УДК 681.3.06

М.М. Колодницький, к.т.н., проф.
Житомирський інженерно-технологічний інститут

ПРОБЛЕМА МОДЕЛЮВАННЯ ТА ДОКУМЕНТУВАННЯ ОБ'ЄКТНО-ОРІЄНТОВАНИХ ПРОГРАМ ДЛЯ MICROSOFT® WINDOWS® ("Критика UML-ізма")

Метою даної роботи є привертання уваги розробників програмного забезпечення, а також наукових дослідників до проблеми використання сучасних технологій моделювання та документування програм. На прикладі однієї з таких технологій, а саме Unified Modeling Language (UML), показано, що вона має не тільки позитивні риси, але й певні недоліки, і, зокрема, при її застосуванні у випадку об'єктно-орієнтованого програмування (ООП) з використанням системної бібліотеки класів. Наведено приклад простої програми для MICROSOFT® WINDOWS®, для якої застосування UML, на думку автора, є не досить ефективним і не досягає основної мети UML – моделювання програми, тобто представлення її у спрощеному виді для кращого її розуміння. Наводиться альтернативний спосіб візуального представлення такого класу програм. Проведений в роботі аналіз та наведені результати можуть допомогти розробникам об'єктно-орієнтованих програм і, зокрема, програм для MICROSOFT® WINDOWS® знайти більш ефективний шлях вирішення проблеми їх моделювання та документування.

Вступ

Будь-яка технічна система має певну послідовність станів її існування – так званий життєвий цикл (англ. – *life cycle*) [1]. В найбільш загальній формі життєвий цикл (ЖЦ) технічної системи складається із етапів:

1) *визначення нової проблеми* або *постановка задачі*;
2) *проектування* та *створення* системи;
3) *експлуатація* системи, включаючи етапи впровадження, безпосереднього використання, ремонту, модифікації та супроводження;
4) *ліквідація* системи або її розвиток (перетворення) у нову систему з новими, більш прогресивними характеристиками.

Програмні системи (англ. – *software*) є також технічними системами, але вони мають ряд суттєвих відмінностей, які, в свою чергу, відображаються на структурі та змісту їх життєвого циклу. Розглядом таких особливостей, зокрема, займається дисципліна *програмна інженерія* (англ. – *software engineering*), що виникла порівняно недавно [2] і яка на сьогодні напрацювала досить багатий арсенал принципів, методів та технологій [3–6]. В той же час, і сьогодні в програмній інженерії існують задачі, які потребують свого вирішення. Серед таких задач не останнє місце займає проблема моделювання та документування програм, особливо для випадку об'єктно-орієнтованих програм і, зокрема, програм для MICROSOFT® WINDOWS®, що побудовані з використанням системної бібліотеки класів. Розгляду такої проблеми і присвячена дана робота.

В роботі стисло показана структура етапів життєвого циклу програмної системи, після чого на основі загального означення моделі звертається увага на специфіку при побудові моделей програмних систем на різних етапах ЖЦ, і далі розглядається Unified Modeling Language як інструмент моделювання. Автор проводить критичний аналіз мови UML і показує, що вона, маючи певну сферу свого ефективного застосування, може проявити певні слабкі сторони в деяких інших випадках. Наприкінці роботи представляється деякий альтернативний метод документування програм.

1. Життєвий цикл програми – одна з головних концепцій програмної інженерії

Нова ідея про технічну або програмну систему спричиняє нову "*постановку задачі*", яка складається в основному з двох компонент:

1.1. Аналіз вимог або потреб користувача у новій програмній системі (requirements analysis).
1.2. Формулювання або специфікація вимог (requirements specification).

Кожна з цих компонент має наступні складові:

1.1. Аналіз вимог (requirements analysis):
- виявлення вимог (requirements elicitation);
- моделювання вимог (requirements modeling);
- перевірка вимог (requirements verification).

1.2. Формулювання вимог до нової програмної системи (requirements specification):

– формування мети розробки (goal formulation);
– виявлення обмежень, що мають бути прийняті до уваги (constrains clarification);
– власне, опис вимог (requirements description).

Таким чином, на самому початку життєвого циклу програмної системи розробники мають провести роботу по виявленню вимог до системи (*requirements elicitation* або *requirements gathering*). Для цього інженери проводять вивчення *функціональних* та *нефункціональних* вимог до програмної системи, які іноді називають також якісними характеристиками системи (*qualities* або *quality requirements*) – рис.1.

