Т.А. Кирик
Рівненський державний гуманітарний університет

Еволюційні процеси у програмуванні

Комп'ютерне програмування виступає особливим видом діяльності людини. З одного боку його розуміють як мистецтво (Д. Кнут), з іншого воно підпорядковане законам логіки. Програмування тісно пов'язане з математикою, хоча за своєю природою значно відрізняється від неї. Результат програмування не абстрактний, а цілком реально існуючий продукт, і це безперечно наближує програмування до інженерної діяльності. Під час створення програмного продукту, крім основної мети - реалізації алгоритму, часто є необхідною і оптимізація ефективності його роботи. Математика, інженерія і програмування сьогодні йдуть поруч.

На сьогодні можна виділити кілька мов програмування, які впливають на програмну індустрію: Visual Basic, C, C++, Object Pascal, Java, C#, Eiffel, Оберон та ін. Кожну з них можна використовувати в тій чи іншій мірі в якості алгоритмічної мови. Алгоритмічна мова, взагалі кажучи, відрізняється від мови програмування. Еволюція мов програмування привела до того, що поняття алгоритмічної мови замінилося поняттям мови програмування, яке далі розчинилося у середовищі реалізації (середовище трансляції, виконуюча система, набір стандартних бібліотек)[3]. При реалізації алгоритму в межах певного середовища багато уваги вимагає вивчення середовища, а не тільки мови програмування, яку воно підтримує. За технічними деталями часто втрачається розуміння ідеї алгоритму. А без цього перенести алгоритм на іншу мову та застосувати його буде доволі тяжким завданням. Разом з тим алгоритмічна мова не повинна бути абстрактною, адже перевірку розв'язків часто потрібно виконувати дослідним шляхом. Тобто алгоритмічна мова має реально існувати, вона повинна спиратися на невелику кількість базових понять і бути достатньо виразною для подання різного роду алгоритмів. Мова повинна бути лаконічною, доступною на більшості поширених комп'ютерних платформ[3].

Більшість алгоритмічних мов, таких як С, Паскаль та ін., були створені у 60-70 рр. XX ст. Протягом цього часу з'являлися нові мови, принесені ними ідеї були закладені у сучасні мови програмування (так сталося із концепцією об'єктно-орієнтованого програмування).

Якщо порівнювати такі мови програмування, як С, Паскаль, Java, то слід сказати, що створювалися вони в різний час і з різною метою (Паскаль - для навчання алгоритмізації і програмуванню, С - для професійного програмування), доповнювалися в процесі практичного використання різними корисними інструкціями, і в кінцевому результаті прийшли майже до повної тотожності. Близькі за змістом конструкції різних мов програмування відрізняються в основному способом запису. Тобто вони мають подібний зміст (семантику), але різний порядок слідування компонент (синтаксис) і різні ключові слова (лексику). Звідси можна було б зробити висновок, що різні мови програмування надають програмісту однакові можливості для реалізації алгоритмів. У [1] описується на семантичному рівні "універсальна" мова програмування. Опис такої єдиної семантичної бази надає можливості створення "універсального компілятора", який міг би працювати з будь-якою існуючою мовою, яка має спільну з іншими семантичну базу. Спільна семантика дозволить користуватися одним синтаксичним аналізатором і єдиним генератором виконуваного коду для існуючих мов. Лексика ж усіх сучасних мов ідентична. Але кожна мова містить конструкції, характерні тільки для неї. Так звану "область перетину" для мов С, Паскаль, Java складає оператор циклу з параметром. Але навіть для цієї конструкції характерне те, що область перетину охоплює тільки цикли з параметром +1, -1. Таким чином всі інші цикли з параметром будуть віднесені до "області об'єднання". Так, в область об'єднання потрапляють тип множини у Паскалі, директиви препроцесора С. Аналіз "області перетину" сучасних мов програмування дає відповідь на запитання, які конструкції є найбільш життєздатними. Звичайно, ряд конструкцій із області об'єднання можуть бути використані, але це покаже тільки практика.

Якщо семантика мови буде визначена чітко, то з'являється можливість відмовитися від тексту в якості інструмента введення програми, на зміну текстовому редактору прийде структурний, який дозволить зберігати готові структури (типи даних, оператори). Перехід до структурного редактора надасть можливості on-line компіляції програми (код оператора створюється під час його введення), створення ж виконуваного файла стане компонуванням відкомпільованих частин програми. Перехід на структурний редактор і відмова від тексту як носія програми дозволить зробити наступний крок після створення візуальних систем програмування. Візуальні системи (Delphi, Visual C++), вже сьогодні дозволяють створювати інтерфейс програми в інтерактивному режимі. Але при необхідності запрограмувати якийсь алгоритм знову потрібно працювати з текстом програми.

