нащадках. При спробі визначити такий метод отримаємо попередження від компілятора, а у випадку спроби виклику абстрактного методу (це можливо, оскільки його оголошено) в процесі роботи програми буде згенеровано помилку.

Абстрактними можуть бути лише віртуальні методи.

Доповнивши раніше визначений клас, отримаємо:

```
type
  TDot = class
   . . .
   function Area: Real; virtual; abstract;
```

```
End;
TCircle = class(TDot)
 . . .
 function Area: Real; override;
End;
```

Існує також поняття абстрактного класу. *Абстрактний клас* – це клас, який не може мати екземплярів. Практично це означає, що в ньому присутні методи, які оголошені, але не визначені. Найчастіше такі класи є шаблонами, за якими з використанням механізму наслідування будуються класи реально існуючих в програмі об'єктів.

5.12. Реалізація відношення агрегації

Агрегація – це коли полями класу є раніше описані класи. В цьому випадку у конструкторі нового класу потрібно обов'язково викликати конструктори його полів-класів, а у деструкторі викликати відповідні деструктори полів-класів.

```
TDot=class(TObject)
 . . .
End;
TLine=class(TObject)
  t1, t2: TDot;
  Constructor Create;
  Destructor Destroy;
  Function Len: Real;
End;
constructor TLine.Create;
Begin
  t1:=TDot.Create;
  t2:=TDot.Create;
End;
destructor TLine.Destroy;
```

```
Begin
  t1.Free;
  t2.Free;
End;

function TLine.Len: Real;
Begin
  Len:=Sqrt(sqr(t1.x-t2.x)+sqr(t1.y-t2.y));
End;
```

5.13. Інкапсуляція

Інкапсуляція (encapsulation) – це механізм об'єднання даних та коду, який маніпулює цими даними, а також захисту і того і іншого від зовнішнього втручання, або невірного використання.

При використанні об'єктів більша частина коду залишається закритою. Рідко вдається дізнатися, і це правильно, якими внутрішніми даними оперує об'єкт, і дуже часто відсутня можливість прямого доступу до них. Звичайно припускається, що для звернення до цих даних використовуються методи, які захищені від несанкціонованого доступу. Це є ОО підходом до класичної концепції програмування, яка відома як приховання даних (information hiding).

Lazarus забезпечує інкапсуляцію, засновану на класах, проте використовуючи структуру модуля, можна додати до неї ще і класичну інкапсуляцію, про яку вже говорилося вище.

Для інкапсуляції, основаної на класах, мова Objec Pasclal пропонує три специфікатори доступу: `private` (приватний), `protected` (захищений) і `public` (публічний). Четвертий `published` (опублікований), управляє RTTI-інформацією (run-time type information) і інформацією періоду розробки

(design-time information), проте він дає ту саму програмну доступність, що й специфікатор public.

Директива `public` представляє поля і методи, які вільно доступні з будь-якого розділу програми.

Директива `private` представляє поля і методи класу, які доступні лише в межах даного класу;

Директива `protected` використовується для оголошення методів і полів з обмеженою „видимістю”. Видимість полів і методів класу співпадає з видимістю розділу private з єдиною відмінністю. Елементи розділу protected доступні також всередині класів-нащадків.

В загальному випадку поля класу мають бути оголошені як private, а методи як public. Проте це не завжди має місце. Методи можуть бути приватними чи захищеними, якщо вони необхідні тільки для внутрішнього застосування в цілях виконання певних часткових обчислень чи для реалізації властивостей. Поля можуть бути описані як захищені, для того щоб ними можна було керувати в класах-нащадках.

У наведених вище прикладах всі поля булі публічними, тому ніщо не заважає користувачу самостійно змінити значення поля `Distance`, не звертаючись до методу `SetDistance`, тобто може виникнути ситуація, коли значення поля `Distance` не буде відповідати реальному положенню точки. Отже необхідно поле Distance перенести в область private і таким чином унеможливити доступ до нього поза методами класу. Тепер у коді не можна написати щось подібне до … `TextOut(10,10,FloatToStr(d.Distance))`, тому необхідно потурбуватися про метод доступу до даного поля, щоб ми могли отримати його значення. Зрозуміло, що теж саме можна сказати і про поля координат точок та покажчика стану точки.

Окрім полів, до захищеної області може бути перенесений метод SetDistance, оскільки зміна відстані точки має відбуватися лише за допомогою методу `MoveTo`, який в свою чергу викликає метод `SetDistance`. Тобто немає сенсу надавати користувачу можливість безглуздо викликати зазначений метод.

Зважаючи на це, клас перепишеться у вигляді:

```
TDot=class(TObject)
 private
  x, y: Integer;
  Visible: Boolean;
  Distance: Real;
  procedure SetDistance;
 public
  constructor create(Ax, Ay: Integer);
  function GetX: Integer;
  function GetY: Integer;
  function GetVisible: Boolean;
  function GetDistance: Real;
  function GetX: Integer;
  function GetY: Integer;
  function Coord: String;
  procedure Show; virtual;
  procedure Hide; virtual;
  procedure MoveTo(NewX, NewY: Integer);
End;
```

Довизначення нових методів буде мати вигляд:

```
function TDot.GetX: Integer;
Begin
  GetX:=X;
End;
function TDot.GetY: Integer;
Begin
```

```
  GetY:=Y;
End;


function TDot.GetVisible: Boolean;
Begin
  GetVisible:=Visible;
End;

function TDot.GetDistance: Real;
Begin
  GetDistance:=Distance;
End;
```

5.14. Властивості

В попередньому пункті ми розглянули можливості інкапсуляції, перенісши поля та деякі методи в захищений блок. Це змусило нас описати додаткові методи доступу до значень цих полів. Проте в ОО мовах існує механізм, який дозволяє полегшити використання інкапсуляції. Таким механізмом є використання властивостей.

Властивість – це спеціальний механізм класів, який регулює доступ до полів. Властивості є досить значимим механізмом ООП і гарним обґрунтуванням ідеї інкапсуляції. З точки зору користувача класу, властивість виглядає саме як поле, оскільки її значення зазвичай можна прочитати чи записати.

Проте для читання чи запису значення, властивості можуть безпосередньо посилатися на дані (захищені поля) чи на методи доступу з використанням інструментів read і write.

Опис властивості має такий вигляд:

```
property <ім'я властивості>: <тип властивості> read
          <захищене поле чи метод доступу для
          зчитування властивості поля> write
          <захищене поле чи метод доступу для запису
          значення властивості в захищене поле >;
```

Для прикладу повернемося то описаного раніше класу точки і дещо перевизначимо його призначення і разом з тим функціонування.

Так, наприклад, зважаючи на те, що значення поля Distance змінюється при зміні положення точки, нам необхідно визначити властивість лише для можливості отримання значення з нього.

В такому випадку ми можемо оголосити таку властивість:

```
property Distance: Real read FDistance;
```

У випадку, коли ми будемо зчитувати значення властивості Distance, програма буде здійснювати читання з захищеного поля FDistance, а оскільки інструмент write не описано, то властивість Distance є властивістю тільки для читання, і запис в неї неможливий.

Оскільки властивість Distance має доступ тільки для читання, потрібно змінити код методу SetDistance, вказавши замість Distance ім'я захищеного поля FDistance.

Наступне оголошення

```
property X: Real read FX write SetX;
```

вказує, що при зверненні до властивості `X` дані будуть читатися з захищеного поля `FX`, а при наданні значення властивості `X` буде викликатися метод `SetX`. Оскільки зміна будь-якої координати призводить до зміни положення фігури, в методі `SetX` викликається метод `MoveTo`, який перемальовує точку та змінює значення поля `Distance`. Ті ж міркування справедливі і до координати `Y`.

Допустимими є різні комбінації опису властивостей. Наприклад:

```
property X: Real read GetX write SetX;
```

або

```
property X: Real read GetX write FX;
```

Найчастіше фактичні дані і методи доступу є приватними (`private`) (чи захищеними (`protected`)), в той час як властивості – публічними (`public`).

Таким чином, перевизначивши наш клас, ми отримаємо.

```
Type
  TDot=class(TObject)
   private
    Fx, Fy: Real;
    FDistance: Real;
    Visible: Boolean;
    procedure SetX(const Value: Real);
    procedure SetY(const Value: Real);
    procedure SetDistance;
   public
    constructor create(Ax,Ay: Real);
    property X: Real read FX write SetX;
    property Y: Real read FY write SetY;
    property Distance: Real read FDistance;
    function Coord: String;
    function Area: Real; virtual; abstract;
    procedure Show; virtual;
    procedure Hide; virtual;
    procedure MoveTo(NewX, NewY: Integer);
  End;

Constructor TDot.Create(Ax, Ay: Integer);
Begin
  x:=Ax;
  y:=Ay;
  Show;
  SetDistance;
End;
```

```
Procedure TDot.Show;
Begin
  Visible:=True;
  Pixels[X, Y]:=GetColor;
End;

Procedure TDot.Hide;
Begin
  Visible:=False;
  Pixels[X, Y]:=GetColor;
End;

Function TDot.MoveTo(NewX, NewY: Integer);
Begin
  Hide;
  X:=NewX;
  Y:=NewY;
  Show;
  SetDistance;
End;

procedure TDot.SetX(const Value: Real);
Begin
  If FX=Value Then Exit;
  MoveTo(Value, FY);
End;

procedure TDot.SetY(const Value: Real);
Begin
  If FY=Value Then Exit;
  MoveTo(FX, Value);
End;

procedure TDot.SetDistance;
```

```
Begin
  FDistance:=sqrt(sqr(fx)+sqr(fy))
End;


function TDot.Coord: String;
Begin
  Coord:='['+IntToStr(Fx)+'; '+IntToStr(Fy)+']';
End;
```

Оскільки автори є основними розробниками програмного коду проекту Гран, ними запропонована програма курсу „Технології програмування та створення педагогічних програмних засобів”, що базується на елементах програмного коду проектів Gran1 і Gran2D та принципах побудов цих проектів.

6. Gran1

6.1 Теоретичні відомості про структуру ПЗ Gran1

Розглянемо приклади опису класів, їх дерево наслідування, використання поліморфізму на прикладі класів програми Gran1. На вершині ієрархії описано клас `TFunc`, призначений для зберігання даних про функціональний вираз. Наведемо фрагмент опису цього класу. Деякі несуттєві для розгляду у даному курсі поля та методі в подальших описах будуть опущені

```
type
  TFunc=class(TObject)
    private
    { опис полів }
     ...
    public
```

```
    constructor Create(SF: String);
    function Deriv(Arg: Extended): Extended;
    function PointX(Arg:extended): Extended; virtual;
    function PointY(Arg:extended): Extended; virtual;
    function GetTx: String;
    function Simpson(AI, BI: Extended; IntegralType:
        Byte): Extended;
    procedure ChangeFunc(SF: String; var Chng:
        Boolean);virtual;
    function Eval(ArgX, ArgY: Extended): Extended;
    ...
End;
```

Батьківським класом для `TFunc` є базовий клас мови Object Pascal `TObject`. Навіть, якщо при описі нового класу не вказати батьківський клас, то за замовчуванням ним буде клас `TObject`.

Клас Tfunc містить конструктор `Create(SF: String)`. Конструктор – це специфічний метод, призначений для створення і ініціалізації об'єктів класу. Конструктор займається виділенням пам'яті під об'єкт і ініціалізацією його полів. Для створення об'єкту викликають його конструктор певним чином. У класі `TFunc` конструктор `Create` має єдиний параметр – функціональний вираз. Наведемо приклад створення об'єкту типу `TFunc`, що міститиме вираз 'x':

```
Var f: TFunc;
…
f:=TFunc.Create('x');
```

В описі конструктора потрібно спочатку викликати конструктор батьківського класу, а потім виконати додаткові дії, пов'язані з ініціалізацією полів:

```
constructor TFunc.Create(SF: String);
...
Begin
```

```
    Inherited Create;
    ...
End;
```

Функція

```
function Eval(ArgX, ArgY: Extended): Extended;
```

повертає значення виразу для вказаних значень аргументів. Оскільки в програмі Gran1 вирази бувають як з однією, так і з двома змінними, то дана функція отримує значення двох аргументів. Якщо вираз містить тільки одну змінну, значення другого аргументу ігнорується. Якщо у процесі обчислення значення виразу виникла помилка (ділення на нуль, добування квадратного кореня із від'ємного числа тощо), функція `Eval()` поверне значення константи `AllErr=1E+100` за допомогою обробки виключень. Це дозволяє аналізувати в подальшому такі помилки.

Функція

```
function Deriv(Arg: Extended): Extended;
```

повертає значення похідної для вказаного аргументу. Похідні від виразів, що містять дві змінні, у програмі Gran1 не розглядаються, тому аргумент в цій функції тільки один.

Оскільки на основі виразу будуть визначатися різноманітні функції, для яких потрібно надалі будувати графік, опис класу TFunc містить функції

```
function PointX(Arg:extended): Extended; virtual;
function PointY(Arg:extended): Extended; virtual;
```

що повертають відповідно значення X та Y координати точки графіка у декартовій системі координат.

Дані функції визначені віртуальними, тому що зміст їх буде змінюватися у деяких нащадках, наприклад для функціональної залежності, заданої параметрично, або у полярних координатах (використання поліморфізму).

Функція

```
function Simpson(AI, BI: Extended; IntegralType:
          Byte): Extended;
```

призначена для обчислення визначеного інтегралу методом Сімпсона для даної функціональної залежності.

Процедура

```
procedure ChangeFunc(SF: String; var Chng: Boolean);
virtual;
```

призначена для зміни значення виразу; `SF: String` – це новий вираз, `var Chng: Boolean` – повертає значення `True`, якщо зміна відбулася і `False` в протилежному випадку (зміна може не відбутися, якщо вираз `SF` є некоректним). Ця процедура також є віртуальною, тому що буде змінюватися у нащадках.

Об’єкти, з якими працює програма Gran1, наприклад такі, як явно задана функція Y=Y(X), параметрично задана функція, полярна функція визначаються не тільки виразом, але і відрізком задання. В свою чергу відрізок задання може задаватися виразами, що визначають значення лівого та правого кінців відрізку, тому потрібен клас для визначення кінця відрізку задання. Очевидно, що даний клас буде наслідником класу TFunc. Він описується наступним чином:

```
TRng=class(TFunc)
  private
  FValue: Extended;
  function GetValue: Extended;
  public
  constructor Create(SA: String);
  procedure ChangeFunc(SF: String; var Chng:
       Boolean); override;
  published
  property  Value: Extended read GetValue write
       FValue;
End;
```

У порівнянні з батьківським класом тут з'явилась властивість Value, описана наступним чином:

```
property  Value: Extended read GetValue write
FValue;
```

Значення цієї властивості визначається шляхом обчислення значення виразу, що задає кінець відрізка.

Для відображення графіків об'єктів програми Gran1 потрібен певний візуальний елемент, пов'язаний з декартовою системою координат. Для створення такого компоненту в якості батьківського був вибраний стандартний компонент `TImage`. Розглянемо опис цього компоненту та деяких допоміжних типів даних.

```
type
  MmType=record
    MinArg, MaxArg, MinFunc, MaxFunc: Extended
  End;
```

Цей тип даних призначений для зберігання мінімального та максимального значення по кожній із осей координат.

```
type
  CoordType=(ctDec,ctPolar);
```

Цей тип даних визначає, яка система координат зображується на екрані, декартова або полярна.

```
TMasshImage = class(TImage)
private
  { Private declarations }
  FOsiPoints: Boolean;
  FGraphPoints: Integer;
  FMasshAuto: Boolean;
  FMasshUpdate: TNotifyEvent;
  FCoord: CoordType;
```

```
    FXBasic:Extended;
    FYBasic:Extended;
    FIsMarkBuild:Boolean;
    FgrTextColor,FgrBackColor,FgrMouseColor,
    FgrMarkColor,FgrAxeColor,FgrAddColor: TColor;
    FgrXName,FgrYName: String;
    FgrLabelFont,FgrAxeFont: TFont;
    FgrAdd2Color:TColor;
  protected
    { Protected declarations }
  public
    { Public declarations }
    GraphRang: MmType;
    OldGraphRang: MmType;
    NumOfPoints: Integer;
    ...
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function XWin(X: Extended): Integer;
    function YWin(Y: Extended): Integer;
    function WinX(X: Integer): Extended;
    function WinY(Y: Integer): Extended;
    function WinR(X,Y: Integer): Extended;
    function WinF(X,Y: Integer): Extended;
    procedure ChangeGraphOps;
    procedure SetGraphRang(G: MmType);
    procedure UpToDate;
    procedure SetCoord(C: CoordType);
    procedure SetMasshAuto(M: Boolean);
  published
    { Published declarations }
    property OsiPoints: Boolean read FOsiPoints write
         FOsiPoints;
```

```
property GraphPoints: Integer read FGraphPoints
     write FGraphPoints;
property MasshAuto: Boolean read FMasshAuto write
     SetMasshAuto;
property OnMasshUpdate: TNotifyEvent read
     FMasshUpdate write FMasshUpdate;
property Coord: CoordType read FCoord write
     SetCoord default ctDec;
property XBasic: Extended read FXBasic write
     FXBasic;
property YBasic: Extended read FYBasic write
     FYBasic;
property IsMarkBuild: Boolean read FIsMarkBuild
     write FISMarkBuild;
property grTextColor: TColor read FgrTextColor
     write FgrTextColor;
property grBackColor: TColor read FgrBackColor
     write FgrBackColor;
property grMarkColor: TColor read FgrMarkColor
     write FgrMarkColor;
property grMouseColor: TColor read FgrMouseColor
     write FgrMouseColor;
property grAxeColor: TColor read FgrAxeColor write
     FgrAxeColor;
property grAddColor: TColor read FgrAddColor write
     FgrAddColor;
property grXName: String read FgrXName write
     FgrXName;
property grYName: String read FgrYName write
     FgrYName;
property grLabelFont: TFont read FgrLabelFont write
     FgrLabelFont;
property grAxeFont: TFont read FgrAxeFont write
     FgrAxeFont;
```

```
    property grAdd2Color: TColor read FgrAdd2Color
        write FgrAdd2Color;
End;
```

У цьому класі описано значну кількість властивостей, які відповідають за зовнішній вигляд системи координат і графіків, наприклад властивість

```
property Coord: CoordType read FCoord write SetCoord
    default ctDec;
```

визначає, яку систему координат показувати при побудові графіків - декартову чи полярну, за замовчуванням визначена декартова система координат.