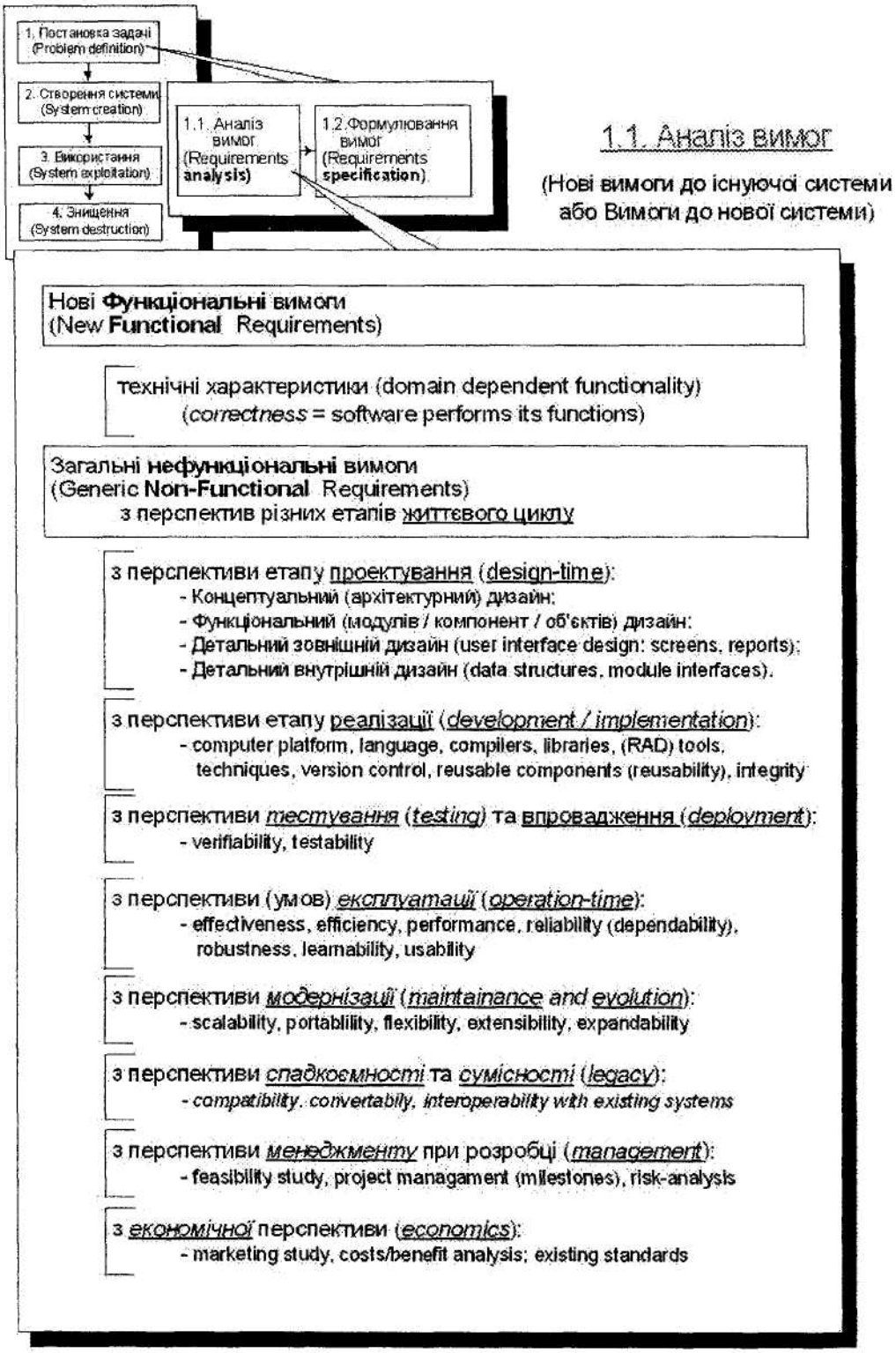


Рис. 1

В даній роботі ми надалі зосередимо увагу на методах, які застосовуються чи розробляють в підтримку процесу створення software на всіх етапах ЖЦ, починаючи зі специфікації вимог і

закінчуючи документуванням програми. Одним із таких загальних методів і є *моделювання*, а одним із нових, сучасних інструменів моделювання програм є Unified Modeling Language, UML – універсальна мова моделювання.

2. Моделювання як один з основних методів, що застосовується на всіх етапах життєвого циклу

Нагадаємо, що в самому загальному випадку *модель* деякої системи чи явища є специфічний об'єкт, що створюється у формі певного опису (або матеріального витвору), який відображає суттєві властивості системи з метою її дослідження [1]. Отже, модель будується для того, щоб краще розуміти систему, що розробляється. Не винятком у цьому є програмні системи. Але у випадку дослідження програмних систем їх моделлю може бути навіть лише певне візуальне зображення тексту програмної системи або алгоритму її роботи. Тут часто немає потреби у використанні складних математичних співвідношень як апарату моделювання; тут важливо мати лише зручний та простий засіб, що допомагає краще відобразити початкову задачу – проектування алгоритму чи програми, кодування, тестування тощо.

Саме таким засобом моделювання програм і є UML – універсальна мова моделювання [7, 8]. Хоч розвиток UML сам по собі і був певною мірою об'єднанням трьох своїх попередників, він як самостійний метод став відомим лише в 1994–95 роках, тобто відносно недавно. Основною метою застосування UML є візуалізація тих чи інших аспектів програмної системи на різних етапах її життєвого циклу.

3. Unified Modeling Language як засіб візуалізації software

Оскільки, в загальному випадку, модель відображає лише суттєві характеристики системи, то для відображення різного набору суттєвих характеристик звичайно використовується різний набір моделей. Це ж саме є справедливим і у випадку UML як інструмента моделювання. За допомогою UML є можливість відображати декілька різних аспектів однієї і тієї ж програмної системи і, більш того, виконувати це на різних етапах її життєвого циклу. На практиці це означає, що UML надає можливість для різного типу візуалізації програми, тобто в UML використовується декілька різних типів діаграм (що є, власне, моделями) для представлення програми з різних точок зору; в UML вони називаються діаграмами класів, об'єктів, взаємодій тощо.