Широкий розвиток комп'ютерної індустрії, впровадження обчислювальної техніки в різні сфери діяльності людини не може не приводити до появи нових завдань перед розробниками програмного забезпечення. Суть будь–якої кризи у програмуванні – протиріччя між потребами в інформаційних системах і рівнем досконалості технологій їх створення і супроводу. Настає момент, коли технології розробки і супроводу прикладних систем стають недостатніми для того, щоб розв'язувати задачі такого рівня складності, який допускають апаратні засоби, також зростають вимоги до якості програмного забезпечення, оскільки зростає складність і значущість виконуваних ними функцій.

Перші обчислювальні машини програмувалися на апаратному рівні шляхом комутації тригерів з'єднувальними кабелями. Пізніше програму вводили перемиканням тумблерів на пульті. Перша

криза інформаційних систем була пов’язана з відставанням продуктивності праці програмування на пультах від потреб у обчисленнях. Так з’явилося програмування на асемблері. З часом цього також стало недостатньо, і з’явилися мови програмування високого рівня. Спочатку всі зусилля розробників були зосереджені на розробці спеціалізованих чи проблемно-орієнтованих мов програмування (Кобол, Сімула, Аспід та ін.). Ці мови були розраховані на розв’язування задач певного типу, що спростило процес розробки програмних продуктів для застосувань у конкретній предметній галузі. Але для спеціалізованих мов характерною є проблема обмеженості сфер застосування. Поступове ускладнення задач, кодів, збільшення об’ємів програмних засобів, що розроблялися, спричинило виникнення нового стилю і техніки програмування. Починаючи із 70-х рр. XX ст. формуються основи методології і теорії програмування (Н.Вірт, Е. Дейкстра, Д. Гріс). Тобто третя криза була розв’язана появою структурного програмування, як найбільш продуктивного стилю програмування. Розвиток концепції структуризації привів до необхідності структуризації даних, що і визначило у мовах програмування появу механізмів абстрагування типів даних (Модула). Абстрагування типів даних лягло в основу технології модульного програмування. У модульному програмуванні використовується принцип обмеження доступу до даних: програма ділиться на модулі так, щоб дані, що опрацьовуються модулем, були сховані у ньому.

Четверта криза була викликана необхідністю масового створення систем з графічним інтерфейсом користувача. Ця криза була розв’язана появою об’єктно-орієнтованого стилю програмування. Об’єктно-орієнтований метод програмування найбільш повно реалізує технологію структурного та модульного програмування, є зручним способом для створення складних ієрархічних систем, які допускають розширення. Об’єктно-орієнтована парадигма пропонує новий підхід до розробки програмного забезпечення.

Основна ідея ООП полягає у передаванні повідомлень до об’єктів. Для цього потрібно, щоб об’єкти визначалися разом із повідомленнями, на які вони будуть реагувати. Такий спосіб розробки програм не притаманний процедурному програмуванню, для якого характерне визначення структур даних, що потім передаються у підпрограми як параметри. Список сучасних об’єктно–орієнтованих мов складають Object Pascal, C++, C#, Java [5] та ін. Розглянемо, як реалізовані основні компоненти об’єктно–орієнтованого програмування у цих мовах. Безумовно, головним компонентом виступає *об’єкт,* який є структурою, що об’єднує елементи даних і операції над ними. Кожен об’єкт є екземпляром певного класу. Об’єкти є не тільки порціями даних, а й джерелом дії. Клас є описом поведінки представника класу, типом даних. Клас визначає спосіб взаємозв’язків його екземплярів із зовнішнім світом. Якщо такі мови як Object Pascal, C++, Java, Basic у початковому їх варіанті процедурні, тобто надбудова засобів ООП була виконана пізніше, то C# є мовою об’єктно-орієнтованого програмування. У першій групі мов основні типи відокремлені від об’єктних. У мові C# всі типи, і вбудовані, і визначені користувачем, породжені від базового класу object [4].

