УДК 004.415.2.925.8

ФІСУН М.Т., БИКОВ Д.П.

МОДЕЛЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ОСНОВІ ІНТЕГРАЦІЇ МОДУЛЬНОЇ, ПОШАРОВОЇ ТА ОБ'ЄКТНО-ОРІЄНТОВАНОЇ АРХІТЕКТУР

Загальна постановка проблеми та її зв'язок з науково-практичними задачами. Потреби у змінені (реінжинірінгу) існуючого програмного забезпечення може виникнути в ході вирішення широкого кола задач з його модернізації. В загальному випадку зміна існуючого програмного забезпечення здатні зачепити не тільки його код, але й усю решту артефактів, пов'язаних з програмною системою, що трансформується. Однією з найбільш суттєвих різновидів *є зміна архітектури програмної системи.* У якості прикладів можна привести такі чинники, що потребують змін архітектури існуючого ПЗ [1, 2]:

- *Перетворення, обумовлені функціональними змінами ПЗ*. При цьому бажано, щоб впровадження нової функціональності не зачепило існуючу логіку системи. Також бажано, щоб складність впровадження нової функціональності в існуючу систему не перевищило суттєвим чином складність реалізації цієї функціональності в рамках нового проекту. Гарна архітектура дозволяє досягнути поставлених цілей [3]. Таким чином, зміна існуючої архітектури – гарний крок на шляху впровадження нової функціональності, який, до того ж, полегшує подальшу еволюцію системи.
- *Зміна платформи ПЗ*. Вкрай бажано, щоб зміна платформи ПЗ як можна менш зачепила існуючий код, і щоб можна було обмежитися змінами тільки у вузькій платформо-залежному прошарку системи. Виділення такого прошарку – архітектурна задача. Її розв'язок завжди пов'язаний з необхідністю зміни архітектури.
- *Перетворення, що пов'язані з реорганізацією компанії-розробника ПЗ***.** Прикладом такої реорганізації може стати переведення реінжинірінгу існуючого ПЗ на аутсорсинг. Рішення про залучення аутсорсинга – типовий крок щодо оптимізації виробництва взагалі і програмних продуктів зокрема. Нажаль, цей крок часто ускладнюється проблемою виділення та передачі компонентів для зовнішньої розробки. Зміна архітектури програмної системи здатна полегшити розв'язання цієї задачі.

Наведений вище перелік сценаріїв, що приводять до потреби зміни архітектури існуючого ПЗ, на цьому не вичерпується: наведені вище приклади лише демонструють широкий спектр задач, що обумовлюють необхідність подібних змін. Наприклад, до програмного продукту можуть бути висунуті вимоги багатоплатформеності або мобільності [4].

Специфіка опису архітектури та її змін заключається в тому, що, на відміну від програмного коду, архітектура не має явного представлення, за виключенням, можливо, тих випадків, коли вона явно задокументована. Однак, навіть в останньому

оптимістичному випадку, важко гарантувати відповідність задокументованої архітектури фактичній багаторівневій логічній структурі, яка насправді існує в системі.

На даний час не існує загально прийнятого визначення терміну "архітектура програмного забезпечення". В той же час існує велика кількість різноманітних визначень цього поняття, що мають багато в чому подібний смисл. В цій роботі цей термін будемо вживати у смислі [5]: *архітектура програмного забезпечення – це первина організація системи, сформована її компонентами, відношеннями між компонентами та зовнішнім середовищем, а також принципами, що визначають дизайн та еволюцію системи.*

Способом опису архітектури та її змін мають стати моделі архітектур, формальний апарат, методи та інструментарій яких розвивався разом з методологією, методами та інструментарієм програмування взагалі.

Розвиток практичних і теоретичних методів систематичного програмування пройшов низку етапів, таких як модульний, структурний, складальний, пошаровий тощо. В останні роки в програмуванні переважало становлення й розвиток об'єктно-орієнтованого проектування (ООП) програмних систем (ПС) [6, 7] взагалі та програмного забезпечення (ПЗ) систем автоматизованого проектування (САПР) зокрема.