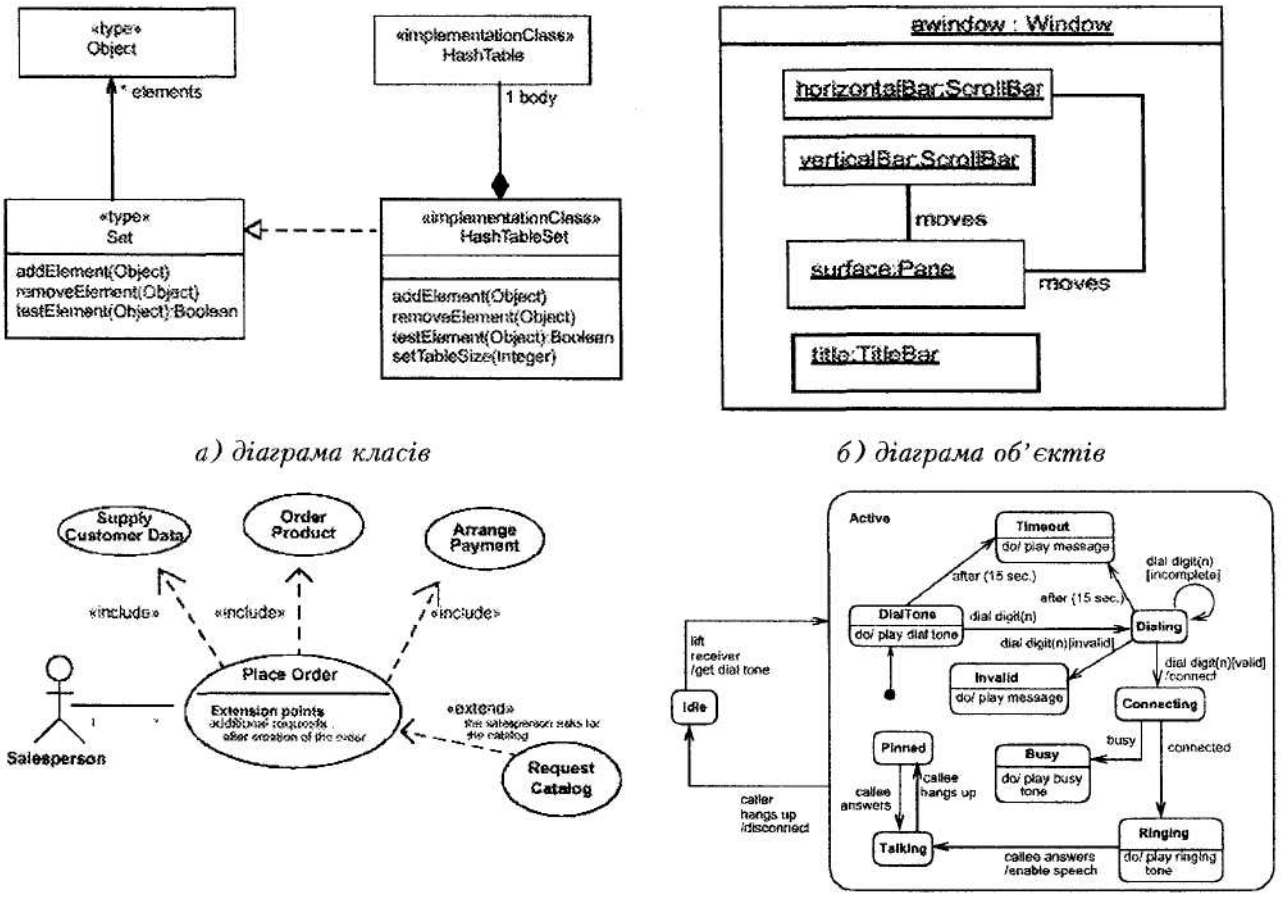
Отже, універсальна мова моделюання UML надає у використання наступні типи діаграм [7, 8]:

- діаграма прикладів використання (1. Use case diagram);
- діаграма класів (2. Class diagram);
- діаграма об'єктів (3. Object diagram);
- діаграми поведінки (behavior diagrams):
  - ✧ діаграма станів (4. Statechart diagram);
  - ✧ діаграма активностей (5. Activity diagram);
  - ✧ діаграми взаємодій (interaction diagrams):
    - ▪ діаграма послідовності виконання (6. Sequence diagram);
    - ▪ діаграма взаємодії (7. Collaboration diagram);
- діаграми реалізації програми (Implementation diagrams):
  - ✧ діаграма компонентів (8. Component diagram);
  - ✧ діаграма поставки програми до користувача або впровадження програми (9. Deployment diagram).

На рис. 2 наводяться деякі приклади візуального представлення програм з використанням діаграм різних типів. В [9] наводиться також ряд інших прикладів їх використання.

Однією з особливостей (яка є, власне, новизною) UML є те, що цей засіб орієнтований на моделювання об'єктно-орієнтованих (ОО) програм. В той же час, потрібно чітко пам'ятати, що об'єктно-орієнтований опис задачі, тобто опис, аналіз, а потім і дизайн (елементів) задачі на основі ОО підходу (ООП), тобто з застосуванням "об'єктів" та "класів" об'єктів, є "техніка" так би мовити "прив'язана" до реалізації, тобто – до мови програмування, а не до, власне, предметної задачі, що розв'язується. Якби не було проблем у компілятора при розв'язку, наприклад, задач, пов'язаних з "видимістю", тобто областю доступу (scope) змінних, їх часом життя та таке інше, тоді не було б також і ніякої мови про те, що "в прикладній задачі при її аналізі ми виділяємо об'єкти" – так званий об'єктно-орієнтований аналіз (ООА). Як бачимо, ніяких "об'єктів" початково в прикладній задачі немає; все "упирається" лише в спосіб реалізації (тобто програмування) цієї задачі. Та ж сама задача

часто може бути чудово розв'язана з використанням звичайних "структур даних" та "алгоритмів", як це і практикувалося при імперативному програмуванні. (Куди тоді ділися об'єкти, якщо вони "існували" в задачі при її об'єктно-орієнтованому аналізі?). Тобто (ще раз), об'єкти – це зручна модель, використання якої спричинено лише способом її подальшої реалізації – ОО програмуванням, але аж ніяк не самою прикладною задачею, її, так би мовити, "об'єктною природою", як це подекуди подається в літературі. (Для більш детального знайомства з деякими особливостями ООП з використанням C++, а також про можливі пастки, що виникають при цьому, можна ознайомитися в роботі автора [10]).