Властивість

```
property grAxeColor: TColor read FgrAxeColor write
    FgrAxeColor;
```

визначає колір осей координат.

Описувати призначення інших властивостей не будемо, оскільки це досить легко визначити із їх назв. Зауважимо, що такий опис властивостей дозволяє встановити їх початкові значення у візуальному редакторі Lazarus.

Початок системи координат компоненти `TImage` знаходиться у лівому верхньому кутку області зображення, а координатами для цієї області зображення є цілі невід'ємні числа. В свою чергу об'єкти програми Gran1 оперують дійсними значеннями координат, а центр координат може бути де завгодно, як всередині області зображення, так і за її межами. Тому потрібно передбачити методи для перетворення координат компоненти TImage (екранних координат) в координати об'єктів Gran1 (математичні координати) і навпаки.

Функції

```
function XWin(X: Extended): Integer;
function YWin(Y: Extended): Integer;
```

повертають відповідно екранні `X` та `Y` для математичних `X` та `Y` координат.

Для прикладу наведемо код методу XWin:

```
function TMasshImage.XWin(X: Extended): Integer;
  var t:extended;
Begin
  with GraphRang do
  t:=(X-MinArg)*Width/(MaxArg-MinArg);
  if abs(t)<30000 then Xwin:=Round(t) else
       Xwin:=Sign(t)*30000
End;
```

Функції

```
function WinX(X: Integer): Extended;
function WinY(Y: Integer): Extended;
```

виконують зворотні перетворення. Їх можна, наприклад, використати, щоб відображати математичні координати при русі мишки над компонентою, створеною на основі класу TMasshImage, оскільки координати мишки у стандартних візуальних компонентах є екранними.

Для прикладу наведемо код методу WinX:

```
function TMasshImage.WinX(X: Integer): Extended;
Begin
  with GraphRang do
  WinX:=(maxarg-minarg)*x/Width+minarg;
End;
```

Програма Gran1 дозволяє будувати графіки функцій у полярних координатах, але, оскільки компонента типу TMasshImage пов'язана з декартовими координатами, побудова графіків таких функцій відбувається шляхом перетворення полярних координат у декартові. В свою чергу існує потреба відображати полярні координати при русі мишки над компонентою, створеною на основі класу TMasshImage, в цьому випадку можна скористатися функціями

```
function WinR(X,Y: Integer): Extended;
```

```
function WinF(X,Y: Integer): Extended;
```

які за екранними координатами повертають полярні координати ρ та φ відповідно.

Для прикладу наведемо код методу `WinR`:

```
function TMasshImage.WinR(X,Y: Integer): Extended;
  var xrf,yrf: Extended;
Begin
  with GraphRang do
  Begin
    xrf:=(maxarg-minarg)*X/Width+minarg;
    yrf:=(maxfunc-minfunc)*(Height-Y)/Height+minfunc;
  End;
  WinR:=Sqrt(Sqr(xrf)+Sqr(yrf));
End;
```

Для побудови графіків потрібно знати мінімальне та максимальне значення по кожній з осей (масштаб). Ці значення зберігаються в полі `GraphRang` типу `MmType`. Крім того, ці значення можуть бути визначені користувачем, або визначатися автоматично за списком об'єктів програми Gran1, відібраних користувачем для побудови їх графіків. Вид масштабування зберігається у властивості

```
FMasshAuto: Boolean;
```

якщо її значення `True`, масштабування автоматичне, в протилежному випадку масштаб визначається користувачем. Метод

```
procedure SetGraphRang(G: MmType);
```

дозволяє встановити новий масштаб, якщо масштаб визначається користувачем.

Метод

```
procedure UpToDate;
```

призначений для побудови графіків. Оскільки компонент типу `TMasshImage` “не знає” графіки яких об’єктів програми Gran1 треба будувати, цей метод генерує подію

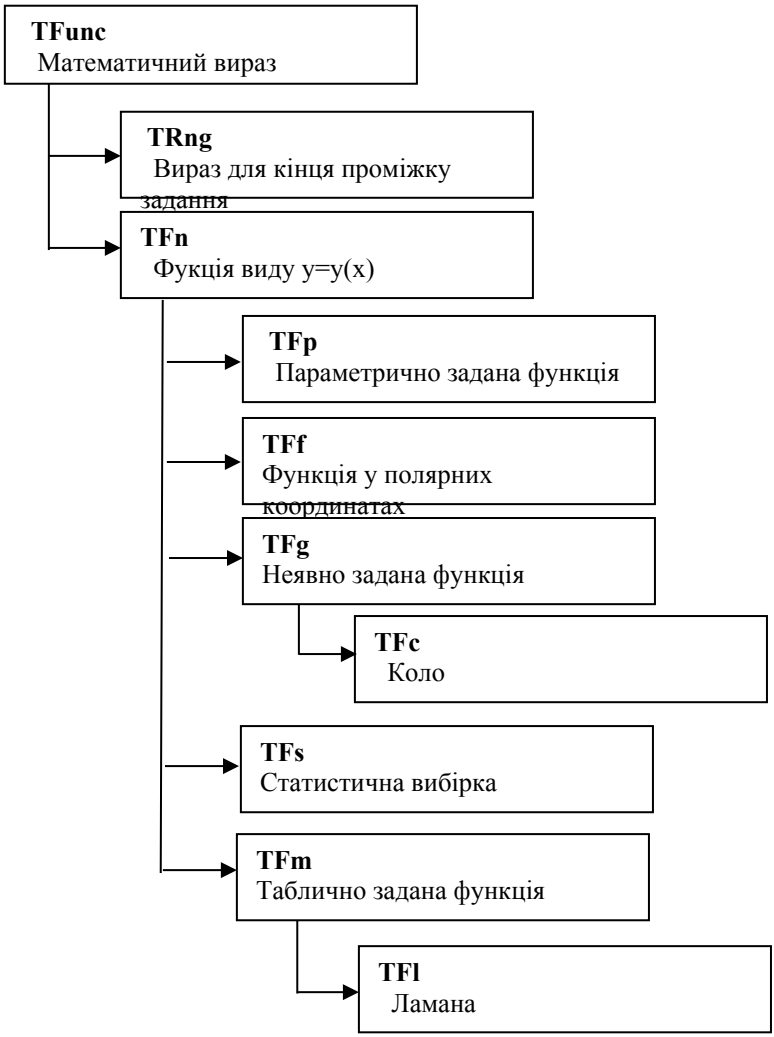
```
FMasshUpdate: TNotifyEvent;
```

яка буде оброблена власником компоненти типу `TMasshImage`. Наведемо код даного методу

```
procedure TMasshImage.UpToDate;
  var Save_Cursor:TCursor;
Begin
  Save_Cursor:=Screen.Cursor;
  Screen.Cursor:=crHourglass;
  try
  try
    OnMasshUpdate(Self);
  finally
    Screen.Cursor:=Save_Cursor;
  End;
  except
  End;
End;
```

У даному методі використана обробка виключних ситуацій, наприклад у процесі побудови графіків курсор миші приймає форму пісочного годинника і гарантується, що після закінчення побудови він набуде тієї форми, якої був до початку побудови.

Розглянемо тепер, як здійснюється опис і реалізація основних об’єктів програми Gran1, таких як функція Y=Y(X), параметрично задана функція, функція у полярних координатах, неявно задана функція тощо. Ці об’єкти утворюють ієрархію, на вершині якої знаходиться математичний вираз.



Розглянемо клас, що описує функцію `Y=Y(X)` у програмі Gran1.

```
TFn=class(TFunc)
  FArg:Char;
  Mm: MmType;
  A, B: TRng;
  FMark: Boolean;
```

```
  FColor: TColor;
  FGraph: Boolean;
  GBuild: Boolean;
  BGraphPoints: Integer;
  BuildByPoints: Boolean;
  WLine: Integer;
  constructor Create(SFy,SA,SB: String; GColor:
       TColor);
  constructor Load(f: TIniFile; Sec: String);
  procedure Save(f: TIniFile; Sec: String); virtual;
  destructor Destroy; override;
  procedure MmF; virtual;
  function GetType: FType; virtual;
  function GetTypeName: String; virtual;
  function GetName: String; virtual;
  function GetNameP: String; virtual;
  function MarkPossible: Boolean; virtual;
  function IntPossible: Boolean; virtual;
  procedure GraphBuild(I: TMasshImage); virtual;
  procedure Change(var Chng: Boolean); virtual;
  procedure Info(M: TMemo); virtual;
  procedure Nerav(AA: Extended; Sgn: Byte; I:
       TMasshImage; N: TNerDlg);
  procedure IntImage(AI, BI: Extended; IntegralType:
       Byte; II: TMasshImage); virtual;
  function GetColor: TColor;
  function LineLong(AI, BI: Extended): Extended;
  procedure LLongImage(AI, BI: Extended; II:
       TMasshImage);
  function GetHead: String; virtual;
  function Spoverh(ai,bi:extended;oss:byte):extended;
  procedure ScanParams; override;
End;
```

Цей клас TFn є нащадком класу TFunc, тому наслідує всі його властивості. Розглянемо тепер, які нові властивості з'явилися у цьому класі.

Поле FArg: Char містить позначення аргументу. у даному класі це поле приймає значення 'X'.

Поля A, B: TRng; визначають лівий та правий кінці відрізка задання функції. Поле Mm: MmType; містить найбільше та найменше значення по осях OX та OY. Значення Mm обчислюються кожного разу після зміни виразу функції або зміни відрізку задання за допомогою методу MmF, в якому організується цикл від найменшого до найбільшого значення відрізку задання. На кожному кроці цього циклу за допомогою методів PointX() та PointY() із класу батьківського класу TFunc обчислюються значення X та Y координат відповідної точки у декартовій системі координат і знаходяться найменше і найбільше із цих значень:

```
procedure TFn.MmF;
...
Begin
...
  t:=A.Value;
  while t<=B.Value do
    Begin
      x:=PointX(t);
      y:=PointY(t);
      if (x<Mm.MinArg) and (x<AllErr) then
            Mm.MinArg:=x;
      if (x>Mm.MaxArg) and (x<AllErr) then
            Mm.MaxArg:=x;
      if (y<Mm.MinFunc) and (y<AllErr) then
            Mm.MinFunc:=y;
      if (y>Mm.MaxFunc) and (y<AllErr) then
            Mm.MaxFunc:=y;
```

```
        t:=t+Step
      End;
...
End;
```

Оскільки об’єкти програми Gran1 можна записувати у файл та зчитувати з файлу, крім звичайного конструктора

```
constructor Create(SFy,SA,SB: String; GColor:
TColor);
```

що створює новий об’єкт класу TFn за переданими виразами для функції, кінців відрізка задання та кольору графіка, у класі TFn передбачено конструктор

```
constructor Load(f: TIniFile; Sec: String);
```

який створює новий об’єкт на основі даних із вказаного файлу.

Для збереження даних про об’єкти програми Gran1 використовуються текстові файли спеціального виду – так звані ini-файли. Ці файли мають наступний вигляд

```
[секція 1]
параметр1.1=значення1.1
параметр1.2=значення1.2
параметр1.3=значення1.3
параметр1.4=значення1.4
...
[секція 2 ]
параметр2.1=значення2.1
параметр2.2=значення2.2
параметр2.3=значення2.3
параметр2.4=значення2.4
...
```

Дані про кожен об’єкт програми записується у окрему секцію і відповідно зчитується з неї:

```
constructor TFn.Load(f: TIniFile; Sec: String);
var SFy, SA, SB: String;
    Color: TColor;
Begin
  SFy:=f.ReadString(Sec,'Func','');
  SA:=f.ReadString(Sec,'A','');
  SB:=f.ReadString(Sec,'B','');
  Color:=TColor(f.ReadInteger(Sec,'Color',0));
  Create(SFy,SA,SB,Color);
  ...
End;
```

Для запису даних про об'єкт використовується метод

```
procedure Save(f: TIniFile; Sec: String); virtual;
```

який записує у вказану секцію тип об'єкту та його параметри. Метод є віртуальним, тому що для різних класів у файл потрібно записувати різні параметри.

Розглянемо тепер метод

```
procedure GraphBuild(I: TMasshImage); virtual;
```

призначений для побудови графіка функції у вікні візуального об'єкту `I: TMasshImage`, що є параметром цього методу. Метод описано віртуальним, тому що у нащадках він може змінюватися (наприклад, він повинен змінитися у нащадку, що описує неявно задану функцію, оскільки алгоритми побудови графіка для функцій, явно і неявно заданих, практично не мають нічого спільного).

Реалізувати цей метод можна наступним чином:

організується цикл від найменшого до найбільшого значення відрізку задання, на кожному кроці цього циклу за допомогою методів `PointX()` та `PointY()` із класу батьківського класу `TFunc` обчислюються значення `X` та `Y` координат відповідної точки у декартовій системі координат і малюється лінія від попередньої точки до поточної:

```
...
t:=A.Value;
 while t<=B.Value do
   Begin
      x:=PointX(t);
      y:=PointY(t);
      I.Picture.BitMap.Canvas.MoveTo(Xwin(x1),
            Ywin(y1));
      I.Picture.BitMap.Canvas.LineTo(Xwin(x2),
            Ywin(y2));
      t:=t+step;
   End;
...
```

Зрозуміло, що наведений алгоритм є дуже схематичним, оскільки тут, наприклад, не передбачена ситуація виникнення розриву. Реальний алгоритм значно складніший.

У модулі, де описано класи функцій, є тип

```
type
  FType=(Normal, Param, Polar, TwoArg, Polinom, Stat,
       Lom, Circ);
```

для визначення типу функції:

`Normal` – функція y=y(x);

`Param` – параметрично задана функція;

`Polar` – функція у полярних координатах;

`TwoArg` – неявно задана функція;

`Polinom` – таблично задана функція, що апроксимується поліномом;

`Stat` – статистична вибірка;

`Lom` – ламана;

`Circ` – коло.

Функція `GetType` повертає тип функції.

Поле `BGraphPoints: Integer`; містить кількість точок, за якими будується графік функції.

Розглянемо тепер, як описані класи для інших об'єктів програми Gran1. Клас для функцій у полярних координатах описується, як нащадок класу функцій Y=Y(X).

```
TFf=class(TFn)
   constructor Create(SFy,SA,SB: String; GColor:
       TColor);
   constructor Load(f: TIniFile; Sec: String);
   function PointX(Arg:extended):extended;override;
   function PointY(Arg:extended):extended;override;
   function IntPossible:Boolean;override;
   function GetType:FType;override;
   function GetTypeName: String; override;
   function GetHead: String; override;
End;
```

Конструктор `Create(SFy,SA,SB: String; GColor: TColor);` має такий же заголовок, як і конструктор предка, але у ньому слід відобразити той факт, що змінюється позначення аргументу з 'X' на 'F', тому його код виглядатиме так:

```
constructor TFf.Create(SFy, SA, SB: String; Gcolor:
       TColor);
Begin
  inherited Create(SFy, SA, SB, Gcolor);
  FArg:='F'
End;
```

Такі ж зміни матиме і конструктор `Load()`:

```
constructor TFf.Load(f: TIniFile; Sec: String);
Begin
  inherited Load(f,Sec);
  FArg:='F'
End;
```

Для того, щоб у декартових координатах правильно будувати графік функції, заданої у полярних координатах, потрібно переписати також методи `PointX()` та `PointY()`:

```
function TFf.PointX(Arg:extended):extended;
Begin
  PointX:=Eval(Arg,0)*Cos(Arg)
End;

function TFf.PointY(Arg:extended):extended;
Begin
  PointY:=Eval(Arg,0)*Sin(Arg)
End;
```

Після такого перевизначення батьківський метод побудови графіка може залишатися незмінним, як незмінним може залишатися і метод `MmF()` обчислення найбільшого і найменшого значення по осях OX та OY. Це є ще одним прикладом використання поліморфізму.