Класи можуть бути пов’язані один з одним співвідношеннями спадкування, контейнера. Механізм спадкування дозволяє розширювати опис існуючих об'єктів. Спадкування є одним з механізмів, за допомогою якого один клас об'єктів може включатися в роботу іншого класу об'єктів. Спадкування надає могутній механізм моделювання відношень, що існують у реальному світі. Відношення спадкування між класами означає, що похідний клас успадковує всі властивості та поведінку базового класу і має додаткові, специфічні властивості та методи. Відношенням спадкування користуються, якщо спосіб зв’язку між класами має вигляд “клас 2 *є* класом 1” (наприклад, “лялька *є* іграшкою” чи “лялька *належить* до об’єктів типу іграшка”). Тобто родовий (базовий) клас виступає узагальненням похідних від нього класів. Концептуально кожен екземпляр похідного класу виступає представником і родового класу теж. Про контейнерне відношення між класами говорять, якщо об'єкт класу class1 містить об'єкт класу class2 , тобто деяке поле класу class1 має тип class2 (наприклад, “літак *має* двигун”).

Мови Object Pascal, Java, C# підтримують модель одиночного спадкування (в якості базового класу може виступати тільки один клас). У C++ реалізована модель множинного спадкування. В мові Basic (середовище реалізації Visual Basic) передбачено неповну підтримку механізмів об’єктно-орієнтованого програмування, допускається контейнерне відношення між класами, але не відношення спадкування. У Visual Basic.NET цей недолік усунутий запровадженням механізму спадкування реалізації класу, спадкування форми.

Реакцію об’єкта на зовнішні повідомлення, поведінку об’єкта визначають методи (підпрограми, описані як частина класу). При порівнянні різновидів методів у різних мовах програмування, простежується спільність (тотожність) у типових методах. Так методи-процедури, методи-функції у Object Pascal є підпрограмами, які викликаються для певного екземпляру класу. Відповідні дії у C++ виконують функції-члени класу. Класові процедури та функції Object Pascal за призначенням відповідають статичним функціям C++. Такі методи оголошуються як частина класу, але можуть бути використані безвідносно до екземплярів класу. Якщо звичайні методи отримують покажчик на певний екземпляр класу, то класові методи викликаються за допомогою посилання на клас. Крім того класи можуть мати спеціальні методи-конструктори, які служать для створення об’єктів та ініціалізації їх полів. Конструктори функціонують подібно до класових методів. Тобто вони викликаються за допомогою посилання на клас і повертають значення, яке є посиланням на створеного представника класу. Спеціальні методи іншого виду – деструктори, служать для ліквідації представника класу. Якщо поля об’єкта розміщені у відкритій частині класу, то можливий прямий доступ до них. Але, взагалі кажучи, такий спосіб доступу не є найкращим. Адже у такому випадку деталі реалізації класу стають відомими, тобто втрачаються переваги інкапсуляції. Метод-аксесор, який використовується для доступу до поля (можливо, коректної його зміни) має такі переваги як надійність даних, цілісність посилань, передбачені побічні ефекти. Так, надійний стан об’єкта може

бути забезпечений за допомогою перевірки значення поля на допустимість. При необхідності виконання певних дій при звертанні до поля достатньо відповідні дії реалізувати у методі, будемо мати передбачуваний побічний ефект. Якщо доступ до об'єкта здійснюється тільки за допомогою методів, то при змінах у реалізації класу достатньо буде змінити лише реалізацію методів звертання до полів.

Видимість полів і методів у Object Pascal, C++, Java визначається належністю до тої чи іншої частини класу. Найвищий рівень обмеження видимості елементів забезпечується закритою частиною класу (private). Вони видимі тільки у визначенні класу. Захищені (protected) елементи можуть бути видимі у даному класі та у похідних класах, але зовні визначення класу вони не доступні. Відкрита (public) частина класу служить інтерфейсом із зовнішнім світом, тобто елементи даних чи підпрограми, описані тут, доступні як у тілі класу, так і за його межами.