а) діаграма класів *б) діаграма об'єктів*

в) діаграма прикладів використання *с) діаграма станів*

Рис. 2. Приклади діаграм UML

Таким чином, одна й та ж прикладна задача може бути вирішена (тобто спочатку проаналізована, а потім і запрограмована) як з використанням об'єктів (ООА та ООП), так і без них – звичайними засобами процедурного програмування. Тобто "об'єкт" не є чимось "внутрішньо притаманним" прикладній задачі, що вирішується. Це лише штучний прийом (артефакт), створений поточним рівнем розвитку технологій – мов програмування, їх компіляторів тощо. Це – не "ідол", це – лише технологія (технологія, яка, проте, є сучасною і досить потужною). Відповідно до цього, інструментальні засоби, такі як UML, що "покликані" полегшити роботу з об'єктами, є не "панацеєю", як це іноді подається в літературі, а лише ще однією технологією, яка надає зручні засоби в одних випадках та проявляє промахи (fault) – в інших.

4. Критика UML

Отже, маючи певну сферу свого ефективного застосування, UML може проявити певні слабкі сторони в інших випадках її застосування. Нижче наведемо приклади таких "слабких" рис UML.

З точки зору проблеми моделювання програми, весь набір діаграм UML виглядає дещо неоднорідним. Так, поруч з *діаграмами* для відображення *структури даних* (object diagram), розробник програми може розглядати *діаграми впровадження програми* (deployment diagram). З одного боку, це – і непогано, тому що дозволяє вирішувати задачі моделювання для різних етапів

життєвого циклу програми. З іншого боку, важко віднайти спільні риси в таких діаграмах: вони виглядають, на думку автора, як набір різнорідних інструментів, "стулених до однієї купи" для вирішення однієї задачі – моделювання програми на всіх етапах її життєвого циклу. Така задача хоч і складається з різних за природою підзадач, все ж вони є елементами однієї проблеми, і тому було б бажано мати більш "гомогенний" набір інструментів.

Інший приклад – діаграми для моделювання структури даних: *діаграми класів* (class diagram) та *об'єктів* (object diagram). На думку автора, такий поділ є досить "надуманим", оскільки "класи" та "об'єкти класів" є категоріями, доповнюючими один одного. Класи, як аналог типу даних, недоцільно розглядати самі по собі в програмі, вони декларуються з єдиною метою – створення об'єктів таких класів (тобто "змінних таких типів") [10]. Те ж саме є справедливим по відношенню до об'єктів – опис просто структури об'єктів без будь-якої інформації про класи (на основі яких вони створюються) виглядає дещо, так би мовити, "відірваним від життя".

Звичайно, в деяких випадках на практиці є корисним мати опис "ієрархії класів". Ним користуються, зокрема, при описі бібліотек класів, таких, наприклад, як Microsoft Foundation Class (MFC) Library. В той же час, такою діаграмою ієрархії класів потрібно користуватися дуже обережно та уважно, оскільки вона показує лише "відношення спадковості" класів і зовсім не надає ніякої інформації про "вкладені класи" (nested classes) та про члени класів, які є об'єктами інших класів (тобто інформації, яка потрібна була б при аналізі агрегатних об'єктів – composite objects).

Далі, в UML виділяються дві окремих групи діаграм: статичні діаграми (*діаграми структури*) та *діаграми поведінки* об'єктів. З одного боку, це також непогано, тому що дозволяє детально зосередитися на певних рисах програми: або на її структурі даних, або на алгоритмах їх обробки. З іншого боку, це протирічить основному принципу об'єктно-орієнтованого програмування, при якому об'єкт саме і поєднує в собі ці два поняття – структури даних та методи їх обробки (поведінки об'єкта). Тому представлення об'єкта з використанням розділу діаграм на такі два типи може привести до втрати цілісності у сприйнятті об'єкта. Таке представлення було зручним при звичайному імперативному програмуванні, коли структури даних і алгоритми їх обробки розглядались як дві відносно незалежні сутності.

В той же час, об'єктивно кажучи, в UML *діаграми послідовності виконання* (sequence diagram) певною мірою вирішують задачу відображення "динаміки" виклику функцій об'єкта того чи іншого класу. Проте і тут подекуди зустрічаються випадки, коли такий опис не завжди є задовільним (див. приклад проблеми і варіант її вирішення в наступному розділі даної статті) .

Діаграми варіантів використання (user cases diagram), на думку автора, є взагалі надуманими. Як приклад – рис. 2,*в*. Фактично, ці діаграми зображують просто "перелік" певних фактів. Такий перелік можна було б подати і за допомогою звичайного тексту. Зображення ж такого тексту за допомогою "малюночка" не стільки додає щось до розуміння проблеми, що досліджується, скільки, на думку автора, більше створює враження "наукоподібності" в дослідженнях.

Діаграми діяльності є просто новою назвою давно відомого засобу зображення – *flow-chart* або *data flow*, де блоки діаграми відповідають не операторам мови програмування (лінійна послідовність, розгалуження, цикл), а цілим групам певних дій ("діяльності"), причому дій, які виконує не програма на етапі її роботи, а дій, що виконуються розробником цієї програми на етапі її створення. Як бачимо, нічого принципово нового тут також немає, це просто нова область застосування "старого", відомого апарату – flow-chart.

Діаграми станів також не є принципово новим поняттям, оскільки вони використовувались для графічного зображення скінченного автомата ще з кінця 50-х років 20-го сторіччя. В теорії скінченних автоматів вони називаються "*графом скінченного автомата*", або "*графом переходів*", або "*графом станів*" автомата. (Дивись також в [1], в розділі "Теорія скінченних автоматів", взаємозв'язок між такими двома "різними" способами представлення, як "граф скінченного автомата" та "flow chart".). Врешті-решт, в теорії UML її автори навіть майже не змінили назви: а саме, тут використовується термін "state chart". Але тоді все ж постає питання: "що ж тут нового, крім сфери застосування?".