Клас для параметрично заданих функцій також описується, як нащадок класу функцій Y=Y(X).

```
TFp=class(TFn)
  Fx: TFunc;
  constructor Create(SFy, SFx, SA, SB: String;
       GColor: TColor);
  constructor Load(f: TIniFile; Sec: String);
  procedure Save(f: TIniFile; Sec: String); override;
  destructor Destroy; override;
  function PointX(Arg: Extended): Extended; override;
  function PointY(Arg: Extended): Extended; override;
  function GetType: FType; override;
  function GetTypeName: String; override;
  function GetName: String; override;
  function GetNameP: String; override;
  function IntPossible: Boolean; override;
```

```
    procedure Change(var Chng:Boolean); override;
    function GetHead: String; override;
    procedure ScanParams; override;
    function ParamExist: Boolean; override;
  End;
```

Оскільки параметрично задана функція описується вже не одним виразом, як функція Y=Y(X), а двома Y=Y(T), X=X(T), у класі `TFp` потрібно додати нове поле `Fx: TFunc`, що міститиме вираз для X=X(T). Відповідні зміни потрібно внести і в конструктори:

```
constructor TFp.Create(SFy,SFx,SA,SB: String; GColor:
     TColor);
Begin
  inherited Create(SFy,SA,SB,GColor);
  Fx:=TFunc.Create(SFx);
  FArg:='T'
End;

constructor TFp.Load(f: TIniFile; Sec: String);
var SFx: String;
Begin
  inherited Load(f,Sec);
  SFx:=f.ReadString(Sec,'FuncX','');
  Fx:=TFunc.Create(SFx);
  FArg:='T';
End;
```

а також у метод, що відповідає за збереження об’єкту у файлі:

```
procedure TFp.Save(f: TIniFile; Sec: String);
Begin
  inherited Save(f,Sec);
  f.WriteString(Sec,'FuncX',Fx.GetTx);
End;
```

Потребує змін і деструктор, щоб звільнити пам'ять, виділену під поле Fx:

```
destructor TFp.Destroy;
Begin
  Fx.Free;
  inherited Destroy;
End;
```

Також потрібно внести зміни у метод для обчислення X координати, оскілки X-координата є результатом обчислення виразу X=X(T):

```
function TFp.PointX(Arg:extended):extended;
Begin
  PointX:=Fx.Eval(Arg,0)
End;
```

І нарешті розглянемо клас для неявно заданих функцій, тобто функцій виду G(X,Y)=0:

```
TFg=class(TFn)
  FArgY: Char;
  YA,YB: TRng;
  NameGxy: String[5];
  PrecisionGraph: Integer;
  constructor Create(SFy, SAx, SBx, SAy, SBy:
     String; GColor: TColor);
  constructor Load(f: TIniFile; Sec: String);
  procedure Save(f: TIniFile; Sec: String);
     override;
  destructor Destroy; override;
  function GetType: Ftype; override;
  function GetTypeName: String; override;
  function IntPossible: Boolean; override;
  procedure GraphBuild(I: TMasshImage); override;
  procedure Info(M: TMemo); override;
  procedure MmF(II: TMasshImage); override;
  procedure Change(var Chng: Boolean); override;
```

```
    function GetHead: String; override;
    procedure ScanParams; override;
  End;
```

Оскільки вираз неявно заданої функції має два аргументи, така функція розглядається вже не на відрізку задання, а на прямокутнику задання. Границі цього прямокутнику по осі OX наслідуються від батька – класу `TFn`, а по осі OY описуються у полях `YA,YB: TRng;` також потрібно додати поле `FArgY: Char;` для зберігання назви другого аргументу (‘Y’).

Повинні змінитися і конструктори:

```
constructor TFg.Create(SFy,SAx,SBx,SAy,SBy: String;
       GColor: TColor);
Begin
  inherited Create(SFy,SAx,SBx,GColor);
  YA:=TRng.Create(SAy);
  YB:=TRng.Create(SBy);
  FArg:='X';
  FArgY:='Y';
  PrecisionGraph:=6;
End;

constructor TFg.Load(f: TIniFile; Sec: String);
var SFy, SA, SB, SYA, SYB: String;
    Color: TColor;
Begin
  inherited Load(f,Sec);
  FArg:='X';
  FArgY:='Y';
  SYA:=f.ReadString(Sec,'YA','');
  SYB:=f.ReadString(Sec,'YB','');
  YA:=TRng.Create(SYA);
  YB:=TRng.Create(SYB);
```

```
        PrecisionGraph:=f.ReadInteger(Sec,'PrecisionGr
        aph',6);
End;
```

метод збереження даних у файлі:

```
procedure TFg.Save(f: TIniFile; Sec: String);
Begin
  inherited Save(f,Sec);
  f.WriteString(Sec,'YA',YA.GetTx);
  f.WriteString(Sec,'YB',YB.GetTx);

        f.WriteInteger(Sec,'PrecisionGraph',PrecisionG
        raph);
End;
```

та деструктор:

```
destructor TFg.Destroy;
Begin
  YA.Free;
  YB.Free;
  inherited Destroy;
End;
```

Як вже зазначалося вище, побудова графіка неявно заданої функції докорінно відрізняється від побудови графіків всіх функцій, що були розглянуті раніше, тому метод `GraphBuild()` повинен бути повністю переписаний. Алгоритм побудови графіка неявно заданої функції є неочевидним і досить складним, тому наводити його ми не будемо. Ідея полягає у пошуку на прямокутнику задання точок, де вираз G(X,Y) змінює знак.

Значна кількість операцій програми Gran1, наприклад побудова графіків, інтегрування, розв’язування нерівностей може виконуватися над

групою об’єктів. Доцільно таку групу представити у вигляді списку, що базується на основі стандартного списку TList:

```
type
 TFuncs=class(TList)
  GRI:TMasshImage;
  ...
  constructor Create;
  procedure LineLongs;
  procedure Kasat;
  procedure Neravs;
  procedure LLong;
  procedure SParams;
  procedure SquareLs;
  procedure VSoxyLs(OpsType: Integer);
  procedure Pirson;
  procedure NormFunc;
  procedure SysNerav;
  procedure FgNerav(Z: Char);
  procedure DoFixParam;
  {procedure PodobLam(var Fl: TFl; var CR: Boolean);}
public
  { Public declarations }
 ...
End;
```

Поле GRI є посиланням на візуальний об’єкт, в якому будуються графіки. Це посилання необхідне, оскільки деякі операції, наприклад розв’язування нерівностей, виводять допоміжні побудови (у випадку розв’язування нерівностей це зображення кореня на осі OX). Також тут описано методи, які відповідають за виконання тієї чи іншої операції над об’єктами з цього списку, наприклад метод SysNerav відповідає за розв’язування системи нерівностей для об’єктів, що є у списку.

Інтерфейс програми Gran1 базується на так званому MDI-стандарті, коли в межах головного вікна програми існують декілька вікон для відображення відповідних даних. Так у вікні “Список об’єктів” зображуються створені користувачем об’єкти програми Gran1, причому ці об’єкти користувач може відмічати для виконання групових операцій. Вікно “Графік” призначене для відображення графіків об’єктів, а у вікно “Відповіді” записуються результати обчислень. Кожному вікну відповідає певний клас, що є нащадком класу `TForm`. Розглянемо спочатку особливості класу, що реалізує роботу зі списком об’єктів програми Gran1. Для створення такого класу з ім’ям `TObjList` доцільно скористатися візуальним конструктором і розмістити у вікні візуальні компоненти `ObjListBox: TCheckListBox;` для подання списку об’єктів, `ObjMemo: TMemo;` для виведення даних про активний об’єкт у списку об’єктів та `ObjComboBox: TComboBox;` для вибору поточного типу об’єктів (у програмі можна створити тільки об’єкт поточного типу). Також потрібно додати у опис посилання на візуальну компоненту, в якій будуть будуватися графіки: `GRI: TMasshImage;`

```
type
 TObjList = class(TForm)
   ObjComboBox: TComboBox;
   ObjListBox: TCheckListBox;
   ObjMemo: TMemo;
   ...
   procedure FormCreate(Sender: TObject);
   procedure FormClose(Sender: TObject; var Action:
       TCloseAction);
   procedure NewObjPopupClick(Sender: TObject);
   procedure DelObjPopupClick(Sender: TObject);
   procedure ChngObjPopupClick(Sender: TObject);
```

```
  procedure ObjListBoxDrawItem(Control: TWinControl;
      Index: Integer; Rect: TRect; State:
      TOwnerDrawState);
  procedure FormDestroy(Sender: TObject);
  procedure ObjComboBoxChange(Sender: TObject);
  procedure SelAllPopupClick(Sender: TObject);
  procedure DeSelAllPopupClick(Sender: TObject);
  procedure ObjListBoxClick(Sender: TObject);
  procedure ObjListBoxKeyDown(Sender: TObject; var
      Key: Word; Shift: TShiftState);
  procedure ObjLPopupMenuPopup(Sender: TObject);
  procedure ObjListBoxClickCheck(Sender: TObject);
  procedure DelLastObjPopUpClick(Sender: TObject);
  procedure DelAllObjPopupClick(Sender: TObject);
  procedure FormActivate(Sender: TObject);
  procedure FormResize(Sender: TObject);
  procedure DelActiveObjPopUpClick(Sender: TObject);
  ...
private
  { Private declarations }
...
public
  { Public declarations }
  GRI: TMasshImage;
  GlobFType:FType;
  Funcs:TFuncs;
  FuncsG:TFuncs;
  NeedSave: Boolean;
  procedure
      MakeFuncsList(Funcs:TFuncs;OLSel:Boolean);
  procedure SetMasshtab;
  procedure ReadFuncs(FileName: String);
  procedure SaveFuncs(FileName: String);
  procedure SetObjMemo;
```

```
    procedure MmFAll;
  End;
```

У програмі Gran1 участь у побудові графіків беруть тільки ті об'єкти, у властивостях яких це визначено, а у різноманітних операціях, таких як інтегрування, розв'язування нерівностей тощо беруть участь ті об'єкти, які відмічено у візуальному списку об'єктів. Таким чином, ці дві множини об'єктів можуть не співпадати. Тому у класі `TObjList` визначено два списки об'єктів: список `FuncsG: TFuncs;` міститиме посилання на об'єкти, графік яких треба будувати, а список `Funcs: TFuncs;` міститиме посилання на об'єкти, які відмічено у візуальному списку об'єктів.

Для створення нового об'єкту викликається метод `NewObjPopupClick(Sender: TObject)`, який, наприклад, можна прив'язати до команди контекстного меню:

```
procedure TObjList.NewObjPopupClick(Sender: TObject);
  var Fn:TFn;
      C:Boolean;
Begin
  case GlobFType of
    Normal: Fn:=TFn.Create('X', '-5', '5',
          GlobColors.NextFColor);
    Polar: Fn:=TFf.Create('F', '0', '2*Pi',
          GlobColors.NextFColor);
    Param: Fn:=TFp.Create('T', 'T', '-5', '5',
          GlobColors.NextFColor);
    TwoArg: Fn:=TFg.Create('X', '-5', '5', '-5', '5',
          GlobColors.NextFColor);
    Polinom:Fn:=TFm.Create(GlobColors.NextFColor);
    Lom:    Fn:=TFl.Create(GlobColors.NextFColor);
    Stat:   Fn:=TFs.Create(GlobColors.NextFColor);
    Circ:   Fn:=TFc.Create('X*X+Y*Y-1', '-1', '1', '-
          1', '1', GlobColors.NextFColor);
```

```
    End;
    Fn.Change(C);
    if C then
      Begin
        NeedSave:=True;
        Fn.MmF(GRI);
        ObjListBox.Items.AddObject(Fn.GetName, Fn);
        ObjListBox.Checked[ObjListBox.Items.Count-
           1]:=True;
        ObjListBox.ItemIndex:=ObjListBox.Items.Count-1;
        ...
      end
    else
      Fn.Free;
  End;
```

У цьому методі створюється об'єкт програми Gran1, тип якого визначається згідно значенню змінної `GlobFType`, а значення цієї змінної залежить від вибору у списку типів ObjComboBox. Після створення об'єкту викликається метод зміни його параметрів (виразу функції, виразів кінців відрізка задання тощо `Fn.Change(C);`), вказується, що список об'єктів змінився і потребує запису у файл при виході із програми (`NeedSave:=True;`), обчислюються мінімальні і максимальні значення по осях для автоматичного масштабування (`Fn.MmF(GRI)`); створений об'єкт додається до списку об'єктів (`ObjListBox.Items.AddObject(Fn.GetName, Fn);`), відмічається

```
(ObjListBox.Checked[ObjListBox.Items.Count-1]:=True;
    ObjListBox.ItemIndex:=ObjListBox.Items.Count-
    1;).
```

Метод

```
procedure TObjList.MakeFuncsList(Funcs:TFuncs; OLSel:
    Boolean);
```

формує список об'єктів програми Gran1 для виконання тієї чи іншої операції; якщо параметр цього методу `OLSel` має значення `True`, у список потраплять всі відмічені у візульному списку об'єкти, якщо ж цей параметр має значення `False`, до списку потрапить тільки поточний об'єкт з візуального списку.

Метод

```
procedure TObjList.MakeFuncsListG(FuncsG:TFuncs);
```

призначений для формування списку об'єктів, графік яких потрібно будувати:

```
procedure TObjList.MakeFuncsListG(FuncsG:TFuncs);
  var j: Integer;
Begin
  FuncsG.Clear;
  with ObjListBox do
  Begin
    if Items.Count=0 then Exit;
    for j:=0 to Items.Count-1 do
      if TFn(Items.Objects[j]).GBuild then
        FuncsG.Add(Items.Objects[j]);
  End;
End;
```

Для зміни поточного об'єкту викликається метод `ChngObjPopupClick(Sender: TObject)`, який також можна прив'язати до команди контекстного меню:

```
procedure TObjList.ChngObjPopupClick(Sender:
     TObject);
  var C,S:Boolean;
      I: Integer;
Begin
```

```
  with ObjListBox do
   Begin
    if Items.Count>0 then
     Begin
      (Items.Objects[ItemIndex] as TFn).Change(C);
      if C then
       Begin
         NeedSave:=True;
        (Items.Objects[ItemIndex] as TFn).MmF(GRI);
         I:=ItemIndex;
         S:=Checked[I];
         Items[ItemIndex]:=(Items.Objects[ItemIndex]
            as TFn).GetName;
         Checked[I]:=True;
         Checked[I]:=S;
         FindParams;
       End;
      C:=C and (Items.Objects[ItemIndex] as
          TFn).FGraph;
      if C then
       Begin
         SetMasshtab;
         GRI.UpToDate;
       End;
     End;
    End;
   ObjListBoxClick(Sender);
 End;
```

Метод для запису списку об’єктів на диск у ini-файл виглядає наступним чином:

```
 procedure TObjList.SaveFuncs(FileName: String);
```

```
var f: TIniFile;
    tf: TextFile;
    i: Integer;
    Sec: String;
Begin
  try
  try
  AssignFile(tf,FileName);
  Rewrite(tf);
  Writeln(tf,'[Gran1]');
  for i:=1 to ObjListBox.Items.Count do
    Writeln(tf,'[Obj'+IntToStr(i)+']');
  CloseFile(tf);
  f:=TIniFile.Create(FileName);

       f.WriteInteger('Gran1','Num',ObjListBox.Items.
       Count);
  f.WriteString('Gran1','XName', GRI.grXName);
  f.WriteString('Gran1','YName', GRI.grYName);
  f.WriteFloat('Gran1','MinArg',
       GRI.GraphRang.MinArg);
  f.WriteFloat('Gran1','MaxArg',
       GRI.GraphRang.MaxArg);
  f.WriteFloat('Gran1','MinFunc',
       GRI.GraphRang.MinFunc);
  f.WriteFloat('Gran1','MaxFunc',
       GRI.GraphRang.MaxFunc);
  f.WriteInteger('Gran1','Coord',
       Integer(GRI.Coord));
  f.WriteBool('Gran1','MasshAuto',GRI.MasshAuto);
  for i:=0 to ObjListBox.Items.Count-1 do
  Begin
    Sec:='Obj'+IntToStr(i+1);
```

```
    (ObjListBox.Items.Objects[i] as TFn).Save(f,Sec);
    f.WriteBool(Sec,'Checked',ObjListBox.Checked[i]);
  End;
  NeedSave:=False;
  finally
  f.Free;
  End;
  except
    MyMessageDlg('Unable to save!',mtError,[mbOK],0);
  End;
End;
```

Для зчитування списку об’єктів з ini-файлу використовується метод

```
procedure TObjList.ReadFuncs(FileName: String);
var f: TIniFile;
    n,i: Integer;
    Sec,TypeName: String;
    G: Mmtype;
    CT: CoordType;
procedure CreateFunc;
var Fn:TFn;
    C: Boolean;
Begin
  Sec:='Obj'+IntToStr(i);
  TypeName:=f.ReadString(Sec,'ObjType','');
  C:=f.ReadBool(Sec,'Checked',True);
  if TypeName='Normal' then
    Fn:=TFn.Load(f,Sec);
  if TypeName='Param' then
    Fn:=TFp.Load(f,Sec);
  if TypeName='TwoArg' then
    Fn:=TFg.Load(f,Sec);
  if TypeName='Circ' then
```

```
    Fn:=TFc.Load(f,Sec);
  if TypeName='Polar' then
    Fn:=TFf.Load(f,Sec);
  if TypeName='Stat' then
    Fn:=TFs.Load(f,Sec);
  if TypeName='Lom' then
    Fn:=TFl.Load(f,Sec);
  if TypeName='Polinom' then
    Fn:=TFm.Load(f,Sec);
  Fn.MmF(GRI);
  ObjListBox.Items.AddObject(Fn.GetName, Fn);
  ObjListBox.Checked[ObjListBox.Items.Count-1]:=C;
  ObjListBox.ItemIndex:=ObjListBox.Items.Count-1;
  ObjListBoxClick(Self);
End;

Begin
  try
  try
  f:=TIniFile.Create(FileName);

  GRI.grXName:=f.ReadString('Gran1','XName','X');
  GRI.grYName:=f.ReadString('Gran1','YName','Y');
  G.MinArg:=f.ReadFloat('Gran1', 'MinArg',
       GRI.GraphRang.MinArg);
  G.MaxArg:=f.ReadFloat('Gran1', 'MaxArg',
       GRI.GraphRang.MaxArg);
  G.MinFunc:=f.ReadFloat('Gran1', 'MinFunc',
       GRI.GraphRang.MinFunc);
  G.MaxFunc:=f.ReadFloat('Gran1', 'MaxFunc',
       GRI.GraphRang.MaxFunc);
  CT:=CoordType(f.ReadInteger('Gran1', 'Coord',
       Integer(GRI.Coord)));
```

```
    GRI.MasshAuto:=f.ReadBool('Gran1', 'MasshAuto',
         GRI.MasshAuto);
    GRI.SetGraphRang(G);
    GRI.Coord:=CT;
    n:=f.ReadInteger('Gran1','Num',0);
    for i:=1 to n do
      CreateFunc;
    FindParams;
    finally
    f.Free;
    End;
    except
    End;
    GRI.UpToDate;
  End;
```

Клас для реалізації вікна “Графік” `TGr = class(TForm)` повинен містити візуальну компоненту, в якій і будуть будуватися графіки математичних об’єктів (`I: TMasshImage;`), а також візуальні компоненти для показу мінімальних і максимальних значень по кожній із осей і поточних математичних координат курсора миші:

```
  MasshBotStatus: TStatusBar;
   CoordStatus: TStatusBar;
   MasshTopStatus: TStatusBar;
```

Розглянемо тепер деякі важливі методі цього класу.