Удосконалення методології, методів та інструментарію ООП досягло такого рівня розвитку, що воно стало базисом генеруючого (породжувального) програмування, в якому визначені шляхи його розвитку та усунення ряду недоліків, пов'язаних з використанням готових компонент, властивостей мінливості, взаємодії, синхронізації та ін. Породжувальне програмування заклало базис майбутнього програмування, заснованого на сучасних методах програмування, нових формалізмах та об'єднувальних моделях, за допомогою яких можна буде створювати більш довговічні й якісні програмні продукти [8]. Однак питання створення інтегруючих моделей ще недостатньо пророблені. Особливо це стосується програмного забезпечення САПР, оскільки воно являється дуже складним, об'ємним та багатофункціональним, тому задача аналізу існуючих моделей та розробка нових об'єднувальних моделей програмного забезпечення взагалі та моделей архітектур ПЗ *є актуальною.*

Огляд публікацій та аналіз невирішених питань.

Еволюція архітектур прикладного програмного забезпечення (ППЗ), зокрема ППЗ систем автоматизованого проектування (САПР) пройшла декілька етапів, основними з яких є [8, 9, 10]: модульна, блочна, пошарова, об'єктна, інші.

В [11] запропоновано теоретичні основи багаторівневого структурного проектування програм на базі регулярних схем алгоритмів та алгоритмічних алгебр структур даних, але там не розглядаються питання пошарової архітектури ПЗ.

В [12] пошарова архітектура розглядається на дуже загальному рівні, а питання поєднання цієї архітектури з іншими не розглядаються.

В [13] досліджено підходи щодо системної та функціональної архітектур при проектування пакетів прикладних програм (ППП) взагалі та, зокрема, ППП для СППР (систем підтримки прийняття рішень), CASE (Computer Aided Software Engineering), інтелектуальної обробки інформації. Загальну архітектуру ППП пропонується реалізовувати з чотирьох шарів: уведення інформації, корегування інформації, виведення (показ) інформації та СКБД. Але програмне забезпечення САПР є більш складнішим і більш функціональним, чим розглянуті в наведеній роботі ППП. До того ж методологія розробки ППП в ті часи ще не використовувала методи ООП. Теж саме можна сказати й про роботу [14]. В [15] більш детально розглянуті засоби об'єктно-орієнтованого проектування і об'єктної архітектури ПЗ, однак там не розглянуті інші архітектури. В [9] запропонована п'ятирівнева пошарова архітектура ППЗ САПР, яка

включає і об'єктну архітектуру в кожному шарі, але запропонована схема не доведена до той степені формалізації, щоб її можна було назвати моделлю.

З наведеного аналізу можна зробити висновок, що питання єдиної моделі архітектури ПЗ, яка б поєднувала пошарову та об'єктну архітектури, ще недостатньо висвітлені, що і є предметом дослідження даної статті, яке можна розглядати як продовження дослідження, представленого в [9].

Результати досліджень. Оскільки пошарова архітектура вийшла із блочної (модульної) структури, то побудову моделі почнемо для останньої.

1. Формальний опис блочної моделі. Для представлення модульних структур використовують математичний апарат теорії графів. Граф G визначається трійкою G =(V,E,F), де: V – множина вершин (точок, вузлів) графа; E – множина ребер (дуг) графа; F – відношення множини E на множину V. З цього випливає, що E є кінцевою підмножиною прямого добутку VxVxZ, де Z – множина цілих чисел. З урахуванням наведеного граф можна задавати двійкою G =(V,E). Програмна структура, що складається з модулів, описується орієнтованим мультиграфом. У ньому кожна дуга відповідає оператору CALL і з'єднує вершину, що відповідає модулю, який викликає, з вершиною, що відповідає модулю, який викликається, а кількість дуг між вершинами відповідає кількості викликів. Як відомо, мультиграф можна звести до простого зваженого графа, де кожне ребро ще має таку характеристику, як кількість дуг, що з'єднують вершини. В [14] модель модульної структури програмного агрегату визначається як трійка T =(G,P,F), де G =(V,E) – орієнтований граф модульної структури; P – множина модулів, що входять до складу програмного агрегату; F – функція відображення V на P, тобто F: V → P. У загальному випадку елементу із E може відповідати декілька вершин із V. Якщо виключити випадки застосування рекурсивних функцій, то граф модульної структури можна представити орієнтованим деревом, при цьому програмний код представляється сукупністю програмних модулів P ={pi}, i=1,...N. Кожному модулю pi поставимо у відповідність підмножину pi→{pj}€P, j=1,…,Nj; Nj≤N-1. Таку структуру програми можна представити, як говорилося, у вигляді дерева аналогічно конструктивному графу промислового виробу.