Діаграма компонентів, якщо мова йде про компоненти ActiveX / OLE / COM [1], є безперечно корисним і певною мірою новим засобом візуалізації, хоча б тому, що самі ActiveX / OLE / COM-об'єкти є відносно новим явищем, вони "введені в обіг" в другій половині 90-х років. В той же час, для досить великої кількості прикладних програм такі компоненти все ще не застосовуються досить активно, а в ряді випадків їх застосування в ОО програмі навіть і недоцільно. Тому в таких випадках використання діаграм компонентів є взагалі непотрібним і навіть неможливим.

З іншого боку, простий аналіз проблеми повного документування software, а саме документування архітектури software, чи таких складових software, як інтерфейс користувача, а також написання різних видів документацій за software на різних етапах життєвого циклу, також показує, что UML не охоплює в достатній мірі проблему документування software. При документуванні сучасних ОО програм доцільно було б також взяти до уваги існуючий історичний досвід, а саме існуючі ГОСТи, такі як Єдина Система Програмної Документації (ЄСПД) [11]. (Приклад такого "повного" документування програми можна знайти в [12]). В UML лише охоплюється частина цієї проблеми, роблячи акцент на тому, що проектування та реалізація (програмування) будуть відбуватися на основі об'єктно-орієнтованого підходу, а не на основі імперативного програмування.

Отже, із 9-ти типів діаграм лише деякі з них виявляються майже завжди корисними. Але навіть і тоді існує така сфера застосування, де навіть для цих декількох типів діаграм важко їх ефективно використосувати. Такою сферою є об'єктно-орієнтоване програмування з використанням системної бібліотеки класів, тобто розробка програм, наприклад, для MICROSOFT® WINDOWS® з використанням такої бібліотеки класів, як MFC. Така сфера застосування ООП є сама по собі досить великою та поширеною і викликає інтерес у розробників, проте використання UML тут є в ряді випадків досить проблематичним (див. нижче приклад).

5. Проблема та варіант її вирішення

Розглянемо два приклади досить простих програм для MICROSOFT® WINDOWS®, які згенеровані (автоматично) за допомогою MS Visual Studio Application Wizard. Користувацький інтерфейс програми в обох випадках майже однаковий – рис. 3, проте згенеровані тексти програм досить різні. У першому випадку створюється програма, що має так званий Single Document Interface (SDI) без підтримки Document / View архітектури, у другому випадку – SDI програма з підтримкою Document / View – рис. 5. (Текст програми для 1-го варіанту тут не наводиться у зв'язку з обмеженим розміром статті, проте читач може легко отримати такий текст за допомогою Application Wizard). Звичайно, для реалізації таких рис в програмі використовується системна бібліотека класів MFC.

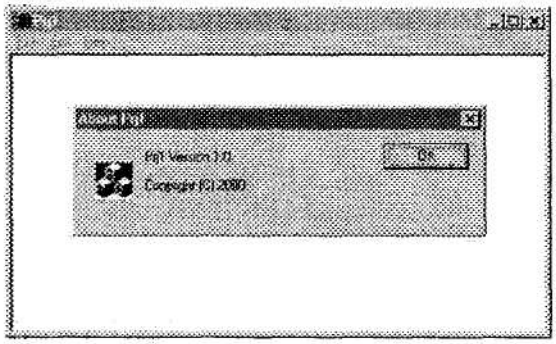


Рис. 3. Користувацький інтерфейс програм

```
        StartUp Code:
0.  WinMainCRTStartup()                                              KERNEL32
    _tWinMain()                                          (MFC-APPMODUL.CPP)
    AfxWinMain()                                                (MFC-WINMAIN.CPP)
        CWinThread* pThread                              (MFC-WINMAIN.CPP)
        CWinApp* pApp                                    (MFC-WINMAIN.CPP)

1.  CPrjlApp theApp;                    (Prjl.cpp)
2.  CPrjlApp::CPrjlApp();               (Prjl.cpp)

        AfxWinInit()                                     (MFC-APPINIT.CPP)
        CWinApp::InitApplication()                       (MFC-APPCORE.CPP)
        pThread->InitInstance()                                 (MFC-WINMAIN.CPP)

3.  CPrjlApp::InitInstance();           (Prjl.cpp)
4.  SetRegistryKey(…);                  (Prjl.cpp)
5.  CMainFrame::CMainFrame();                   (MainFrm.cpp)
6.    CFrameWnd::CFrameWnd();                   (MFC-WinFrm.cpp)
7.      CWnd::CWnd();                           (MFC-WinCore.cpp)
8.  CChildView    m_wndView;                    (MainFrm.h)
9.  CMainFrame::OnCreate(…);                    (MainFrm.cpp)
10. CMainFrame::PreCreateWindow(…);             (MainFrm.cpp)
11. CWnd::PreCreateWindow(…);                   (MFC)
12. AfxRegisterWndClass(…)                              (WinAPI)
13. CMainFrame.m_wndView::CWnd::Create(…);      (MFC)
```

```
14. CChildView::CChildView();                                   (ChildView.cpp)
15. CChildView::PreCreateWindow(…);                             (ChildView.cpp)
16. CWnd::PreCreateWindow(…);                                           (MFC)
17. AfxRegisterWndClass(…);                                                     (WinAPI)
18. CFrameWnd::LoadFrame(…);                                            (MFC)
19. CWnd::ShowWindow(…);                                                        (MFC)
20. CWnd::UpdateWindow();                                               (MFC)
21. CMainFrame::OnSetFocus(…);                          (MainFrm.cpp)
22. CMainFrame.m_wndView::CWnd::SetFocus();                     (MFC)
23. CMainFrame::OnCmdMsg(…);                            (MainFrm.cpp)
24. CMainFrame.m_wndView::OnCmdMsg();           (MainFrm.cpp)
25. CMainFrame.m_wndView::CCmdTarget::OnCmdMsg();               (MFC)
26. CFrameWnd::OnCmdMsg(…);                             (MainFrm.cpp)
27. CMainFrame::CCmdTarget::OnCmdMsg();                                 (MFC)
28. BEGIN_MESSAGE_MAP(…)                                (Prj1.cpp)
29. ON_COMMAND(OnAppAbout);                     (Prj1.cpp)
30. CAboutDlg::CAboutDlg();                     (Prj1.cpp)
31. CAboutDlg::DoDataExchange(…);               (Prj1.cpp)
32. CDialog::DoDataExchange(…);                                         (MFC)
33. END_MESSAGE_MAP();                          (Prj1.cpp)
```