Метод

```
  procedure TGr.GraphsBuild;
```

призначений для побудови графіків відповідних математичних об’єктів. За побудову відповідає цикл

```
  with ObjList.FuncsG do
    Begin
      if Count=0 then Exit;
```

```
      for j:=0 to Count-1 do
       TFn(Items[j]).GraphBuild(I);
    End;
```

який будує графіки всіх об'єктів, які є у списку об'єктів, відібраних для побудови.

Метод

```
procedure TGr.Osi;
```

будує осі координат.

Метод

```
procedure TGr.IMasshUpdate(Sender: TObject);
```

викликає побудову осей координат, виводить мінімальні і максимальні значення по осях координат і викликає побудову графіків математичних об'єктів. Цей метод викликається кожного разу, коли генерується подія `FMasshUpdate: TNotifyEvent;`. нагадаємо, що цю подію можна згенерувати, викликавши метод `UpToDate()` візуальної компоненти `TMasshImage`, в якій і відбувається побудова.

```
procedure TGr.IMasshUpdate(Sender: TObject);
var j: Integer;
Begin
  if WindowState=wsMinimized then Exit;
  Osi;
  MasshTopStatus.SimpleText:='MinX='+
FloatToStrF(I.GraphRang.MinArg, ffGeneral,
     ngDigits ,ngPower)+'  MaxY='+
     FloatToStrF(I.GraphRang.MaxFunc, ffGeneral,
     ngDigits,ngPower);
  MasshBotStatus.SimpleText:='MinY='+
     FloatToStrF(I.GraphRang.MinFunc, ffGeneral,
     ngDigits, ngPower)+'
```

```
      MaxX='+FloatToStrF(I.GraphRang.MaxArg,
      ffGeneral, ngDigits ,ngPower);
    GraphsBuild;
...
End;
```

Метод

```
procedure TGr.FormResize(Sender: TObject);
```

обробляє ситуацію зміни розміру вікна “Графік”, оскільки в цьому випадку потрібно змінити і розмір масиву збереження зображення візуальної компоненти, в якій будуються графіки:

```
procedure TGr.FormResize(Sender: TObject);
Begin
  I.Picture.BitMap.Width:=I.Width;
  I.Picture.BitMap.Height:=I.Height;
  I.UpToDate;
End;
```

Щоб виводити поточні математичні координати при русі курсора миші, потрібно написати обробник події, що виникає при русі миші. У найпростішому варіанті цей обробник виглядатиме так:

```
procedure TGr.IMouseMove(Sender: TObject; Shift:
     TShiftState; X,  Y: Integer);
  var tx: Extended;
        ty: Extended;
 Begin
  case I.Coord of
    ctDec:
      Begin
       tx:=I.WinX(X);
       ty:=I.WinY(Y);
```

```
        CoordStatus.SimpleText:='X='+
             FloatToStrF(tx,ffGeneral, ngDigits,
             ngPower)+'  Y='+FloatToStrF(ty,
             ffGeneral, ngDigits, ngPower);
      End;
    ctPolar:
      Begin
        tx:=I.WinF(X,Y);
        ty:=I.WinR(X,Y);
        CoordStatus.SimpleText:='R='+
             FloatToStrF(ty,ffGeneral, ngDigits,
             ngPower)+'  F='+FloatToStrF(tx,
             ffGeneral, ngDigits, ngPower)+'
             ('+FloatToStrF(tx/pi*180,
             ffGeneral,ngDigits, ngPower)
             +chr(176)+')';
      End;
  End;
End;
```

У програмі Gran1 цей обробник значно складніший, оскільки він також реагує на виділення мишею ділянки візуальної компонети, в якій будуються графіки, та також на деякі інші події.

Клас для реалізації вікна “Відповіді” type `TAns = class(TForm)` являє собою звичайне вікно, в якому розміщено компонент AnsMemo: TMemo; для відображення відповідей, які отримуються в результаті виконнання тих чи інших операцій програми Gran1.

Всі згадані вище вікна об’єднуються уголовному вікні програми Gran1, якому відповідає клас `T__Main = class(TForm)`. Дане вікно містить головне меню, в якому є всі команди для роботи з програмою Gran1. Побудувати таке меню можна за допомогою стандартного візуального конструктора меню. Розглянемо деякі команди, реалізовані у класі `T__Main`.

Програма Gran1 зберігає свої установки у так званому .ini файлі, про який згадувалося вище. Це реалізовано у методі

```
procedure T__Main.SaveIni;
var f: TIniFile;
    tf: TextFile;
    s: String;
    n: Integer;
Begin
  s:=Application.ExeName;
  n:=Length(s);
  s[n-2]:='I';
  s[n-1]:='N';
  s[n]:='I';
   try
  try
  f:=TIniFile.Create(s);
  f.WriteInteger('MainWindow', 'Left', Left);
  f.WriteInteger('MainWindow', 'Top', Top);
  f.WriteInteger('MainWindow', 'ClientWidth',
       ClientWidth);
  f.WriteInteger('MainWindow', 'ClientHeight',
       ClientHeight);
  f.WriteInteger('MainWindow', 'WindowState',
       Integer(WindowState));
  f.WriteString('MainWindow', 'Language',
       ActualLanguage);
  f.WriteInteger('MainWindow', 'Digits',  ngDigits);
  f.WriteBool('MainWindow', 'AutoSizeWin',
       AutoWinItem.Checked);
  f.WriteInteger('Objects', 'Left', ObjList.Left);
  f.WriteInteger('Objects', 'Top', ObjList.Top);
  f.WriteInteger('Objects', 'Width', ObjList.Width);
```

```
f.WriteInteger('Objects', 'Height',
     ObjList.Height);
f.WriteInteger('Objects', 'ObjMemoHeight',
     ObjList.ObjMemo.Height);
f.WriteBool('Graph', 'SGVisible', Gr.SG.Visible);
f.WriteBool('Graph', 'MVisible',
     Gr.MasshTopStatus.Visible);
f.WriteInteger('Graph', 'SGHeight', Gr.SG.Height);
f.WriteInteger('Graph', 'Left', Gr.Left);
f.WriteInteger('Graph', 'Top', Gr.Top);
f.WriteInteger('Graph', 'Width', Gr.Width);
f.WriteInteger('Graph', 'Height', Gr.Height);
f.WriteInteger('Graph', 'TextColor',
     Gr.I.grTextColor);
f.WriteInteger('Graph', 'BackColor',
     Gr.I.grBackColor);
f.WriteInteger('Graph', 'MarkColor',
     Gr.I.grMarkColor);
f.WriteInteger('Graph', 'AxeColor',
     Gr.I.grAxeColor);
f.WriteInteger('Graph', 'AddColor',
     Gr.I.grAddColor);
f.WriteInteger('Graph', 'MouseColor',
     Gr.I.grMouseColor);
f.WriteString('Graph', 'XName',  gr.I.grXName);
f.WriteString('Graph', 'YName',  gr.I.grYName);
f.WriteInteger('Answers', 'Left', Ans.Left);
f.WriteInteger('Answers', 'Top', Ans.Top);
f.WriteInteger('Answers', 'Width', Ans.Width);
f.WriteInteger('Answers', 'Height', Ans.Height);
f.WriteString('Graph', 'grLabelFontName',
     gr.I.grLabelFont.Name);
f.WriteInteger('Graph', 'grLabelFontSize',
     gr.I.grLabelFont.Size);
```

```
  f.WriteString('Graph', 'grAxeFontName',
       gr.I.grAxeFont.Name);
  f.WriteInteger('Graph', 'grAxeFontSize',
       gr.I.grAxeFont.Size);
  finally
  f.Free;
  End;
  except
  End;
End;
```

Завантаження цих параметрів відбувається автоматично у процесі запуску програми Gran1 шляхом виклику методу

```
procedure T__Main.LoadIni;
var f: TIniFile;
    s, FN: String;
    n, i: Integer;
Begin
  s:=Application.ExeName;
  n:=Length(s);
  s[n-2]:='I';
  s[n-1]:='N';
  s[n]:='I';
  try
  try
  f:=TIniFile.Create(s);
  ActualLanguage:=f.ReadString('MainWindow',
       'Language', '"Українська');
  i:=LanguagesList.LangName.IndexOf(ActualLanguage);
  if i>=0 then
  Begin
```

```
  FN:=LanguagesList.LangFileName[i];
  LoadLangdata(Application, FN);
End;
Top:=f.ReadInteger('MainWindow', 'Top', 30);
Left:=f.ReadInteger('MainWindow', 'Left', 0);
ClientWidth:=f.ReadInteger('MainWindow',
     'ClientWidth', 795);
ClientHeight:=f.ReadInteger('MainWindow',
     'ClientHeight', 506);
ngDigits:=f.ReadInteger('MainWindow', 'Digits', 4);

     AutoWinItem.Checked:=f.ReadBool('MainWindow' ,
     'AutoSizeWin', False);

     WindowState:=TWindowState(f.ReadInteger('MainW
     indow', 'WindowState', Integer(wsNormal)));
ObjList.Top:=f.ReadInteger('Objects', 'Top', 0);
ObjList.Left:=f.ReadInteger('Objects', 'Left',
     440);
ObjList.Width:=f.ReadInteger('Objects', 'Width',
     326);
ObjList.Height:=f.ReadInteger('Objects', 'Height',
     333);
ObjList.ObjMemo.Height:=f.ReadInteger('Objects',
     'ObjMemoHeight', 100);
Ans.Top:=f.ReadInteger('Answers', 'Top', 333);
Ans.Left:=f.ReadInteger('Answers', 'Left', 440);
Ans.Width:=f.ReadInteger('Answers', 'Width', 326);
Ans.Height:=f.ReadInteger('Answers', 'Height',
     167);
Gr.Top:=f.ReadInteger('Graph', 'Top', 0);
Gr.Left:=f.ReadInteger('Graph', 'Left', 0);
Gr.Width:=f.ReadInteger('Graph', 'Width', 440);
Gr.Height:=f.ReadInteger('Graph', 'Height', 500);
```

```
Gr.I.grTextColor:=f.ReadInteger('Graph',
     'TextColor', clBlack);
Gr.I.grBackColor:=f.ReadInteger('Graph',
     'BackColor', clLtGray);
Gr.I.grMarkColor:=f.ReadInteger('Graph',
     'MarkColor', clGray);
Gr.I.grAxeColor:=f.ReadInteger('Graph', 'AxeColor',
     clBlack);
Gr.I.grAddColor:=f.ReadInteger('Graph', 'AddColor',
     clWhite);
Gr.I.grMouseColor:=f.ReadInteger('Graph',
     'MouseColor', clWhite);
Gr.I.grXName:=f.ReadString('Graph', 'XName', 'X');
Gr.I.grYName:=f.ReadString('Graph', 'YName', 'Y');

Gr.I.grLabelFont.Name:=f.ReadString('Graph',
     'grLabelFontName', 'System');
Gr.I.grLabelFont.Size:=f.ReadInteger('Graph',
     'grLabelFontSize', 10);
Gr.I.grAxeFont.Name:=f.ReadString('Graph',
     'grAxeFontName', 'System');
Gr.I.grAxeFont.Size:=f.ReadInteger('Graph',
     'grAxeFontSize', 10);
Gr.MasshTopStatus.Visible:=f.ReadBool('Graph',
     'MVisible', True);
Gr.MasshBotStatus.Visible:=f.ReadBool('Graph',
     'MVisible', True);
Gr.MasshBotStatus.Top:=Gr.ClientHeight-
     Gr.MasshBotStatus.Height;
Gr.SG.Visible:=False;
if not Gr.SG.Visible then Gr.Splitter1.Free;
Gr.SG.Height:=f.ReadInteger('Graph', 'SGHeight',
     44);
Gr.FormResize(Self);
```

```
    finally
    f.Free;
    End;
    except
    End;
  End;
```

Програма Gran1 дозволяє копіювати до буферу обміну зображення у вікні “Графік”, якщо активним є це вікно, або список виразів з вікна “Список об’єктів”, якщо активним є це вікно. Реалізовано це шляхом посилання відповідного повідомлення операційній системі у методі

```
  procedure T__Main.Copy1Click(Sender: TObject);
  Begin
    Screen.ActiveControl.Perform(WM_COPY,0,0);
    if Screen.ActiveForm is TGr then
      ClipBoard.Assign((Screen.ActiveForm as
         TGr).I.Picture.BitMap);
    if Screen.ActiveForm is TObjList then
      with (Screen.ActiveForm as TObjList).ObjListBox
         do
        ClipBoard.AsText:=Items.Strings[ItemIndex];
  End;
```

Зрозуміло, що нами представлено тільки кістяк програми Gran1. Поза розглядом залишилось багато інших методів, що реалізують функціональність програми. Проте вивчення даного кістяку дозволить студентам отримати певні знання щодо моделювання математичних об’єктів, пов’язаних з функціональними залежностями, взаємозв’язок між такими моделями, принципи і конкретні приклади побудови досить великої програми і застосувати отримані знання у подальшій педагогічній діяльності,

пов'язаній з викладанням основ програмування і об'єктно-орієнтованого програмування, а також у майбутніх наукових дослідженнях, можливо і для написання власних педагогічних програмних продуктів з області математики.

6.2. Завдання для студентів

Завдання 1.

Розробити клас TFunc:

```
TFunc=class(TObject)
  Tx,St: String;
  Arr: RealArray;
  public
    constructor Create(SF: String);
    function PointX(Arg:extended): Extended; virtual;
    function PointY(Arg:extended): Extended; virtual;
    function Simpson(AI, BI: Extended): Extended;
    procedure ChangeFunc(SF: String; var Chng:
       Boolean);virtual;
    function Eval(ArgX, ArgY: Extended): Extended;
End;
```

`Tx, St, Arr` – поля для збереження виразу. Конструктор `Create`, методи `Eval` (обчислення значення виразу) та `ChangeFunc` (зміна виразу функції) наведено в додатку.

Метод PointX повинен повертати значення x-координати у декартовій системі координат, метод `PointY` – значення y-координати у декартовій системі координат, метод `Simpson` – значення інтегралу від виразу на

відрізку AI,BI (застосувати будь-який чисельний метод знаходження інтегралу, наприклад метод парабол).

Клас розмістити у окремому модулі NGFunc, підключивши до нього модуль NGMath (наведено у додатку). Даний модуль містить процедури, що на основі рядку запису виразу будують певну структуру даних, яка використовується у методі Eval() для обчислення значення цього виразу.

Написати консольну програму, яка вводить значення виразу з клавіатури і обчислює значення інтегралу від цього виразу на відрізку, також введеному з клавіатури.

Завдання 2.

Розробити клас TRng:

```
TRng=class(TFunc)
  private
    FValue: Extended;
    function GetValue: Extended;
    public
    constructor Create(SA: String);
    procedure ChangeFunc(SF: String; var Chng:
       Boolean); override;
  published
    property  Value: Extended read GetValue write
       FValue;
  End;
```

Клас призначено для задання виразу, що визначає один з кінців відрізка задання майбутньої функції. Від класу NGFunc він відрізняється наявністю властивості Value – значення виразу, що задає кінець відрізку. Функція GetValue повинна обчислювати значення виразу, конструктор Create повинен викликати батьківський конструктор і обчислювати значення властивості Value; метод ChangeFunc теж повинен викликати

батьківський метод і у випадку вдалої зміни виразу обчислити значення властивості `Value`. клас розмістити у модулі `NGFuncs`.

Завдання 3.