На рис. 1а показано схема взаємозв'язків модулів гіпотетичного програмного продукту A (головний модуль), де кількість ліній, що виходять із модуля, відповідає кількості ділянок виклику модуля, а крайня ліва частина прямокутника відповідає ділянці, що відповідає передачі йому параметрів та повернення до модуля, що викликав даний.

Ця схема легко представляється орієнтованим деревом, яке далі по тексту будемо називати просто деревом. Для подальших викладок будемо таку схему представляти зваженим деревом, тобто кожній з'єднувальній дузі будемо ставити у відповідність кількість з'єднань. Таке дерево програмного продукту A представлено на рис. 1b, де прийняті наступні позначення: в прямокутнику вказано код програмного компоненту, а на дугах – кількість звернень до цього компоненту. У правій частині рисунка вказані рівні входження (крапки й число). Компоненти, що є складальними одиницями (вузли крім кореневого), будемо називати програмними агрегатами (ПА), а елементарні компоненти (листя дерева) – програмними модулями (ПМ).

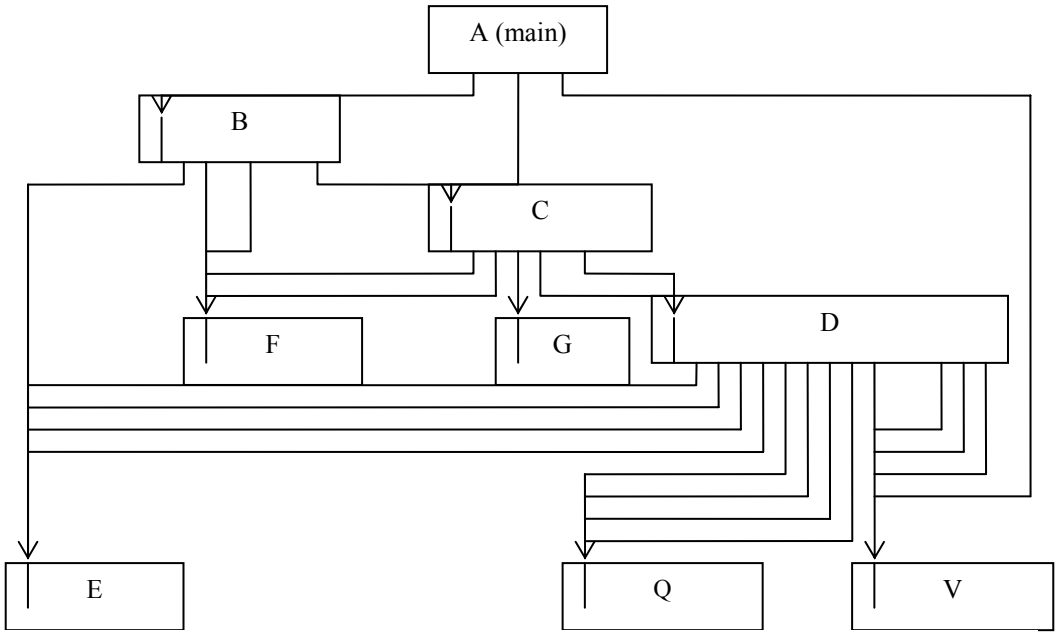

Рис. 1a. Схема викликів модулів головної програми A

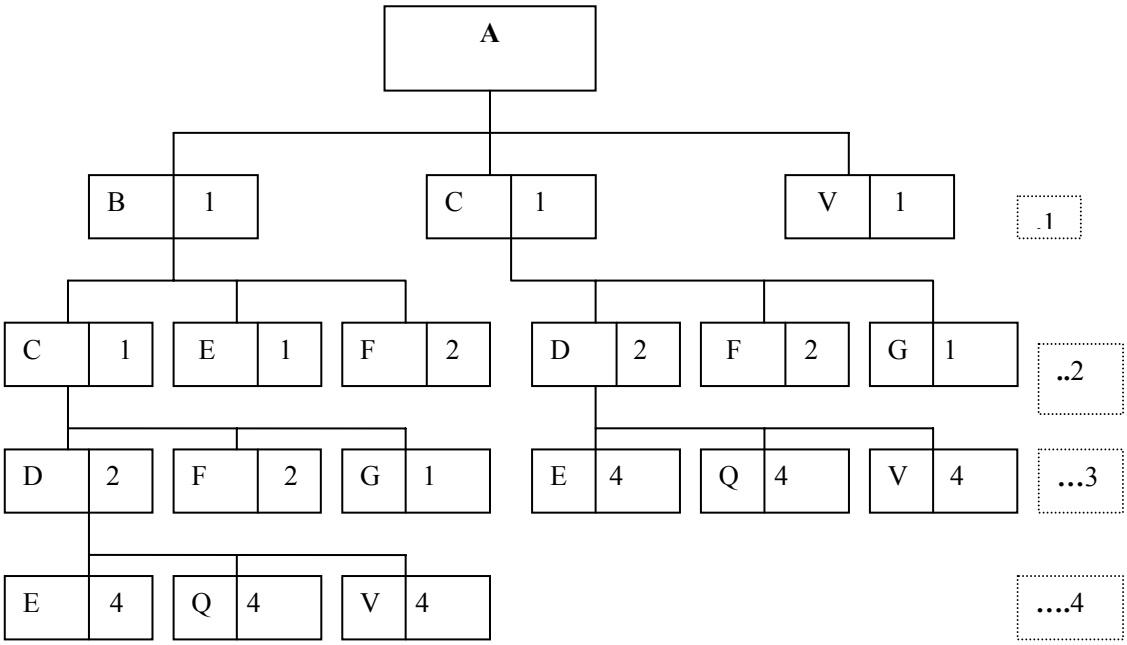

Рис. 1b. Структура програмного продукту A та рівні його компонентів