Рис. 4. Опис послідовності викликів функцій-методів класів для 1-го прикладу

```
//--------------------------------------------------------------------------------------------
BOOL CPr21App::InitInstance()
{
        SetRegistryKey(_T("Local AppWizard-Generated Applications"));

        LoadStdProfileSettings(0);  // Load standard INI file options (including MRU)

        // Register the application's document templates.  Document templates
        //  serve as the connection between documents, frame windows and views.

        CSingleDocTemplate* pDocTemplate;
        pDocTemplate = new CSingleDocTemplate(
                IDR_MAINFRAME,
                RUNTIME_CLASS(CPrj2Doc),
                RUNTIME_CLASS(CMainFrame),       // main SDI frame window
                RUNTIME_CLASS(CPrj2View));
        AddDocTemplate(pDocTemplate);

        // Parse command line for standard shell commands, DDE, file open
        CCommandLineInfo cmdInfo;
        ParseCommandLine(cmdInfo);

        // Dispatch commands specified on the command line
        if (!ProcessShellCommand(cmdInfo))
                return FALSE;

        // The one and only window has been initialized, so show and update it.
        m_pMainWnd->ShowWindow(SW_SHOW);
        m_pMainWnd->UpdateWindow();

        return TRUE;
}
//--------------------------------------------------------------------------------------------
//--------------------------------------------------------------------------------------------
class CPrj2View : public CView
//    ~~~~~~~~~~
{
public:
        CPrj2Doc * GetDocument();

        CPrj2View();
virtual         ~CPrj2View();
virtual void OnDraw(CDC* pDC);  // overridden to draw this view
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
};
//--------------------------------------------------------------------------------------------
//--------------------------------------------------------------------------------------------
BOOL CPrj2View::PreCreateWindow(CREATESTRUCT& cs)
{
        // TODO: Modify the Window class or styles here by modifying the CREATESTRUCT cs

        return CView::PreCreateWindow(cs);
}
```

```
//--------------------------------------------------------------------------------------------
// CPrj2View drawing

void CPrj2View::OnDraw(CDC* pDC)
{
        CPrj2Doc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        // TODO: add draw code for native data here
}
//--------------------------------------------------------------------------------------------
CPrj2Doc* CPrj2View::GetDocument() // non-debug version is inline
{
        ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CPrj2Doc)));
        return (CPrj2Doc*)m_pDocument;
}
//--------------------------------------------------------------------------------------------
//--------------------------------------------------------------------------------------------

4. CPrj2Doc.* h & cpp files

//--------------------------------------------------------------------------------------------
class CPrj2Doc : public CDocument
//    ~~~~~~~~~~
{
public:

        CPrj2Doc();
virtual         ~CPrj2Doc();
virtual BOOL OnNewDocument();
virtual void Serialize(CArchive& ar);
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
};
//--------------------------------------------------------------------------------------------
//--------------------------------------------------------------------------------------------
BOOL CPrj2Doc::OnNewDocument()
{
        if (!CDocument::OnNewDocument())
                return FALSE;

        // TODO: add reinitialization code here
        // (SDI documents will reuse this document)

        return TRUE;
}
//--------------------------------------------------------------------------------------------
```

Рис. 5. Фрагмент тексту згенерованої SDI програми з підтримкою Document / View архітектури

І хоча текст програм є досить коротеньким і виглядає на перший погляд дуже простим, досконально розібратися в роботі таких програм виявляється не дуже просто. Це відбувається саме тому, що, фактично, при роботі програм задіяна велика кількість класів, об'єктів та функцій системної бібліотеки MFC. І для того, щоб прослідкувати їх роботу і розібратися, в чому ж полягає різниця в таких двох різних реалізаціях програм, тут би і стали в пригоді певні моделі чи діаграми.

В даній роботі пропонується для вирішення вказаної проблеми використати наступні види діаграм – рис. 4, 6–7. Тут, на діаграмах рис. 6–7, прийнято такі графічні позначення: прямокутник з округленими кутами позначає "об'єкт" певного класового типу, прямокутник з гострими кутами позначає "базовий клас" (в даних прикладах – системний базовий клас, тобто клас, реалізований в системній бібліотеці MFC), лінії зі стрілками вказують на "виклик функції" даного класу, або базового класу, у випадку "поліморфізму", тобто "перевантажених віртуальних функцій" тощо. Читач, за бажанням, може дещо змінити вид графічних позначень, наведених на діаграмах рис. 6–7, але головне — має бути збережена ідея: на таких діаграмах доцільно, на думку автора, одночасно відображати і "об'єкти", і "базові класи", і те, які функції (звичайні функції-члени, конструктори, віртуальні функції, функції системної необ'єктної бібліотеки WinAPI тощо) і яких класів викликаються і які нові об'єкти при цьому вони створюють. Діаграма на рис. 4 показує "динаміку" виклику функцій і є, за своєю природою, дещо подібною до *діаграми послідовності виконання* (sequence diagram) мови UML. На цій діаграмі також, за бажанням, можуть бути використані певні графічні елементи.