Розробити клас `TMasshImage`, що описує візуальний компонент, у якому будуть будуватися графіки функцій:

```
TMasshImage = class(TImage)
  private
    { Private declarations }
    FMasshUpdate: TNotifyEvent;
  public
    { Public declarations }
    GraphRang: MmType;
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function XWin(X: Extended): Integer;
    function YWin(Y: Extended): Integer;
    function WinX(X: Integer): Extended;
    function WinY(Y: Integer): Extended;
    function WinR(X,Y: Integer): Extended;
    function WinF(X,Y: Integer): Extended;
    procedure SetGraphRang(G: MmType);
    procedure UpToDate;
  published
    { Published declarations }
    property OnMasshUpdate: TNotifyEvent read
          FMasshUpdate write FMasshUpdate;
End;
```

Призначення методів знайти у описі цього класу. Клас розмістити у модулі `MImage`. Передбачити реєстрацію цього класу як візуального компоненту на закладці `Samples` у средовищі програмування Lazarus.

Завдання 4.

Описати клас `TFn`, що визначає функцію Y=Y(X):

```
TFn=class(TFunc)
  FArg:Char;
  Mm: MmType;
  A, B: TRng;
  FColor: TColor;
  BGraphPoints: Integer;
  constructor Create(SFy,SA,SB: String; GColor:
       TColor);
  destructor Destroy; override;
  procedure MmF; virtual;
  function GetType: FType; virtual;
  procedure GraphBuild(I: TMasshImage); virtual;
  procedure Change(var Chng: Boolean); virtual;
  procedure Info(M: TMemo); virtual;
  function GetColor: TColor;
End;
```

У конструктор передаються вираз функції, вирази для кінців відрізка та колір графіка.

Метод `Change` повинен викликати діалогове вікно, у якому користувач повинен ввести нові вирази для функції та кінців відрізка, а також вибрати колір графіка. Введені користувачем вирази потрібно перевірити на правильність і якщо все гаразд, замінити попередні. Якщо така зміна відбулась, параметр `Chng` повинен повернути значення `True`.

Метод Info виводить у свій параметр типу `TMemo` дані про функцію – вираз функції, вирази кінців відрізка задання, мінімальні та максимальні значення аргументу та функції.

Функція `GetColor` повертає колір графіка. Призначення інших методів можна знайти у описі класу. Клас потрібно розістити у модулі `NGFuncs`.

Завдання 5.

Розробити програму для побудови графіка функції. Вікно програми повинно містити компонент типу `TLabel` для відображення імені функції, компонент типу `TMemo` для виведення даних про функцію, компонент типу `TMasshImage` для побудови в ньому графіка функції; кнопки “Змінити функцію” та “Побудувати графік”. Функція з початковим виразом ‘x’, відрізком задання ‘-5’, ‘5’ та червоним кольором графіка повинна бути створена автоматично при запуску програми.

Завдання 6.

Розробити клас TFp, що визначає функцію, задану параметрично:

```
TFp=class(TFn)
  Fx: TFunc;
  constructor Create(SFy, SFx, SA, SB: String;
       GColor: TColor);
  destructor Destroy; override;
  function PointX(Arg: Extended): Extended; override;
  function PointY(Arg: Extended): Extended; override;
  function GetType: FType; override;
  procedure Change(var Chng:Boolean); override;
  procedure Info(M: TMemo); override;
End;
```

У конструктор передаються вирази y=y(t) та x=x(t), вирази для кінців відрізка та колір графіка.

Метод Change повинен викликати діалогове вікно, у якому користувач повинен ввести нові вирази для y=y(t) і x=x(t) та кінців відрізка, а також вибрати колір графіка. Введені користувачем вирази потрібно перевірити на правильність і якщо все гаразд, замінити попередні. Якщо така зміна відбулась, параметр Chng повинен повернути значення True.

Метод Info виводить у свій параметр типу TMemo дані про функцію – вирази y=y(t) та x=x(t), вирази кінців відрізка задання, мінімальні та максимальні значення аргументу та функції.

Клас потрібно розістити у модулі NGFuncs.

Аналогічно до попереднього, розробити клас TFf для функції у полярних координатах:

```
TFf=class(TFn)
  constructor Create(SFy,SA,SB: String; GColor:
       TColor);
  function PointX(Arg:extended):extended;override;
  function PointY(Arg:extended):extended;override;
  function GetType:FType;override;
  procedure Info(M: TMemo); override;
  procedure Change(var Chng:Boolean); override;
End;
```

Клас потрібно розістити у модулі NGFuncs.

Завдання 7.

Вдосконалити програму для побудови графіка функції. У вікно програми потрібно додати компонент для вибору типу функції, що буде створюватися, та кнопку “Створити”, призначену для створення нової функції за вибраним користувачем типом функції. У обробнику події кнопки “Створити функцію” передбачити знищення попередньої функції.

7. Gran2D

7.1. Теоретичні відомості про структуру ПЗ Gran2D

Розгляд особливостей програмного коду будемо проводити у дещо спрощеному вигляді, з реалізацією невеликої кількості можливостей. Проте це дозволить побудути каркас програми, який в подальшому можна буде доповнити новими можливостями.

При створені ПЗ були визначені деякі окремі класи та декілька дерев класів, які реалізують певні об'єкти предметної області.

Зважаючи на специфіку ПЗ, основною необхідністю є візуалізація геометричних побудов, отже нам необхідно визначити клас, який буде містити характеристики нашої робочої області – прямокутної декартової системи координат (ПДСК). Таким клас буде `TGrPlace`, який буде базуватися на стандартному класі `TImage`.

```
type TGrPlace=class(TImage)
  Private
    FPutOsi: Boolean;
    FDrawGrid: Boolean;
    FYNameOsi: String;
    FXNameOsi: String;
    FColorOsi: TColor;
    FColorObl: TColor;
    Fxc: Integer;
    Fyc: Integer;
    FRolX: Integer;
    FRolY: Integer;
    FZoom: Extended;
    FAZoom: Integer;
    FTecDX, FTecDY: Extended;
    FlagDrawScene: Boolean;
```

```
  LastX, LastY: Double;
  AB: Array[1..5] Of Record
     ex, ey: Extended;
   End;
Public
  constructor Create(AOwner: TComponent); override;
  procedure ScreenToAbsZ(x, y: Integer; var xx, yy:
       Extended);
  procedure AbsToScreenZ(x, y: Extended; var xx,yy:
       Extended); overload;
  procedure AbsToScreenZ(x, y: Extended; var xx,
       yy:Integer); overload;
  procedure SetZoom(n: Extended);
  procedure DrawLine(x1,y1,x2,y2:Extended; Lt:
       Byte=0);
  procedure Draw;
  procedure GrMouseMove(Sender: TObject; Shift:
       TShiftState; X, Y: Integer);
  procedure GrMouseDown(Sender: TObject; Button:
       TMouseButton; Shift: TShiftState; X, Y:
       Integer);
  procedure GrMouseUp(Sender: TObject; Button:
       TMouseButton; Shift: TShiftState; X, Y:
       Integer);
  procedure Save(f: TIniFile; Sec: String);
  procedure Load(f: TIniFile; Sec: String);
  property ColorObl: TColor read FColorObl write
       FColorObl;
  property PutOsi  : Boolean read FPutOsi write
       FPutOsi;
  property ColorOsi: TColor read FColorOsi write
       FColorOsi;
  property DrawGrid: Boolean read FDrawGrid write
       FDrawGrid;
```

```
    property XNameOsi: String read FXNameOsi write
          FXNameOsi;
    property YNameOsi: String read FYNameOsi write
          FYNameOsi;
    property TecDX: Extended read FTecDX;
    property TecDY: Extended read FTecDY;
    property AZoom: Integer read FAZoom;
End;
```

Для більш гнучкого налагодження візуального відображення області в неї включені наступні властивості:

`ColorObl : TColor` – колір робочої області;

`PutOsi : Boolean` – необхідність відображення осей координат;

`ColorOsi: TColor` – колір осей координат;

`DrawGrid: Boolean` – необхідність відображення координатної сітки;

`XNameOsi: String` та `YNameOsi: String` – підписи на осях;

`FlagDrawScene: Boolean` – необхідність перемальовувати графічну область;

`LastX, LastY: Double` – поля, які використовуються при переміщенні системи координат.

Деякі з властивостей є властивостями лише для читання (в їх описі відсутня частина write), оскільки їх значення не можуть змінюватися напряму за межами даного класу. Так наприклад `AZoom: Integer` – відповідає кількості точок в 1 см., яка залежить від параметрів екрану; `TecDX: Extended` та `TecDY: Extended` – координати точки в ПДСК, над якою знаходиться курсор миші.

Серед методів визначено методи для переведення координат з ПДСК в координати робочої області та навпаки `ScreenToAbsZ, AbsToScreenZ`. Дані методи використовують деякі поля з області private: `FRolX:`

Integer, FRolY: Integer – координати центру ПДСК; Fxc: Integer, Fyc: Integer – середина графічної області; FAZoom: Integer – кількість точок в 1 см., FZoom: Extended – масштаб.

Серед полів визначено поле

```
AB: Array[1..5] Of Record
  ex,ey: Extended;
End;
```

яке буде містити координати кутів робочої області в ПДСК. Значення даного поля буде перераховуватися наступним чином

```
with AB[1] do ScreenToAbsZ(0,0, ex,ey);
with AB[2] do ScreenToAbsZ(Width,0, ex,ey);
with AB[3] do ScreenToAbsZ(Width,Height, ex,ey);
with AB[4] do ScreenToAbsZ(0,Height, ex,ey);
AB[5]:=AB[1];
```

де Width та Height – ширина та висота, відповідно, графічної області.

Процедура

```
procedure TGrPlace.ScreenToAbsZ(x, y: Integer; var
     xx, yy: Extended);
begin
  XX:=(X-Fxc-FRolX)/Fzoom/FAZoom;
  YY:=(Fyc-Y-FRolY)/Fzoom/FAZoom;
end;
```

перераховує екранні координати x, y в координати ПДСК; її можна використати, щоб відображати математичні координати при русі мишки над графічної компонентою.

Процедура

```
procedure TGrPlace.AbsToScreenZ(x, y: Extended; var
     xx, yy: Extended);
```

```
begin
  xx:=Fxc+FRolX+x*Fzoom*FAZoom;
  yy:=Fyc-(FRolY+y*Fzoom*FAZoom);
end;
```

виконує зворотне перетворення: для математичних x, y повертає екранні в дійсних чи цілих одиницях.

Метод

```
procedure SetZoom(n: Extended);
```

дозволяє змінювати масштаб на n одиниць.

Методи

```
procedure Save(f: TIniFile; Sec: String);
procedure Load(f: TIniFile; Sec: String);
```

призначені для збереження і завантаження параметрів графічної області в текстовий файли спеціального виду – так звані ini-файли.

Для прикладу

```
procedure TGrPlace.Save(f: TIniFile; Sec: String);
begin
  f.WriteInteger(Sec, 'ColorObl', FColorObl);
  f.WriteBool   (Sec, 'PutOsi', FPutOsi);
  f.WriteInteger(Sec, 'ColorOsi', FColorOsi);
  f.WriteBool   (Sec, 'DrawGrid', FDrawGrid);
  f.WriteString (Sec, 'XNameOsi', FXNameOsi);
  f.WriteString (Sec, 'YNameOsi', FYNameOsi);
  f.WriteFloat  (Sec, 'Zoom', FZoom);
  f.WriteInteger(Sec, 'RolX', FRolX);
  f.WriteInteger(Sec, 'RolY', FRolY);
end;
```

Для виконання побудови лінії за її типом (пряма – `Lt=0`, промінь – `Lt=1`, відрізок – `Lt=2`) описано метод:

```
procedure DrawLine(x1,y1,x2,y2:Extended; Lt: Byte=0);
```

реалізація якого має вигляд:

```
procedure TGrPlace.DrawLine(x1,y1,x2,y2:Extended; Lt:
     Byte=0);
Var ColBounds, i: integer;
    Bounds: Array[1..2] Of TPoint;
    xx, yy: Extended;

Procedure IncBounds;
Var tx,ty:Extended;
Begin
  inc(ColBounds);
  AbsToScreenZ(xx,yy, tx,ty);
  with Bounds[ColBounds] Do
   Begin
     x:=round(tx);
     y:=round(ty);
   End;
End;

Begin
  With Canvas Do
   Begin
    //пошук кінців лінії
    ColBounds:=0;
    For i:=1 To 4 Do
     If ColBounds<2 Then
      Begin
        If X_Line_Line(x1,y1,x2,y2, AB[i].ex,
            AB[i].ey, AB[i+1].ex, AB[i+1].ey,xx,yy)
            Then
         If (odd(i) And
            (xx<=max(AB[i].ex,AB[i+1].ex)) And
            (xx>=min(AB[i].ex,AB[i+1].ex)) ) Or  (Not
```

```
              odd(i) And (yy< max(AB[i].ey,AB[i+1].ey))
              And (yy>min(AB[i].ey,AB[i+1].ey)) )
            Then IncBounds;
        End;
      . . .
       If colBounds>=2 Then
        Begin
          With Bounds[1] Do Moveto(x,y);
          With Bounds[2] Do lineto(x,y)
        End;
    End;
End;
```

Пошук кінців лінії використовується для визначення координат перетину цієї лінії з межами робочої області. В методі опущено встановлення кінців у випадку побудови відрізка або променя.

Підпрограма `X_Line_Line(x11,y11, x12,y12, x21,y21, x22,y22: Extended; var x,y: Extended): Boolean`, яка використовується при пошуку кінців лінії, являє собою функцію знаходження точки перетину двох прямих, кожна з яких задана координатами двох точок. У випадку, якщо прямі не перетинаються, тобто є паралельні, вона повертає хибу.

Реалізація даної функції має вигляд:

```
function X_Line_Line(x11,y11, x12,y12, x21,y21,
     x22,y22: Extended; var x,y: Extended): Boolean;
Var x1,y1,x2,y2,a1,b1,c1,a2,b2,c2: Extended;
begin
  Result:=True;
  x1:=x12-x11; y1:=y12-y11;
  x2:=x22-x21; y2:=y22-y21;
  a1:=y1; b1:=-x1; c1:=y11*x1-x11*y1;
  a2:=y2; b2:=-x2; c2:=y21*x2-x21*y2;
```

```
    If (a1*b2-a2*b1=0) Or (a1*b2-a2*b1=0) Then
     Begin
       Result:=False;
       Exit;
     End;
    x:= (b1*c2-b2*c1)/(a1*b2-a2*b1);
    y:=-(a1*c2-a2*c1)/(a1*b2-a2*b1);
  end;
```

Одним з основних є метод Draw, призначений для малювання робочої області, а саме відображення координатної сітки, осей координат та їх підписів.

```
  procedure TGrPlace.Draw;
  Var
    q, c1: Double;
    a, b, c: Double;
  Procedure DrawOsi_Grid;
  Begin
    With Canvas Do Begin
     If (FDrawGrid) Then
      Begin
       For j:=Trunc(AB[1].ex/q) To trunc(AB[3].ex/q) Do
        For j1:=trunc(AB[1].ey/q) DownTo
              trunc(AB[3].ey/q) Do
          Begin
            AbsToScreenZ(j*q,j1*q,tx,ty);
            pixels[tx,ty]:=FColorOsi;
          End;
      End;
     If FPutOsi Then
      Begin
        //побудова осей та виведення підписів на них
        . . .
```

```
    End;
  End;
End;

begin
  If FlagDrawScene Then Exit;
  DecimalSeparator:='.';
  FlagDrawScene:=True;
  Align:=alClient;
  Fxc:=Width div 2;
  Fyc:=Height div 2;
  with AB[1] do ScreenToAbsZ(0,0, ex,ey);
  with AB[2] do ScreenToAbsZ(Width,0, ex,ey);
  with AB[3] do ScreenToAbsZ(Width,Height, ex,ey);
  with AB[4] do ScreenToAbsZ(0,Height, ex,ey);
  AB[5]:=AB[1];
   Try
    Picture.Bitmap.Width:=Width;
    Picture.Bitmap.Height:=Height;
    Canvas.Brush.Color:=FColorObl;
    Picture.Bitmap.Canvas.fillRect(ClientRect);
    a:=(AB[1].ex-AB[3].ex);
    b:=(AB[1].ey-AB[3].ey);
    c:=max(a,b);
    c1:=1;
    While c<c1 Do c1:=c1/10;
    c:=c+c1;
    Q:=0.0000000000001;
    Repeat
      Q:=Q*10;
    until c/q<21;
    DrawOsi_Grid;
   Except
```

```
    End;
   FlagDrawScene:=False
end;
```

Серед методів, визначених в даному класі, також присутні методи, які відносяться до обробки певних подій:

переміщення курсору миші над робочою областю:

```
procedure GrMouseMove(Sender: TObject; Shift:
     TShiftState; X, Y: Integer);
```

натиснення та відпускання кнопок миші:

```
procedure GrMouseDown(Sender: TObject; Button:
     TMouseButton; Shift: TShiftState; X, Y:
     Integer);
  procedure GrMouseUp(Sender: TObject; Button:
     TMouseButton; Shift: TShiftState; X, Y:
     Integer);
```

Дані методи, в залежності від натисненних кнопок миші, використовуються для відображення координат ПДСК чи характеристик об’єкту під курсором, виділення об’єкту в робочій області, переміщення об’єкту, виклику відповідного контекстного меню та ін..