Структура дерева програми залежить від технології її компонування. Крайнім випадком є нуль-дерево, тобто така програма, що не має жодної підпрограми або функції. Як і для складного промислового виробу, для блоків і модулів ПЗ може застосувати процедури розвузлування (просте, структурне й сумарне) і входження (просте, структурне й сумарне). Результати названих процедур для програмного продукту, представленого графом на рис. 1b, наведено в таблицях на рис. 2, де:

a) – однорівневе розвузлування програмного блоку **B**;
b) – структурне розвузлування програмного продукту **A**;
c) – сумарне розвузлування програмного продукту **A**;
d) – однорівневе входження програмного модуля **V**;
e) – структурне входження програмного модуля V до програмного продукту **A**;
f) – сумарне входження програмного модуля V до програмного продукту **A**.

s	*k*	*q*
B	C	1
B	E	1
B	F	2

a)

s	*k*	*q*	*l*
A	B	1	1
A	C	1	1
A	V	1	1
B	C	1	2
B	E	1	2
B	F	2	2
C	D	2	3
C	F	2	3
C	G	1	3
D	E	4	4
D	Q	4	4
D	V	4	4
C	D	2	2
C	F	2	2
C	G	1	2
D	E	4	3
D	Q	4	3
D	V	4	3

b)

k	*s*	*q*	*maxl*
V	D	4	4
V	D	3	4
V	A	1	1

d)

s	*k*	*q*	*maxl*
A	B	1	1
A	C	2	2
A	D	4	3
A	E	17	4
A	F	6	3
A	G	2	3
A	Q	16	4
A	V	17	4

c)

k	*s*	*q*	*maxl*
V	D	4	4
D	C	3	2
C	B	2	1
B	A	1	1
V	D	3	4
D	C	2	2
C	A	1	1
V	A	1	1

e)

k	*s*	*q*	*maxl*
V	A	17	4

f)