Автор гадає, що використання запропонованих діаграм (моделей) дозволяє більш виразно (і значить – ефективно) відобразити природу таких задач, і це буде сприяти кращому їх розумінню.

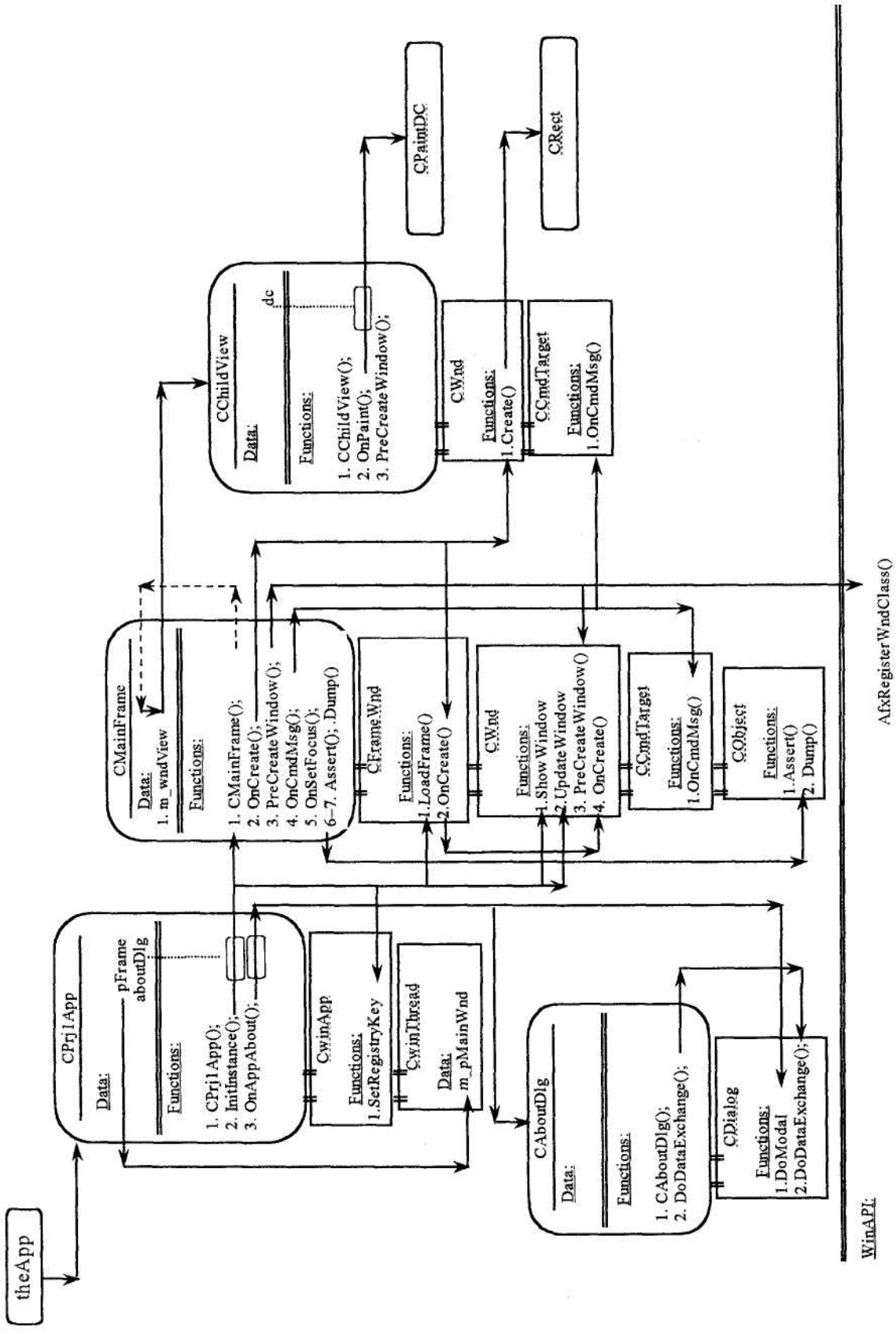


Рис. 6. Варіант 1-й, без підтримки Doc/View. Діаграма об'єктів класового типу з відображенням ієрархії класів та викликів функцій