Для реалізації можливості переміщення системи координат метод може мати вигляд:

```
procedure TGrPlace.GrMouseMove(Sender: TObject;
     Shift: TShiftState; X,  Y: Integer);
begin
  ScreenToAbsZ(X,Y,FTecDX,FTecDY);
  If (ssCtrl in Shift) And (ssLeft in Shift) Then
   Begin
     FRolX:=FRolX+round((FTecDX*FAZoom-LastX)*FZoom);
     FRolY:=FRolY+round((FTecDY*FAZoom-LastY)*FZoom);
     ScreenToAbsZ(X,Y,FTecDX, FTecDY);
```

```
      Draw;
    End;
  LastX:=FTecDX*FAZoom;
  LastY:=FTecDY*FAZoom;
end;
```

Йдучи далі, можна зазначити, що програмний засіб Gran2D працює з певною групою геометричних об'єктів (точка, лінія, коло, …). Виходячи з цього, необхідно визначити деякі загальні особливості, характеристики всіх цих об'єктів і розмістити їх в об'єкті, який буде головним у дереві ієрархії геометричних об'єктів, а інші об'єкти (класи) будуть доповнюватися власними специфічними лише для них характеристиками.

Таким чином, при створенні ППЗ Gran2D було визначено головний клас TParentObj, від якого будуть походити всі інші класи геометричних об'єктів.

Опис даного класу має наступний вигляд

```
TParentObj=class(TObject)
private
  { опис полів }
  . . .
protected
  function GetTypeName: String; virtual; abstract;
public
  RefCount: Integer;
  Ref: Array[1..2] Of Integer;
  GrPlace: TGrPlace;
  Constructor Create;
  procedure ShowSett(M: TStrings; Move:
       Boolean=False); virtual; abstract;
  function ShowInfo: String; virtual; abstract;
  procedure Draw; virtual; abstract;
  procedure Save(f: TIniFile; Sec: String); virtual;
```

```
    procedure Load(f: TIniFile; Sec: String; List:
         TStrings); virtual;
    property TypeObj: ShapeType read FTypeObj write
         FTypeObj;
    property Name: String  read FName write FName;
    property Visible: Boolean  read FVisible write
         FVisible;
    property Select: Boolean read FSelect write
         FSelect;
    property ReCalc: Boolean read FReCalc write
         FReCalc;
  End;
```

Для даного класу були виділені такі властивості: будь-який об'єкт повинен мати ім'я `Name: String`, належати до певного типу `TypeObj: ShapeType` (точка, пряма, відрізок, промінь, та ін.) `ShapeType=(shNone, shDot, shLineVidriz,shLinePromin,shLine)`, містити покажчики, чи є об'єкт в даний момент видимим `Visible: Boolean` та виділеним `Selected: Boolean`. Об'єкт може залежати від інших об'єктів, наприклад, лінія залежить від двох точок. Для цього введено поле `RefCount: Integer` для визначення кількості базових об'єктів та поле `Ref : Array[1..2] Of Integer`, яке буде відповідати за цей зв'язок. При розширенні можливостей програмного засобу, наприклад введенні об'єкту ламана, розмір масиву може бути збільшено.

Зрозуміло що при зміні положення точки, яка є базовою, наприклад, для лінії, лінія також має змінити власне положення. Проте дана зміна має відбутися лише після перерахунку координат точки, для збільшення швидкодії. Так як такий перерахунок може відбуватися з різних частин програми, введено поле `ReCalc: Boolean`, яке буде містити покажчик

того, що характеристики об'єкту перераховані і можна приступати до перерахунку об'єктів, залежних від даного.

Зважаючи на те, що ПЗ базується на відображенні об'єктів, необхідне поле, що є посиланням на об'єкт графічної області `GrPlace: TGrPlace`.

При визначенні методів для батьківського класу необхідно передбачити можливість роботи з файлами, збереження та завантаження об'єктів. Для цього призначені процедури, які в нащадках будуть довизначатися в залежності від введення нових характеристик об'єкту:

```
procedure Save(f: TIniFile; Sec: String); virtual;
procedure Load(f: TIniFile; Sec: String; List:
     TStrings); virtual;
```

Реалізація даних методів для класу `TParentObj` така:

```
procedure TParentObj.Save(f: TIniFile; Sec: String);
Var ii: Integer;
    ss: String;
begin
  f.WriteString (Sec,'ObjType',   GetTypeName);
  f.WriteString (Sec,'Name',      FName);
  f.WriteBool   (Sec,'Visible',   FVisible);
  f.WriteBool   (Sec,'Select' ,   FSelect);
  f.WriteInteger(Sec,'RefCount',  RefCount);
  For ii:=1 To RefCount Do
   Begin
     ss:='ref'+IntTostr(ii);
     f.WriteInteger(Sec,ss,ref[ii]);
   End;
end;

procedure TParentObj.Load(f: TIniFile; Sec: String;
     List: TStrings);
```

```
Var ii: Integer;
    ss: String;
begin
  (List.Objects[List.Count-1] as TParentObj).Name:=
       f.ReadString(Sec,'Name','Noname');
  (List.Objects[List.Count-1] as
       TParentObj).Visible:=
       f.ReadBool(Sec,'Visible',True);
  (List.Objects[List.Count-1] as TParentObj).Select:=
       f.ReadBool(Sec,'Select', True);
  (List.Objects[List.Count-1] as
       TParentObj).RefCount:=
       f.ReadInteger(Sec,'RefCount',0);
   For ii:=1 To (List.Objects[List.Count-1] as
       TParentObj).RefCount Do
    Begin
        ss:='ref'+IntTostr(ii);
        (List.Objects[List.Count-1] as
            TParentObj).Ref[ii]:=f.ReadInteger(Sec,ss
            ,-1);
    End;
end;
```

Крім того, визначені методи, які будуть виводити коротку `ShowInfo: String` та повну `ShowSett(M: TStrings; Move: Boolean=False)` інформацію про об'єкт. Коротка інформація може виводитися під час наведення курсора миши на об'єкт, повна – при виділенні об'єкту в списку об'єктів. Метод `Draw` призначений для відображення об'єкту.

Більшість з зазначених методів є віртуальними, це необхідно для можливості їх подальшої підміни в класах-нащадках. Крім того, деякі з методів є абстрактними, оскільки їх не можна визначити для загального класу.

Серед методів є захищена функція

```
function GetTypeName: String; virtual; abstract;
```

яка повертає текстову назву об'єкту. Дана функція використовується при записі об'єкту в файл.

Отже після того, як буде описаний батьківський клас з визначеними загальними особливостями, можна приступати до опису похідних класів, тобто класів-нащадків, які будуть описувати сам об'єкт, або знову бути деяким проміжним класом для визначення і доповнення деяких особливостей групи інших об'єктів.

Так наприклад при визначенні класу точка він може бути доповнений власними особливостями, серед яких можна виділити:

- координати точки в ПДСК – `DPx, DPy` та на площині відображення – `DGx, DGy` (оскільки буде необхідність знати як явні координати точки, так і ті, відносно яких вона буде зображуватися на робочій області);
- візуальні властивості, такі як розмір (радіус) – `Dsize: Byte` точки, колір лінії – `Dcolor: TColor`, колір зафарбування – `DFill: TColor` та необхідність зафарбування – `DfillShow: Boolean`;
- поле, яке буде відповідати за підпис точки `DLab: TTextPoint` (визначення класу `TTextPoint` буде показано пізніше);
- параметри підпису: необхідність відображення підпису – `DnameShow: Boolean` та координат – `DcoorShow: Boolean`.

Та зрозуміло, необхідно буде підмінити (довизначити) методи, описані в батьківському класі та при необхідності визначити власні.

Крім того, передбачимо перезавантажений конструктор для створенная об'єкту точка з відповідними координатами.

```
TPointObj=class(TParentObj)
```

```
private
  FDGx, FDGy: Double;
  FDCoorShow: Boolean;
  FDNameShow: Boolean;
  FDFillShow: Boolean;
  FDSize: Byte;
  FDFill: TColor;
  FDColor: TColor;
  procedure SetDGx(const Value: Double);
  procedure SetDGy(const Value: Double);
  procedure SetVisible(const Value: Boolean);
  function GetTypeName: String; override;
  procedure ShowCaption;
public
  DPx, DPy  : Integer;
  DLab: TTextPoint;
  constructor Create(TypeDot: ShapeType; APlace:
       TGrPlace=nil); overload;
  constructor Create(x1, y1: Double; TypeDot:
       ShapeType; APlace: TGrPlace=nil); overload;
  Destructor Destroy; override;
  function ShowInfo: String; override;
  procedure ShowSett(M: TStrings; Move:
       Boolean=False); override;
  procedure Draw; override;
  procedure Save(f: TIniFile; Sec: String); override;
  procedure Load(f: TIniFile; Sec: String; List:
       TStrings); override;
  property DGx: Double  read FDGx write SetDGx;
  property DGy: Double  read FDGy write SetDGy;
  property DColor: TColor read FDColor write FDColor;
  property DSize: Byte read FDSize write FDSize;
  property DFill: TColor read FDFill write FDFill;
```

```
    property DFillShow : Boolean read FDFillShow write
         FDFillShow;
    property DNameShow: Boolean read FDNameShow write
         FDNameShow;
    property DCoorShow: Boolean read FDCoorShow write
         FDCoorShow;
  End;
```

Метод

```
  procedure ShowCaption;
```

призначений для формування та відображення підпису точки в залежності від налагоджень. Його реалізація має вигляд:

```
  procedure TPointObj.ShowCaption;
  Var ss: String;
  begin
    DLab.Visible:=((DNameShow) Or (DCoorShow)) And
         Visible;
    If DNameShow
     Then ss:=Name+''
     Else ss:='';
    If DCoorShow Then
     ss:=ss+' ('+SValue(FDGx)+'; '+SValue(FDGy)+')';
    DLab.GraphCaption(ss, clBlack, 8, 'Courier New');
  end;
```

Методи

```
  procedure SetDGx(const Value: Double);
  procedure SetDGy(const Value: Double);
```

будуть викликатися при зміні координат точок в ПДСК і перераховувати ці координати в координати робочої області, для подальшого відображення точки, та перераховувати положення підпису.

Визначення методу `SetDGx` має вигляд:

```
  procedure TPointObj.SetDGx(const Value: Double);
```

```
Var yyy: Integer;
begin
  FDGx:= Value;
  GrPlace.AbsToScreenZ(Value, 0, Dpx, yyy);
  DLab.Left:=Dpx+10;
end;
```

Реалізація процедури відображення точки запишеться в такому вигляді:

```
procedure TPointObj.Draw;
Var tx, ty, sz: Integer;
begin
  DLab.Visible:=False;
  If Visible Then
   Begin
     GrPlace.Canvas.Pen.Color:=FDColor;
     GrPlace.Canvas.Pen.Style:=psSolid;
     GrPlace.Canvas.Pen.Width:=1;
     GrPlace.Canvas.Brush.Color:=FDFill;
     GrPlace.Canvas.Brush.Style:=bsSolid;
     sz:=FDSize;
     tx:=DPx;
     ty:=DPy;
     If DFillShow
      Then
       GrPlace.Canvas.ellipse(tx-sz, ty-sz, tx+sz+1,
       ty+sz+1)
      Else
       GrPlace.Canvas.Arc(tx-sz, ty-sz, tx+sz+1,
          ty+sz+1, tx, ty-sz, tx, ty-sz);
  End;
end;
```

Крім методу Draw необхідно визначити методи виведення інформації про об'єкт, метод GetTypeName та довизначити методи збереження та завантаження об'єкту.

Аналогічно клас прямої може бути визначеним так:

```
TLineObj=class(TParentObj)
private
  Ax, By, Cz: Extended;
  FName1: String;
  FName2: String;
  FLSize: Byte;
  FLStyle: Byte;
  FLColor: TColor;
  function GetTypeName: String; override;
public
  Gx1, Gy1, Gx2, Gy2: Extended;
  Px1, Py1, Px2, Py2: Integer;
  constructor Create(TypeLine: ShapeType; APlace:
       TGrPlace=nil);
  function ShowInfo: String; override;
  procedure ShowSett(M: TStrings; Move:
       Boolean=False); override;
  procedure SetPropLine2Dot(xx1, yy1, xx2, yy2:
       Extended);
  procedure Draw; override;
  procedure SetNameDot(s1,s2: String);
  procedure Save(f: TIniFile; Sec: String); override;
  procedure Load(f: TIniFile; Sec: String; List:
       TStrings); override;
  property LStyle: Byte read FLStyle write FLStyle;
  property LColor: TColor read FLColor write FLColor;
  property LSize: Byte read FLSize write FLSize;
```

```
End;
```

Серед специфічних властивостей визначені:

- тип лінії `LStyle` (пряма, промінь, відрізок);
- коефіцієнти рівняння отримуваної прямої `Ax, By, Cz`;
- координати точок, на яких будується пряма в ПДСК `Gx1, Gy1, Gx2, Gy2`; та відповідних точок робочої області `Px1, Py1, Px2, Py2`;
- `FName1, FName2` – назви точок, на яких базується лінія та методи доступу до них, оскільки поля є захищеними.

Крім того, виділено метод `SetPropLine2Dot`, за допомогою якого визначаються коефіцієнти прямої за двома точками на ній.

```
procedure TLineObj.SetPropLine2Dot(xx1, yy1, xx2,
       yy2: Extended);
begin
  Gx1:=xx1;
  Gy1:=yy1;
  GrPlace.AbsToScreenZ(xx1, yy1, Px1, Py1);
  Gx2:=xx2;
  Gy2:=yy2;
  GrPlace.AbsToScreenZ(xx2, yy2, Px2, Py2);
  Ax:=Gy2-Gy1;
  By:=Gx1-Gx2;
  Cz:=Gy1*(Gx2-Gx1)-Gx1*(Gy2-Gy1);
end;
```

При необхідності створити лінію не на основі двох точок, а наприклад перпендикулярно чи паралельно до деякої іншої лінії, необхідно ввести метод, який би перераховував коефіцієнти прямої на основі базових об’єктів (точки та прямої).

Для класу лінії, як і для класу точки необхідно визначити та довизначити методи, оголошені в батьківському класі.

Так метод відображення лінії буде мати вигляд:

```
procedure TLineObj.Draw;
begin
  If Visible Then
   Begin
     GrPlace.Canvas.Pen.Color:=LColor;
     GrPlace.Canvas.Pen.Style:=Styles[LStyle];
     GrPlace.Canvas.Pen.Width:=LSize;
     GrPlace.Canvas.Brush.Color:=c_ColorObl;
     GrPlace.Canvas.Brush.Style:=bsSolid;
     SetPropLine2Dot(Gx1, Gy1, Gx2, Gy2);
     GrPlace.DrawLine(Gx1, Gy1, Gx2, Gy2, 0);
   End;
end;
```

Метод виведення повної інформації про об'єкт матиме такий вигляд:

```
procedure TLineObj.ShowSett(M: TStrings; Move:
     Boolean=False);
Var sa: String;
begin
  M.Clear;
  sa:=ShowInfo;
  M.Add(sa);
  If TypeObj=shLineVidriz
   Then M.Add(Format('Довжина відрізку: %s',
        [SValue(Len_Dot_Dot(Gx1,Gy1, Gx2,Gy2))]));
  sa:='';
  If Ax<>0 Then sa:=SValue(Ax)+'x';
  If By<>0 Then
   Begin
     If By<0
      Then sa:=sa+SValue(By)+'y'
      Else sa:=sa+'+'+SValue(By)+'y';
```

```
     End;
    If Cz<>0 Then
     Begin
       If Cz<0
        Then sa:=sa+SValue(Cz)
        Else sa:=sa+'+'+SValue(Cz);
     End;
    M.Add(Format('Рівняння прямої: %s=0',[sa]));
  end;
```

де функція `len_Dot_Dot` призначена для обчислення відстані між двома точками, заданими своїми координатами.

Крім батьківського класу геометричних об'єктів в ППЗ Gran2D був визначений батьківський клас для текстових написів, які можуть міститися в робочій області програми. Серед таких текстових об'єктів в програмі можуть зустрічаються: підпис точки, текстовий напис, підпис обчислень кута та відстані між точками.

```
  TTextObj=class(TLabel)
  private
    XOld, YOld: Integer;
    FlagMove: Boolean;
    procedure LabelMouseUp(Sender: TObject; Button:
         TMouseButton; Shift: TShiftState; X, Y:
         Integer);virtual;
    procedure LabelMouseDown(Sender: TObject; Button:
         TMouseButton; Shift: TShiftState; X, Y:
         Integer);virtual;
    procedure LabelMouseMove(Sender: TObject; Shift:
         TShiftState; X,Y: Integer); virtual;
  public
    GrPlace: TGrPlace;
    constructor Create(AOwner: TComponent); overload;
         override;
```

```
    constructor Create(AOwner: TComponent; APlace:
         TGrPlace); overload;
    procedure SetPopMenu(APopupMenu: TPopupMenu);
    procedure Save(f: TIniFile; Sec: String); virtual;
         abstract;
    procedure Load(f: TIniFile; Sec: String; List:
         TStrings); virtual; abstract;
  end;
```

Даний клас характеризується такими загальними властивостями для всіх написів: приналежність до графічної області `GrPalce : TGrPlace;` поля, які використовуються при переміщенні напису по робочій області `XOld: Integer, YOld: Integer`.

Серед визначених методів присутній метод `SetPopMenu` – підключення власного контекстного меню до об'єкту. Абстрактні методі, які будуть перевизначені в нащадках, для збереження та завантаження інформації про об'єкт `Load, Save`, та методи по опрацюванню подій миші, які надають можливість переміщувати напис по робочій області `LabelMouseDown, LabelMouseMove, LabelMouseUp`.