Важливою компонентою об'єктно-орієнтованої технології програмування є поліморфізм ("множинність форм"). Поліморфізм - забезпечення різних реакцій методів на повідомлення. Конкретна форма реакції на повідомлення може визначатися ще на етапі трансляції програми (статичний поліморфізм). Якщо об'єкт пов'язується із повідомленням під час виконання програми (пізнє зв'язування), то кажуть про динамічний поліморфізм. В C++ поліморфізм проявляється у підтримці механізмів перевантаження функцій (методів класу у тому числі), операцій, віртуальних функцій. Механізми перевантаження функцій, віртуальних функцій притаманні й іншим об'єктно-орієнтованим мовам. Виклик віртуального методу допомагає реалізувати різні варіанти поведінки при використанні у різних класах методу з одним і тим же іменем. Віртуальний метод дозволяє похідному класу модифікувати поведінку методу, заданого у класі-попереднику. Тобто поліморфізм є абстракцією поведінки. Можна спричинювати певну поведінку за іменем якогось екземпляра класу, не знаючи точно, яка саме реалізація методу при цьому буде викликана, і навіть не знаючи типу, до якого цей об'єкт належить. Процес визначення реалізації методу виконуватиметься динамічно, тобто один і той же машинний код може викликати *метод1* похідного класу *клас1* на одному проході і *метод2* похідного класу *клас2* на другому, якщо при різних проходах використовуються різні екземпляри об'єктів.

Всі об'єктно-орієнтовані мови підтримують механізм опрацювання виключних ситуацій. Виключення є способом передавання повідомлень про помилку часу виконання у ту частину програми, яка передбачена для опрацювання відповідної ситуації. У традиційному способі опрацювання помилок часу виконання введені значення до початку їх опрацювання перевіряються на коректність значень. Так званий "захисний" код може стати доволі об'ємним, адже перевірку на допустимість доведеться виконувати чи не після кожного оператора. В сучасних об'єктно-орієнтованих мовах пропонуються спеціальні синтаксичні конструкції для розв'язування таких проблем. Код, який може генерувати помилку, відокремлений від коду, який реагує на помилку. Код, який виявляє помилку, *викидає* виключення, що служить сигналом про стан помилки. Код який реагує на такий сигнал, називають *обробником* виключення. Коли потрібний обробник знайдений, він отримує управління, як правило, не повертаючись до коду, що викликав помилку. Після виконання коду обробника помилок або продовжується виконання програми, або здійснюється пошук іншого обробника виключення. Управління виключеннями є засобом підвищення надійності програм.

Паралельно з розвитком процедурного стилю програмування на початку 70-х років XX ст. з'явились непроцедурні мови логічного програмування і програмування штучного інтелекту (Лісп, Пролог). Так, Пролог-програма є набором *тверджень* та *правил*, які задають відношення між об'єктами, і *мети* (цільового запиту). Процес виконання програми трактується як процес встановлення загальної значимості логічної формули за правилами, встановленими семантикою тієї чи іншої мови. Результат обчислень є побічним продуктом процедури висновку. Такий метод являє собою повну протилежність програмуванню якою-небудь із процедурних мов. Сьогодні Пролог-мова, призначена для програмування прикладних програм, в яких використовуються засоби і методи штучного інтелекту, створення експертних систем і подання знань.

Причина теперішньої п'ятої кризи - масове впровадження інформаційних технологій (ІТ) у бізнес та побут і широке поширення розподілених систем. Нинішній час - це час повсюдного використання інформаційних систем (ІС). Потреба в індивідуальних програмах складає мільйони екземплярів. Поки що потреби користувачів частково задовольняються за рахунок типізованих програм чи пакетів, в яких передбачено деякі можливості налагодження. Однак розв'язування трохи складніших задач вимагає навичок і умінь програміста, якими переважна більшість користувачів не володіють. Вихід з цієї ситуації - у відкритих компонентних технологіях розробки ІС і поширенні компонентів через загальнодоступні глобальні мережі. Імовірно, у перспективі побудова прикладних програм буде не розробкою, а "складанням" її з готових компонентів, і займатися цим буде не програміст (який вміє крок за кроком "пояснити машині", як розв'язувати задачу), а кваліфікований користувач, який вміє сформулювати, що йому потрібно «на виході», у термінах, "зрозумілих" системі управління компонентами. Центр ваги при розробці переміститься з програмування на проектування.

При проектуванні задачі важливими є коректна її постановка, формулювання критеріїв, яким має відповідати розв'язок задачі. У 80-х роках XX ст. дослідження причин невдач при реалізації великих програмних проектів показало, що близько 56% помилок допускаються на етапі формулювання вимог до програми. При цьому витрачається в середньому 82% усіх зусиль, витрачених колективом на усунення помилок проекту, у той час як на етапі кодування програм допускається відповідно 7% помилок і витрачається 1% зусиль на їхню ліквідацію. У цей час формулюється теза про те, що метою програмування є не породження програми як такої, а створення технологічних умов, коли програмне забезпечення, що розробляється, легко адаптується до нових

обставин і нового розуміння розв'язуваної задачі . Р. Хеммінг так формулює цю тезу: "... обчислювальна практика вимагає постійного вивчення досліджуваної задачі не тільки перед організацією обчислень, але також у процесі його розвитку й особливо на тій стадії, коли отримані числа ... витлумачуються мовою первісної задачі".