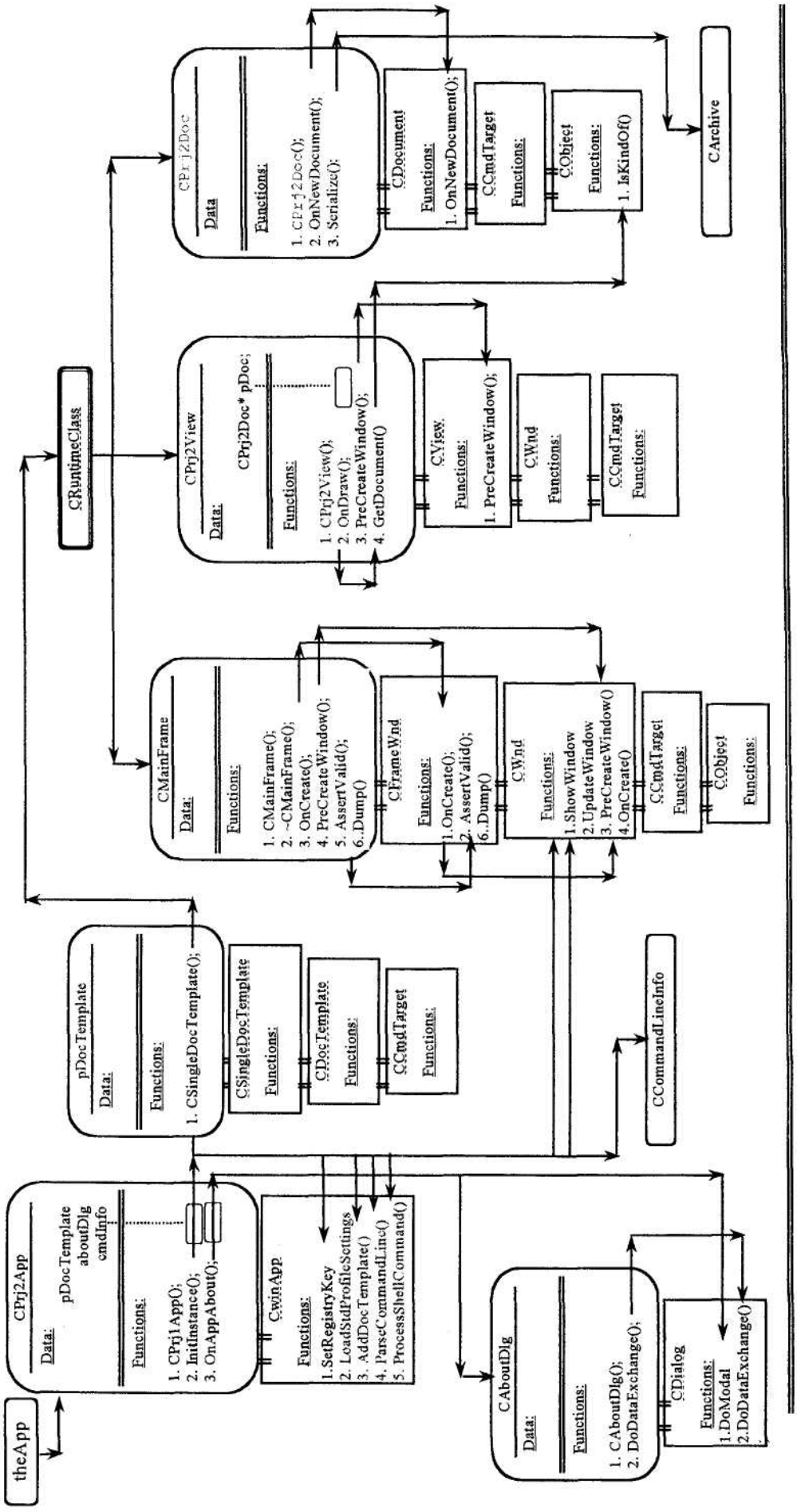


Рис. 7. Варіант 2-й, з підтримкою. Doc/View. Діаграма об'єктів класового типу з відображенням ієрархії класів та викликів функцій

Висновки

Отже, UML – не панацея (як про це подекуди з захопленням повідомляється в літературі)! Це просто наступна (в довгому історичному переліку подібних) спроба графічного зображення програм, спроба, що подекуди є не найбільш вдалою. В той же час UML мають свою область ефективного їх застосування, наприклад, етап дизайну ОО програм.

Популярність UML, напевно, пояснюється тим, що "в потрібний час" UML зайняла нішу візуалізації ООП. Проте на сьогодні UML ще не вирішує всіх проблем в програмній інженерії; тут залишаються ще невирішеними задачі повного документування програм (з використанням певних ГОСТ, ЕСПД), документування програм для MICROSOFT® WINDOWS® з використанням системної бібліотеки класів та можливо ще ряд інших.

ЛІТЕРАТУРА:

1. *Колодницький М.М.* Елементи теорії САПР складних систем. – Житомир: ЖІТІ, 1999. – 512 с.
2. Software engineering. Report on a conference sponsored by the NATO Science Committee. Garmisch,Germany,7th to 11th October 1968. – 136 p.
3. *Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli.* Fundamentals of software engineering. Englewood Cliffs, N.J.: Prentice-Hall International, 1991. – 573 p.
4. *Ian Sommerville.* Software Engineering. Addison-Wesley, 2001. – 686 p.
5. *Roger Pressman.* Software engineering. A Practitioner's Approach, McGraw-Hill, 2000.
6. Encyclopedia of software engineering / Edited by John J.Marciniak. Chichester: Wiley, 1994. Vol. 1–2.
7. *Буч Г., Рамбо Дж., Джекобсон А.* Язык UML. Руководство пользователя. – М.: ДМК, 2000. – 432 с.
8. http://www.rational.com/uml/resources/documentation/index.jsp
9. *Янчук В.М.* та інші // Методи та засоби математичного моделювання міграції радіонуклідів у природних екосистемах. Том 2. Міждисциплінарний аналіз проблеми / В.М. Янчук, М.М. Колодницький, А.М. Ковальчук, В.Г. Левицький, О.О. Орлов. – Житомир: ЖІТІ, 2002. – 224 с.
10. *Колодницький М.М.* Аналіз парадигми ООП мовою С++. ("Що таке класи?") // Вісник ЖІТІ / Технічні науки, 2002. – № 2 (21). – С. 120.
11. *Колодницький М.М., Данильченко О.М., Колодницька Р.В.* Методичний посібник по виконанню та оформленню дипломних та курсових робіт. – Житомир: ЖІТІ, 1996. – 78 с.
12. *Колодницький М.М.* Інтегрований малті-медійний пакет програм для автоматизації обліку комерційної діяльності малого підприємства – "Інтеграл" // Вісник ЖІТІ / Технічні науки, 1997. – № 5. – С. 130–142.

КОЛОДНИЦЬКИЙ Микола Михайлович – кандидат технічних наук, професор кафедри комп'ютерних інтегрованих систем Житомирського інженерно-технологічного інституту.

Наукові інтереси:

- математичне моделювання систем;
- комп'ютерні інформаційні технології.

Подано 13.06.2002