Рис. 2. Результати процедур розвузлування та входження програмних агрегатів та модулів програмного продукту на рис. 1b

На рис. 2 прийнято такі позначення атрибутів таблиць:

s – код ПП або ПА як складальної одиниці;

k – код ПМ або ПА як компонента складальної одиниці;

q – кількість компонентів;

l – рівень входження компонента;

maxl – максимальний рівень входження компонента.

Для визначення певних операцій над модульними структурами часто використовується матричне представлення їх графів взагалі та матриці викликів зокрема. На рис. 3 представлена матриця викликів для програмного продукту **A**, що на рис. 1b.

S =

	A	B	C	D	E	F	G	Q	V	**R** =
A		1	1						1	3
B			1		1	2				4
C				2		2	1			5
D					4			4	4	12
E										0
F										0

G										0
Q										0
V										0
P										
=	0	1	2	2	5	4	1	4	5	

Рис. 3. Матриця викликів для програмного продукту А

Елемент матриці **S**ij визначає кількість звернень (операторів CALL або функцій) із модуля `i` до модуля `j`. Праворуч та знизу матриці наведено характеристичні вектори `P` та `R`, де `pj` показує, скільки разів у програмному продукті зустрічається виклик модуля `j`, а `ri` – скільки інших модулів викликає модуль `i`.

Після того, як будуть виконані розрахунки з розвузлування та входження і сформовані відповідні вихідні таблиці й матриці цих задач, можна проводити аналіз структури програми. В даній роботі аналіз різних архітектур програмного забезпечення проводиться з позицій оцінки мобільності ПЗ, оскільки мобільність є майже головною вимогою до, наприклад, програмного забезпечення САПР [7]. Це обумовлено його складністю та великими обсягами програмного коду, існуванням і появою нових різноманітних форматів уводу/виводу інформації, інтеграції САПР з багатьма підсистемами як в самій САПР, так і з іншими системами автоматизації. Проблема мобільності формулюється як забезпечення можливості виконання одної й той ж програми у різних апаратно-операційних середовищах без суттєвої переробки програми, тобто мобільність програмних засобів – це підготовленість до перенесення із одного апаратно-операційного середовища в інше [4]. Відомо, що біля 70% вартості програмних продуктів припадає на супровід, який включає виправлення помилок, модифікацію структур і форматів даних, уведення додаткової функціональності, перенесення на іншу програмну або програмно-апаратну платформу. Встановлення вимог до мобільності програмних засобів може бути зведено до визначення трудомісткості та тривалості процесів адаптації до нових характеристик користувачів та зовнішнього середовища, інсталяції версій базового програмного забезпечення та/або заміни крупних компонентів версій ПЗ на вимогу замовників або конкретних користувачів. Структуру понять, що визначають мобільність ПЗ, схематично показано на рис. 4.

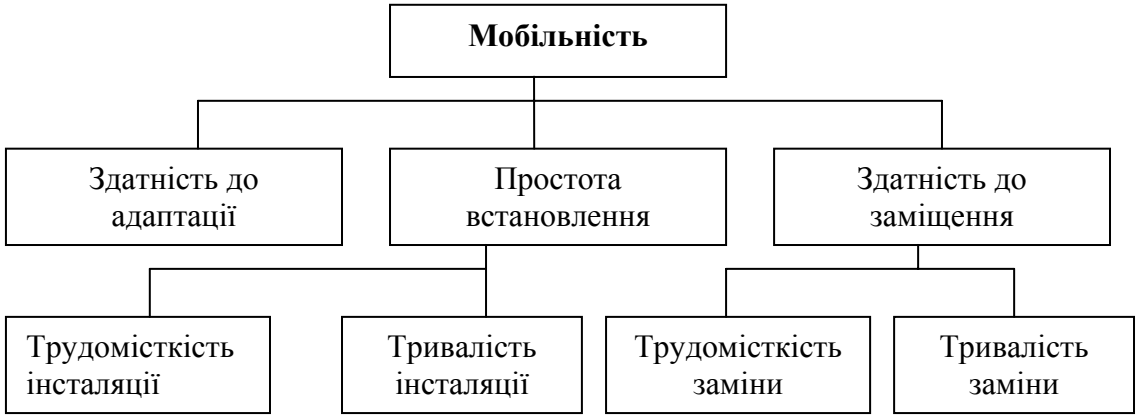

Рис. 4. Складові характеристики "Мобільність"