Реалізація даних методів така:

```
  procedure TTextObj.LabelMouseDown(Sender: TObject;
       Button: TMouseButton; Shift: TShiftState; X, Y:
       Integer);
  begin
    If Button<>mbLeft
     Then FlagMove:=False
     Else Begin
      FlagMove:=True;
      XOld:=X;
      YOld:=Y;
     End
  end;
```

```
procedure TTextObj.LabelMouseMove(Sender: TObject;
     Shift: TShiftState; X, Y: Integer);
begin
  If (FlagMove) Then
   Begin
    Left:=Left+X-XOld;
    Top:=Top +Y-YOld;
   End;
end;

procedure TTextObj.LabelMouseUp(Sender: TObject;
     Button: TMouseButton; Shift: TShiftState; X, Y:
     Integer);
begin
  FlagMove:=False;
end;
```

Як нащадки від `TTextObj` визначимо клас для підпису точки `TText-Point` та клас для текстового напису `TTextGraph`, перевизначивши деякі методи та ввівши додаткові поля.

```
TTextPoint=class(TTextObj)
public
   Dx, Dy: Integer;
   FWith, FHeight: Integer;
   procedure GraphCaption(const Value: String;
     AColor: TColor; ASize: Integer; AName: String);
   procedure LabelMouseMove(Sender: TObject; Shift:
     TShiftState; X, Y: Integer);override;
end;
```

```
TTextGraph=class(TTextObj)
private
  FTexts: TStrings;
  procedure SetTexts(const Value: TStrings);
public
  constructor Create(AOwner: TComponent); overload;
       override;
  Destructor Destroy; override;
  procedure Save(f: TIniFile; Sec: String); override;
  procedure Load(f: TIniFile; Sec: String; List:
       TStrings); override;
  property Texts: TStrings read FTexts write
       SetTexts;
end;
```

Так в класі `TTextPoint` визначено: змінні `Dx`, `Dy` в яких містяться координати точки до якої відноситься підпис; `FWith`, `FHeight` – відстані між точкою та підписом, які будуть використовуватися при переміщенні точки, щоб зберігати встановлену відстань; метод `GraphCaption` для формування та відображення підпису; та перевизначено метод LabelMouseMove – в якому буде передбачено неможливість відвести підпис далеко від точки.

Так в методі LabelMouseMove буде перевірятися, якщо напис знаходиться в межах визначеного кола (радіусом 15 пікселів) то його буде перемыщено батькывським методом:

```
procedure TTextPoint.LabelMouseMove(Sender: TObject;
       Shift: TShiftState; X, Y: Integer);
Var w, h: Double;
begin
  w:=Width/2;
  h:=Height/2;
```

```
    If sqr((Left+X-XOld-(Dx-w))/(15+w))+sqr((Top+Y-
         YOld-(Dy-h))/(15+h))<1
     Then inherited;
    FWith:=Left-Dx;
    FHeight:=Top-Dy;
  end;
```

В класі `TTextGraph` описані методи роботи з файлами.

ПЗ Gran2D базується на MDI-стандарті, коли в межах головного вікна програми існують декілька вікон для відображення відповідних даних. Проте всі елементи можна розмістити в межах одного вікна, як це було в версії ПЗ до 2005 року.

Будь-який сучасний програмний засіб, в більшості випадків, повинен мати головне меню (`GlobalMenu: TMainMenu`) та рядок статусу для відображення деякої поточної інформації (`StatusL: TStatusBar`). Вміст головного меню буде залежати від можливостей, визначених в ПЗ. Також деякі пункти головного меню можуть дублюватися в контекстних меню візуальних елементів. Так, наприклад, можливість збільшення та зменшення масштабу може відображатися в контекстному меню робочої області, а можливість створення та зміни об’єктів в контекстному меню списку об’єктів.

Для організації створення таких об’єктів, як точка та пряма, використовуються два діалогові вікна (рис. 1, рис. 2). Дані вікна будуть використовуватися і при необхідності змінити вже існуючий об’єкт, зрозуміло, що в такому випадку необхідно завантажити в елементи інтерфейсу властивості об’єкта, що редагується.

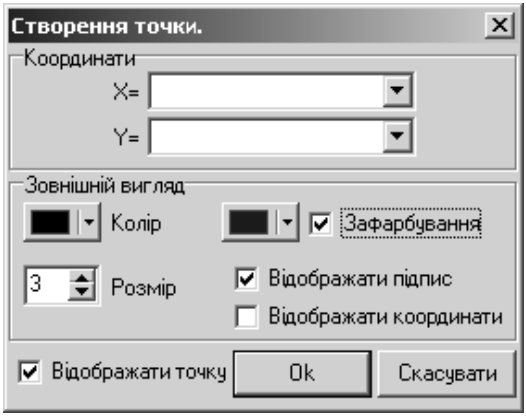


Рис. 1

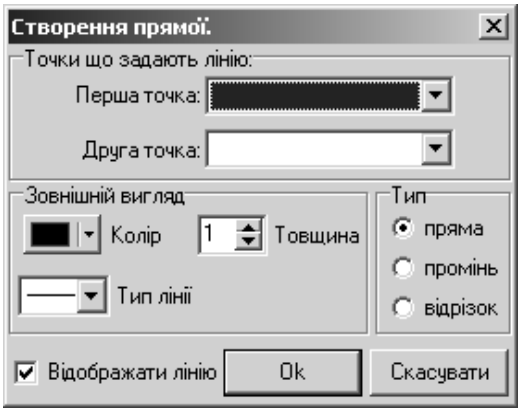


Рис. 2

Для розміщення створених об’єктів буде використовуватися елемент `ListObj: TCheckListBox`, який в кожному елементі списку буде містити деякий об’єкт.

При зміні параметрів робочої області або характеристик об’єкту необхідно буде перераховувати інші, залежні об’єкти та перемальовувати робочу область. Для перерахунку одного об’єкту описано процедуру

ReCalcActObj, в якій спочатку, викликаючись ркурсивно, перераховуються об'єкти, від яких залежить поточний, а потім власне перераховується сам об'єкт. Для перерахунку всіх об'єктів описано процедуру ReCalcObj. Перемалювання робочої області викликається процедурою ReDraw в якій спочатку перемальовується координатна площина, а потім всі об'єкти.

```
Procedure ReCalcActObj(n: Integer);
Var j: Integer;
Begin
  With FMain.ListObj.Items Do
   For j:=1 To (Objects[n] as TParentObj).RefCount Do
    With (Objects[n] as TParentObj) Do
     Begin
      If ((Objects[ref[j]] as
          TParentObj).ReCalc=False) And
          ((Objects[ref[j]] as TParentObj).RefCount>0)
       Then ReCalcActObj(ref[j]);
      (Objects[ref[j]] as TParentObj).ReCalc:=True;
    End;
  With FMain.ListObj.Items Do
   Case (Objects[n] as TParentObj).TypeObj Of
     shDot: Begin
        (Objects[n] as TPointObj).DGx:=
        (Objects[n] as TPointObj).DGx;
        (Objects[n] as TPointObj).DGy:=
        (Objects[n] as TPointObj).DGy;
     End;
     shLineVidriz..shLine : Begin
        (Objects[n] as
            TLineObj).SetPropLine2Dot( (FMain.ListObj
            .Items.Objects[(Objects[n] as
            TParentObj).Ref[1]] as TPointObj).DGx,
```

```
            (FMain.ListObj.Items.Objects[(Objects[n]
            as  TParentObj).Ref[1]] as
            TPointObj).DGy,
            (FMain.ListObj.Items.Objects[(Objects[n]
            as TParentObj).Ref[2]] as TPointObj).DGx,
            (FMain.ListObj.Items.Objects[(Objects[n]
            as TParentObj).Ref[2]] as
            TPointObj).DGy);
      End;
   End;
End;

Procedure ReCalcObj;
Var i: Integer;
Begin
  DecimalSeparator:='.';
  With FMain.ListObj.Items Do
  Begin
    For i:=0 To Count-1 Do
     (Objects[i] as TParentObj).ReCalc:=False;
    For i:=0 To Count-1 Do
     Begin
     Try
      If ((Objects[i] as TParentObj).ReCalc=False)
         And ((Objects[i] as TParentObj).RefCount>0)
       Then ReCalcActObj(i);
      (Objects[i] as TParentObj).ReCalc:=True;
      If (Objects[i] as TParentObj).TypeObj=shDot
         Then
       Begin
        (Objects[i] as TPointObj).DGx:=(Objects[i] as
            TPointObj).DGx;
```

```
          (Objects[i] as TPointObj).DGy:= (Objects[i]
              as TPointObj).DGy;
         End;
         Except
         End;
        End;
    End;
    If (FMain.ListObj.ItemIndex<>-1)
     Then FMain.ListObjClick(Application);
End;


procedure ReDraw;
Var i: Integer;
begin
  GrPlace.Draw;
  For i:=0 To FMain.ListObj.Items.Count-1 Do
   (FMain.ListObj.Items.Objects[i] as
     TparentObj).Draw;
end;
```

Зрозуміло, що було розглянуто лише малу частину об’єктів ПЗ Gran2D, і тільки з їх основними можливостями.

В інших програмах, що будуть використовувати описані вище класи, ці класи необхідно буде доопрацювати, або розширити у відповідності до предметної області.

Таким чином створення ППЗ можна рзбити на декілька етапів. На першому етапі аналізується предметна область, виявляються її об‘єкти та властивості об’єктів, встановлюються зв‘язки між об’єктами і на основі цього аналізу та вимог до ППЗ створюють проект ППЗ. На другому етапі реалізують класи, що відповідають виділеним на першому етапі об’єктам та

створюють ППЗ з інтерфейсом, що дозволяє зручно працювати з об'єктами предметної області. На третьому етапі тестують отриманий ППЗ.

7.2. Завдання для студентів

Завдання 1.

Розробити клас TFunc:

```
TFunc=class(TObject)
    Tx,St: String;
    Arr: RealArray;
  public
    constructor Create(SF: String);
    function PointX(Arg:extended): Extended; virtual;
    function PointY(Arg:extended): Extended; virtual;
    function Simpson(AI, BI: Extended): Extended;
    procedure ChangeFunc(SF: String; var Chng:
         Boolean);virtual;
    function Eval(ArgX, ArgY: Extended): Extended;
End;
```

`Tx, St, Arr` – поля для збереження виразу. Конструктор `Create`, методи `Eval` (обчислення значення виразу) та `ChangeFunc` (зміна виразу функції) наведено в додатку.

Метод `PointX` повинен повертати значення `X`-координати у декартовій системі координат, метод `PointY` – значення `Y`-координати у декартовій системі координат.

Клас розмістити у окремому модулі `NGFunc`, підключивши до нього модуль `NGMath` (наведено у додатку). Даний модуль містить процедури, що на основі рядку запису виразу будують певну структуру даних, яка використовується у методі `Eval()` для обчислення значення цього виразу.

Написати консольну програму, яка вводить значення виразу з клавіатури і обчислює значення інтегралу від цього виразу на відрізку, також введеному з клавіатури.

Завдання 2.

Розробити клас для відображення ПДСК `TgrPlace`, розмістивши його в модулі `UGraphObj`:

```
TGrPlace=class(TImage)
  Private
  FPutOsi: Boolean;
  FColorOsi: TColor;
  FColorObl: TColor;
  Fxc: Integer;
  Fyc: Integer;
  FRolX: Integer;
  FRolY: Integer;
  FZoom: Extended;
  FAZoom: Integer;
  FTecDX, FTecDY: Extended;
  FlagDrawScene: Boolean;
  LastX, LastY: Double;
  AB: Array[1..5] Of Record
     ex, ey: Extended;
   End;
  Public
  constructor Create(AOwner: TComponent); override;
  procedure ScreenToAbsZ(x, y: Integer; var xx, yy:
     Extended);
```

```
    procedure AbsToScreenZ(x, y: Extended; var xx,
       yy:Extended); overload;
    procedure AbsToScreenZ(x, y: Extended; var xx,
       yy:Integer);  overload;
    procedure SetZoom(n: Extended);
    procedure DrawLine(x1,y1,x2,y2:Extended; Lt:
       Byte=0);
    procedure Draw;
    procedure GrMouseMove(Sender: TObject; Shift:
       TShiftState; X, Y: Integer);
    property ColorObl: TColor read FColorObl write
       FColorObl;
    property PutOsi  : Boolean read FPutOsi write
       FPutOsi;
    property ColorOsi: TColor read FColorOsi write
       FColorOsi;
    property TecDX: Extended read FTecDX;
    property TecDY: Extended read FTecDY;
    property AZoom: Integer read FAZoom;
End;
```

Описаний клас `TgrPlace` можна оформити в вигляді окремої компоненти та додати її до списку компонент Lazarus і в подальшому використовувати компоненту, а не окремий клас.

Створити програму з візуальним інтерфейсом для відображення ПДСК описаної в класі `TgrPlace`. Крім області де буде розміщено екземпляр класу `TgrPlace`, розмістити компоненту TListBox – для зберігання в ньому списку створюваних об’єктів (буде використовуватися в наступних завданнях), компоненту TMemo – для відображення інформації про виділений об’єкт в списку об’єктів (буде використовуватися в наступних завданнях) та

TMainMenu – для організації виконання операцій (створити пункт головного меню для налагодження ПДСК, виклик якого буде відкривати діалогове вікно з вибором можливих налагоджень). Передбачити автоматичне створення і розміщення класу при запуску програми та його знищення при закритті програми; область ПДСК має автоматично перемальовуватися при зміні розмірів вікна.

Завдання 3.

Описати клас TParentObj, розмістивши його в модулі UParentObj:

```
  TParentObj=class(TObject)
  private
    FName: String;
    FVisible : Boolean;
    FTypeObj: ShapeType;
  public
    RefCount: Integer;
    Ref: Array[1..2] Of Integer;
    GrPlace: TGrPlace;
    Constructor Create;
    procedure ShowSett(M: TStrings; Move:
       Boolean=False); virtual; abstract;
    procedure Draw; virtual; abstract;
    property TypeObj: ShapeType read FTypeObj write
       FTypeObj;
    property Name: String  read FName write FName;
    property Visible: Boolean  read FVisible write
       FVisible;
  End;
```

Розробити клас точки TPointObj, розмістивши його в модулі UPointObj:

```
TPointObj=class(TParentObj)
private
  FDGx, FDGy: Double;
  FDFill: TColor;
  FDColor: TColor;
  procedure SetDGx(const Value: Double);
  procedure SetDGy(const Value: Double);
public
  DPx, DPy  : Integer;
  DLab: String;
  constructor Create(TypeDot: ShapeType; APlace:
    TGrPlace=nil); overload;
  constructor Create(x1, y1: Double; TypeDot:
    ShapeType; APlace: TGrPlace=nil); overload;
  Destructor Destroy; override;
  procedure ShowSett(M: TStrings; Move:
    Boolean=False); override;
  procedure Draw; override;
  property DGx: Double  read FDGx write SetDGx;
  property DGy: Double  read FDGy write SetDGy;
  property DColor: TColor read FDColor write FDColor;
  property DFill: TColor read FDFill write FDFill;
End;
```

Доповнити програму створену в попередньому завданні додавши до неї, як пункт меню, можливість створення точки, виклик якого буде відкривати діалогове вікно з вибором можливих налагоджень нової точки. При створенні точки мають задаватися її можливі властивості та має бути

присутня перевірка яка б унеможливлювала ствоення декількох точок з однаковими іменами. Створені точки, як окресі об'єкти, мають розміщуватися в списку об'єктів. Описати підпрограму, яка б могла використовуватися для перемальовування області побудови та всіх створених об'єктів, описати виклик цієї підпрограми в необхідних місцях коду для правильного функціонування програми (чи замінити ним виклик перемальювання області ПДСК). При відображенні точки, поряд з нею має відображатися і її назва. Передбачити виведення інформації про виділений об'єкт в списку об'єктів в компоненту TMemo.

Завдання 4.

Розробити клас прямої `TLineObj`, розмістивши його в модулі `ULineObj`:

```
  TLineObj=class(TParentObj)
  private
    Ax, By, Cz: Extended;
    FName1: String;
    FName2: String;
    FLSize: Byte;
    FLColor: TColor;
  public
    Gx1, Gy1, Gx2, Gy2: Extended;
    Px1, Py1, Px2, Py2: Integer;
    constructor Create(TypeLine: ShapeType; APlace:
       TGrPlace=nil);
    procedure ShowSett(M: TStrings; Move:
       Boolean=False); override;
```

```
    procedure SetPropLine2Dot(xx1, yy1, xx2, yy2:
      Extended);
    procedure Draw; override;
    procedure SetNameDot(s1,s2: String);
    property LColor: TColor read FLColor write FLColor;
    property LSize: Byte read FLSize write FLSize;
  End;
```

Доповнити програму створену в попередньому завданні, передбачити можливість створення лінії, променя, відрізку через діалогове вікно з можливими налагодженнями для створюваного об'єкту. Створені об'єкти мають розміщуватися в списку об'єктів.

Завдання 5.