Перелічені вище причини привели в середині 80-х років XX ст. до усвідомлення необхідності реалізації інтегрованого оточення підтримки всього життєвого циклу програмного засобу і в першу чергу етапу проектування, що обумовило появу інструментальних засобів автоматизації проектування програмних систем (CASE-технологій).

Спочатку CASE-засоби були орієнтовані на розв'язування задач автоматизованого збирання відомостей з предметної галузі і проектування майбутнього програмного забезпечення(ПЗ), що дозволяло б заощаджувати час при створенні ПЗ за рахунок більш ретельного аналізу вихідних вимог і кращого початкового планування програми. Згодом у CASE-засобах другого покоління цілком чи частково були автоматизовані такі важливі складові життєвого циклу ПЗ, як моделювання відомостей з предметної галузі; програмування; тестування, налагодження ПЗ і вимірювання якості; підтримка документування; супровід. Застосування CASE-інструментів дозволяє значною мірою знизити трудомісткість створення ПЗ, а в окремих випадках замінити програмування автоматичним синтезом програм.

Таким чином, розвиток методів автоматизації розробки ПЗ відбувається на різних основах (модульне програмування, об'єктно-орієнтований підхід, логічне програмування, CASE-технології), що так чи інакше розвивають концепції структуризації в програмуванні. Структуризація сприяє проведенню ефективної декомпозиції проекту, що дозволяє одержувати як цілісне уявлення про ПЗ, так і його деталі. Однак, незважаючи на численні розробки в цій галузі, в цілому проблема автоматизації розробки ПЗ залишається нерозв'язаною з багатьох причин як методологічного, так і практичного характеру.

Удосконалення технічних засобів відображення повідомлень привело до графічного підходу розв'язування проблеми автоматизації розробки ПЗ, що базується на ідеї залучення візуальних форм подання програм, у більшій мірі відповідних образному способу мислення людини. Застосування графічних методів обіцяє кардинально підвищити продуктивність праці програміста. Крім того, графічна форма запису порівняно з текстовим поданням програм забезпечує більш високий рівень їхньої структуризації, дотримання технологічної культури програмування, пропонує більш надійний стиль програмування.

Сьогодні відома досить велика кількість вдалих інструментальних засобів візуального програмування. Насамперед це стосується візуальних засобів розробки екранних форм, меню й інших елементів програми (Visual BASIC, Delphi, Visual C++, Builder C++ і т.д.), засобів автоматизації проектування програмного забезпечення (CASE-засобів), засобів швидкої розробки прикладних програм для інформаційних систем (Visual FoxPro), текстових і графічних редакторів, видавничих систем і т.д.

Як бачимо, принципи розв'язання протиріччя у програмуванні були одними і тими ж: перехід від унікального коду; від машинно- чи системо-залежного до позбавленого таких рис; від покрокових інструкцій до опису предметної галузі. Рівні цих переходів на різних етапах були неоднаковими, але прослідковується загальна тенденція підвищення рівня абстракції й уніфікації.

ЛІТЕРАТУРА

1. Андреев А. Эволюция современных языков программирования. // Мир ПК. – 2001. – №3. – С. 56 -63.
2. Артемов Д. Visual Foxpro 5.0: новые возможности. // Мир ПК. – 1997. – №2. – С.68 -73.
3. Богатырев Р. О программировании и выборе языка для представления алгоритмов. // Мир ПК. – 2001. – №6. – С.50 -51.
4. Бодров В. Введение в C#: классы. // Мир ПК. – 2001. – №7. – С.122-128.
5. Галактионов В. Linux, Java, 3-D графика. // Мир ПК. – 2001. – №12. – С.60- 64.
6. Кунц В., Морзе Н. Основи програмування у середовищі Visual Basic. //Інформатика. – 2002. – №21-24.
7. Отенко В. Объектно–ориентированное программирование на языке ассемблера. http: //www.comizdat.com.
8. Прехельт Л. Эмпирическое сравнение семи языков программирования. // Открытые системы. –2000. – №12. http://www.osp.ru/os/2000/12/045.htm