Розробити класи `TTextObj` та `TTextPoint`, розмістивши їх в модулі UTextObj:

```
  TTextObj=class(TLabel)
  private
    XOld, YOld: Integer;
    FlagMove: Boolean;
    procedure LabelMouseUp(Sender: TObject; Button:
      TMouseButton; Shift: TShiftState; X, Y:
      Integer);virtual;
    procedure LabelMouseDown(Sender: TObject; Button:
      TMouseButton; Shift: TShiftState; X, Y:
      Integer);virtual;
    procedure LabelMouseMove(Sender: TObject; Shift:
      TShiftState; X,Y: Integer); virtual;
  public
    GrPlace: TGrPlace;
```

```
    constructor Create(AOwner: TComponent); overload;
       override;
    constructor Create(AOwner: TComponent; APlace:
       TGrPlace); overload;
    procedure SetPopMenu(APopupMenu: TPopupMenu);
    procedure Save(f: TIniFile; Sec: String); virtual;
       abstract;
    procedure Load(f: TIniFile; Sec: String; List:
       TStrings); virtual; abstract;
  end;


  TTextPoint=class(TTextObj)
  public
    Dx, Dy : Integer;
    FWith, FHeight : Integer;
    procedure GraphCaption(const Value: String; AColor:
         TColor; ASize: Integer; AName: String);
    procedure LabelMouseMove(Sender: TObject; Shift:
         TShiftState; X, Y: Integer);override;
  end;
```

Змінити в класі `TPointObj` поле `Dlab`, задавши йому тип `TTextPoint`. Довизначити конструктор в якому передбачити створення напису, доповнити методи `SetDGx` та `SetDGy` операторами для задання положення напису та внесення в поля підпису необхідних значень, внести зміни в метод відображення точки. Доповнити класс точки необхідними полями для можливості задання форматів підпису та внести необхідні налагодження в вікно створення точки.

Завдання 6.

Довизначити клас `TParentObj` методом для виведення короткої інформації про об'єкт:

```
function ShowInfo: String; virtual; abstract;
```

Перевизначити даний метод в класах нащадках `TPointObj`, `TLineObj`.

Доповнити класи об'єктів методами роботи з файлами для їх збереження та завантаження.

Доповнити клас `TGrPlace` та програму можливістю аналізу знаходження курсора миші над деяким об'єктом, тобто коли курсор миші знаходиться над об'єктом в робочій області, необхідно відображати коротку інформацію про об'єкт.

Завдання 7.

Доповнити програму можливістю:

- змінювати раніше створені об'єкти;
- переміщувати об'єкт точка по робочій області при утримуванні на ній лівої кнопки миші;
- виклику індивідуального контекстного меню в залежності від об'єкту під курсором при натисненні правої кнопки миші.

Додаток

(текст модулю NGMath)

```
unit NGMath;
interface

uses SysUtils;
const
  NumOfConst  = 27;
  AllErr      = 1e+100;
  IlFfuncCall = 2e+202;
  Zero        = 1e+101;
  Overflow    = 3e+303;
type
  RealArray=array[0..numofconst,1..2] of Extended;

  var Ln10 : Extended;


function Minimum(A, B: Extended): Extended;
function Maximum(A, B: Extended): Extended;
function SValue(X: Extended): String;
procedure Convertor(var s: string; var consts:
     realarray; var error: boolean);
function Sign(x:extended):integer;
function TrueFormula(S: String): Boolean;

implementation

function Minimum(A, B: Extended): Extended;
begin
  if A<B then Minimum:=A else Minimum:=B
end;

function Maximum(A, B: Extended): Extended;
begin
  if A<B then Maximum:=B else Maximum:=A
end;

function SValue(X: Extended): String;
  var s,ss:string;
```

```
      i,point:byte;
begin
  if abs(x)>10000 then
    begin
      str(x,ss);
      if ss[1]=' ' then Delete(ss,1,1);
      s:=copy(ss,1,12)+'E+'+copy(ss,length(ss)-1,2);
      i:=12;
      while (i>4) and (s[i]='0') do
        begin
          delete(s,i,1);
          dec(i)
        end;
    end
    else
      begin
        if (abs(x)<0.00001) and (x<>0) then
          begin
            str(x,ss);
            if ss[1]=' ' then Delete(ss,1,1);
            s:=copy(ss,1,12)+'E'+copy(ss,length(ss)-
     4,5);
            i:=12;
            while (i>4) and (s[i]='0') do
              begin
                delete(s,i,1);
                dec(i)
              end;
          end
        else
          begin
            str(x:1:9,s);
            point:=pos('.',s);
            i:=length(s);
            while (i>point) and (s[i]='0') do
              begin
                delete(s,i,1);
                dec(i);
              end;
            if s[i]='.' then delete(s,i,1);
          end;
      end;
  svalue:=s
end;
```

```
procedure convertor(var s:string;var
     consts:realarray;var error:boolean);
  var sl:string;
      op,countop:byte;
procedure coder;
  var i,clos,open,constcount:byte;
procedure fuin;
  const funames:string=
        'Y   T   X   F   SIN COS TG  CTG ASINACOSATG
     ACTGEXP LG  LN  ABS SQRTLOG PI  INT ';
        fucode:string=#29+#30+#30+#30'ABCDFGHIJKLMNOPR';
        {
        'Y   T   X   F   SIN COS ';
        fucode:string=#29+#30+#30+#30'AB';
        }

  var ss:string;
      n:byte;
begin
  ss:='';
  while (s[i] in ['A'..'Z']) and (i<=length(s)) do
    begin
      ss:=ss+s[i];
      inc(i)
    end;
  n:=pos(ss,funames);
  if (n>0) and ((n mod 4)=1) then
    begin
      sl:=sl+fucode[(n div 4)+1];
      dec(i);
      if sl[length(sl)]='P' then
        begin
          consts[constcount]:=Pi;
          inc(constcount);
          sl[length(sl)]:=chr(constcount-1);
        end;
    end
      else error:=true;
end;
procedure valuein;
  var ss:string[20];
      e:integer;
      d:extended;
```

```
procedure dit;
begin
  while  (s[i] in ['0'..'9','.']) and (i<=length(s)) do
    begin
      ss:=ss+s[i];
      inc(i)
    end
end;
begin
  ss:='';
  dit;
  if (i<=length(s)-2) and (s[i]='E') and (s[i+1] in
     ['-','+'])
    then
      begin
        ss:=ss+s[i]+s[i+1];
        i:=i+2;
        dit
      end;
  val(ss,d,e);
  if e>0 then error:=true else begin
  consts[constcount]:=abs(d);
  inc(constcount);
  sl:=sl+chr(constcount-1);
  dec(i); end
end;

function log(s:string):string;
  var s1:string;
      pn,pe,z,k:longint;
begin
  while pos('LOG',s)>0 do
    begin
      pn:=pos('LOG',s);
      if s[pn+3]<>'(' then
        begin
          error:=true;
          exit;
        end;
      k:=1;pe:=pn+3;
      repeat
        inc(pe);
        if s[pe]='(' then inc(k);
        if s[pe]=')' then dec(k);
```

```
      until (k=0) or (pe=length(s));
      if k<>0 then
        begin
          error:=true;
          exit;
        end;
      s1:=copy(s,pn+4,pe-pn-4);
      delete(s,pn,pe-pn+1);
      s1:=log(s1);
      z:=pos(',',s1);
      if z=0 then begin
                    error:=true;
                    exit;
                  end;

     s1:='LN('+copy(s1,z+1,length(s1)-z)+')/LN('+copy(s1
     ,1,z-1)+')';
      insert(s1,s,pn);
    end;
  log:=s;
end;

begin
  constcount:=1;
  open:=0;
  clos:=0;
  sl:=#0+'"';
  consts[0]:=0;
  error:=false;
  s:=log(s);
  i:=1;
  while i<=length(s) do
    begin
      case s[i] of
      'A'..'Z':fuin;
      '0'..'9','.':valuein;
      '(':begin open:=open+1; sl:=sl+' ' end;
      ')':begin clos:=clos+1; sl:=sl+'!' end;
      '+':sl:=sl+'"';
      '-':sl:=sl+'#';
      '*':sl:=sl+'$';
      '/':sl:=sl+'%';
      '^':sl:=sl+'&'
    end;
```

```
    if clos>open then error:=true;
    if ConstCount>NumOfConst then error:=true;
    inc(i)
  end;
  if clos<>open then error:=true;
end;
procedure minus;
  var i:byte;
begin
  if sl[3]='#' then insert(#0,sl,3);
  for i:=1 to length(sl)-2 do
    if (sl[i]=' ') and (sl[i+1]='#')
      then
        insert(#0,sl,i+1)
end;
procedure anothererrors;
  var i,a:byte;
begin
  op:=0;
  for i:=1 to length(sl)-1 do
    begin
      a:=byte(sl[i]);
      if ((a<=Succ(NumOfConst+1)) and (sl[i+1]<'!'))
     or
         ((a<=Succ(NumOfConst+1)) and (sl[i+1]>='A'))
     or
         ((a=32) and (sl[i+1] in ['"'..'&']))
     or
         ((sl[i] in ['"'..'&']) and (sl[i+1]='!'))
     or
         ((a>64) and (sl[i+1]<>' '))
     or
         ((sl[i] in ['"'..'&']) and (sl[i+1] in
     ['"'..'&'])) or
         ((sl[i]=' ') and (sl[i+1]='!'))
     or
         (sl[length(sl)] in ['"'..'&','A'..'Z'])
     or
         ((sl[i]='O') and (ord(sl[i+1])>NumOfConst))
     or
         ((sl[i]='!') and (not (sl[i+1] in
     ['!','"'..'&'])))
         then error:=true;
      if a>=34 then op:=op+1
```

```
    end
end;
procedure analis;
  var open,l,r,lr:byte;
      tmpl,tmpr:char;
procedure searcho(a:byte;var b:byte);
begin
  b:=a+1;
  while (sl[b]<'"') and (b<length(sl)) do
    b:=b+1;
end;
function rang(a:byte):word;
  var i:byte;
begin
  open:=0;
  for i:=1 to a do
    begin
      if sl[i]=' ' then open:=open+1;
      if sl[i]='!' then open:=open-1;
    end;
  if a>=length(sl)
    then
      rang:=0
    else
      begin
        case sl[a] of
          '"','#':rang:=40*open+1;
          '$','%':rang:=40*open+2;
          '&':rang:=40*open+3;
          'A'..'Z':rang:=40*open+4
        end
      end
end;
procedure oper(var lr:byte);
  var l:byte;
begin
  searcho(0,l);
  repeat
    lr:=l;
    searcho(l,l)
  until rang(lr)>=rang(l)
end;
procedure searchl(lr:byte;var l:byte);
begin
```

```
  l:=lr-1;
  while sl[l]='!' do
    l:=l-1
end;
procedure searchr(lr:byte;var r:byte);
begin
  r:=lr+1;
  while sl[r]=' ' do
    r:=r+1
end;
begin
  oper(lr);
  case sl[lr] of
    '"'..'&':begin
               searchl(lr,l); tmpl:=#0;
               if sl[l]<>#31 then s:=s+sl[l] else
    tmpl:=#31;
               searchr(lr,r); tmpr:=#0;
               if sl[r]<>#31 then s:=s+sl[r] else
    tmpr:=#31;
               if (tmpl<>#31) and (tmpr=#31) then
    s:=s+#255;
               s:=s+sl[lr];
               delete(sl,2*l-lr+1,2*r-2*l-1);
               insert(#31,sl,2*l-lr+1)
             end;
    'A'..'T':begin
               searchr(lr,r);
               if sl[r]<>#31 then s:=s+sl[r];
               s:=s+sl[lr];
               delete(sl,lr,2*r-2*lr);
               insert(#31,sl,lr)
             end
  end;
  countop:=countop+1;
  if countop<op then analis
end;
begin
  countop:=0;
  coder;
  s:='';
  if not error then minus;
  if not error then anothererrors;
  if not error then analis
```

```
end;

function sign(x:extended):integer;
begin
  if Abs(x)>=AllErr then
    sign:=2
  else
    begin
      if x>0 then
        sign:=1
      else
        begin
          if x<0 then
            sign:=-1
          else
            sign:=0;
        end;
    end;
end;

function TrueFormula(S: String): Boolean;
  var St :  String;
      Arr:  RealArray;
      Err:  Boolean;
begin
  St:=UpperCase(S);
  Convertor(St, Arr, Err);
  TrueFormula:= not Err;
end;

initialization
  Ln10:=Ln(10);
end.
```

Література

1. Агафонов В.Н. Объектно-ориентированное программирование и абстрактные типы данных // Программирование. - №6, 1990. - С.27-32.

2. Бартків А.Б., Гринчишин Я.Т., Ломакович А.М., Рамський Ю.С. Turbo Pascal: Алгоритми і програми: чисельні методи в фізиці і математиці: Навч. посібник. – К.: Вища школа., 1992. – 247 с.

3. Буч Г. Объектно-ориентированное проектирование с примерами применения: Пер. с англ., совместное издание фирмы “Диалектика” г. Киев и АО “И.В.К.” г. Москва, 1992.

4. Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир. – 1985.

5. Волинський В.П., Козакова Г.О. Методичні рекомендації до використання педагогічних програмних засобів у навчальному процесі. – К.: НПУ ім. М. П. Драгоманова, 2007. – 59 с.

6. Востроκнутов И.Е. Оценка визуальных сред на экране монитора или почему болят глаза при работе на компьютере. // Информатика и образование. - 2002. - №-1.- С. 64 - 67.

7. Дудик М.В., Рамський Ю.С., Цибко Г.Ю. Основи програмування: Навчальний посібник для студентів вищих навчальних закладів фізико-математичних та індустріально-педагогічних спеціальностей. – К.: Міленіум, 2005. – 168 с.

8. Жалдак М.І., Горошко Ю.В., Вінниченко Є.Ф. Математика з комп’ютером: посібник для вчителів. // К.: РННЦ «ДІНІТ», 2004. – 255с.

9. Калверт Чарли, Калверт Марджори, Кастер Джон, Сворт Боб. Borland Kylix. Руководство разработчика.: Пер. с англ. – М.: Вильямс, 2002. – 880 с.

10. Керман, Митчел, К. Программирование и отладка в Delphi. Учебный курс.: Пер. с англ. – М.: Издательский дом «Вильямс», 2002. – 672 с.

11. Уэзерелл Ч. Этюды для программистов: Пер. с англ.- М.: Мир, 1982.- 288 с. ил.

12. Фаронов В. В. Delphi 5. Руководство программиста. - М.: Нолидж, 2001. - 880 с., ил.

13. Хоггер К. Введение в логическое программирование/ Пер. с англ.-М.:Мир,1998.-348 с.

14. Bobrow D. G., If Prolog is the answer, what is the question. // Fifth Generation of Computer Systems, pages 138—145, Tokyo, Japan, November 1984. Institute for New Generation Computer Technology (ICOT), North-Holland.

15. Budd T. A., Multy-Paradigm Programming in LEDA. Addison-Wesley, Reading, Massachusets, 1995.

16. Friedman L. W., Comparative programming languages: generalizing the programming function. Prentice Hall, 1991, page 188.

17. Shriver B. D., Software paradigms. IEEE Software, 3(1):2, January 1986.

18. Spinellis D. D, Programming paradigms as object classes: a structuring mechanism for multiparadigm programming. PhD thesis, University of London, London SW7 2BZ, United Kingdom, February 1994.

19. Wegner P., Concepts and paradigms of object-oriented programming. IEEE Software, 1(1): 7—87, August 1990.

Зміст

Вступ 3
1. Поняття парадигми та технології програмування . . 6
2. Семантика мов програмування 9
3. Вимоги до педагогічних програмних засобів . . . 10
4. Критерії якості ППЗ 14
5. Особливості об’єкто-орієнтовного програмування в середовищі Lazarus 15
5.1. Концепції ООП 16
5.2. Обробка виключних ситуацій 18
5.3. Модуль даних та його структура 20
5.4. Створенная нових класів 22
5.5. Присвоєння об’єктів 27
5.6. Наслідування типів 27
5.7. Підміна, перевизначення та перезавантаження методів . 30
5.8. Наслідування статичних методів 32
5.9. Віртуальні методи і поліморфізм 32
5.10. Раннє та пізнє зв‘язування 33
5.11. Абстрактні методи 34
5.12. Реалізація відношення агрегації 36
5.13. Інкапсуляція 37
5.14. Властивості 40
6. Gran1
6.1 Теоретичні відомості про структуру ПЗ Gran1 . . 44
6.2. Завдання для студентів 87
7. Gran2D
7.1. Теоретичні відомості про структуру ПЗ Gran2D . . 93
7.2. Завдання для студентів 123
Додаток (текст модулю NGMath) 132
Література 141

Навчально-методичне видання

ТЕОРІЯ І МЕТОДИКА РОЗРОБКИ ПЕДАГОГІЧНИХ ПРОГРАМНИХ ЗАСОБІВ

Укладачі:

Горошко Юрій Васильович - кандидат педагогічних наук, доцент Чернігівського національного педагогічного університету ім. Т.Г.Шевченка.

Костюченко Андрій Олександрович - асистент Чернігівського національного педагогічного університету ім. Т.Г.Шевченка

Рецензенти:

Цибко Ганна Юхимівна - кандидат педагогічних наук, завідувач кафедри інформатики та ОТ, доцент Чернігівського національного педагогічного університету ім. Т.Г.Шевченка.

Гур'єв Володимир Іванович - кандидат технічних наук, завідувач кафедри економічної кібернетики та інформатики, доцент Чернігівського державного інституту економіки та управління.

Підписано до друку 11.01.2011р. Формат 60x84/16
Папір офсетний. Гарнітура Таймс. Друк на Ризографі.
Ум.друк.арк. 9,0.
Тираж 50 прим. Зам. № 013.
Віддруковано в авторській редакції

Виготовлення ФОП "Єрмоленко О.М."

Свідотство про внесення субєкта видавничої справи до Державного реєстру

видавців, виготівників і розповсюджувачів видавничої продукції.

Серія ЧГ № 10 від 17 липня 2009 року