Здатність до адаптації означає можливість враховувати широкий спектр вимог щодо функціональності ПЗ і автоматично налагоджуватися під них [16]. Вона може

досягатися різними шляхами, але ця складова мобільності ПЗ тут не розглядається. Не будемо розглядати також і другу компоненту – "простота встановлення", оскільки вона суттєво залежить від платформи, на базі якої розробляється ПЗ, та від інструментальних засобів розробки. Тому зупинимося на третій складовій – "здатність до заміщення", а для спрощення розміркувань – на компоненті останньої – "трудомісткість заміни", яку позначимо B. Наведені нижче міркування щодо трудомісткості заміни можна без особливих проблем перенести і на складову "тривалість заміни".

Хай треба перепрограмувати модуль Mk, тоді

B=B1k+B2k+B3k,

де: B1k - трудомісткість заміни (перепрограмування та тестування) самого модуля;

B2k - трудомісткість перепрограмування ділянок коду виклику модулем Mk модулів першого рівня розвузлування та комплексне тестування модулів усіх структурних дерев модулів Mk, тобто $B2k = \sum Ti * Pi$;

B3k - трудомісткість перепрограмування ділянок коду виклику модуля Mk із модулів, куди безпосередньо входить модуль Mk, та тестування модулів усього структурного дерева від кореневої вершини до модулів Mk, тобто $B3k = \sum Tj * Rj$;

Ti, Tj – трудомісткість заміни модулів уверх та вниз по структурному дереву;

Pi –кількість викликів модулем Mk модулів Mi;

Rj – кількість викликів модуля Mk модулями Mj.

Значення Pi та Rj можна отримати із матриць зв'язності для модулів Mk. Так, для модуля **C**, що входить до складу програмного продукту **A** (рис. 1b) із матриці викликів (рис. 3) кортежі значень Pi та Rj мають значення <0,0,0,2,0,2,1,0,0> та <0,0,2,0,0,0,0,0,0> відповідно.

Із наведеного приходимо до висновку, що при великій кількості модулів і багаторівневій структурі графа програмного продукту задача заміни модулів має велику розмірність, яка обумовлює значну трудомісткість та, відповідно, низьку мобільність. Така монолітна структура не забезпечує ізоляції даних – в різних ділянках коду використовується інформація про структуру усієї системи. Розширення такого програмного продукту є дуже складною задачею, тому що зміна деякої процедури або функції може привести до помилок в інших частинах системи, які, на перший погляд, не мають до неї ніякого відношення. Крім того, налагодження та пошук помилок в такій системі завжди має більш довгий термін часу. Перенесення ж такого програмного продукту на іншу платформу взагалі стає дуже складною задачею.

2. Формальний опис об'єктної моделі. Спочатку наведемо деякі відомі терміни й визначення щодо методології об'єктно-орієнтованого програмування (ООП). *Об'єкт* – це структура даних, фізичний формат якої прихований у визначенні типу. Сукупність об'єктів створює клас об'єктів. Об'єкти мають набір властивостей, названих *атрибутами*, з якими оперує группа *сервисів* або *методів*. Завдяки властивості успадкування методів можна створювати нові класи на основі існуючих, додаваючи нові атрибути та нові методи та використовуючи атрибути й методи вищих за ієрархією класів об'єктів. Для спрощення розгляду питань мобільності ПЗ будемо розглядати тільки ієрархічні структури класів, оскільки вони найбільш поширені, а такі структури також можна представити деревами. На рис. 5 наведено приклади діаграм класів для двох структур.

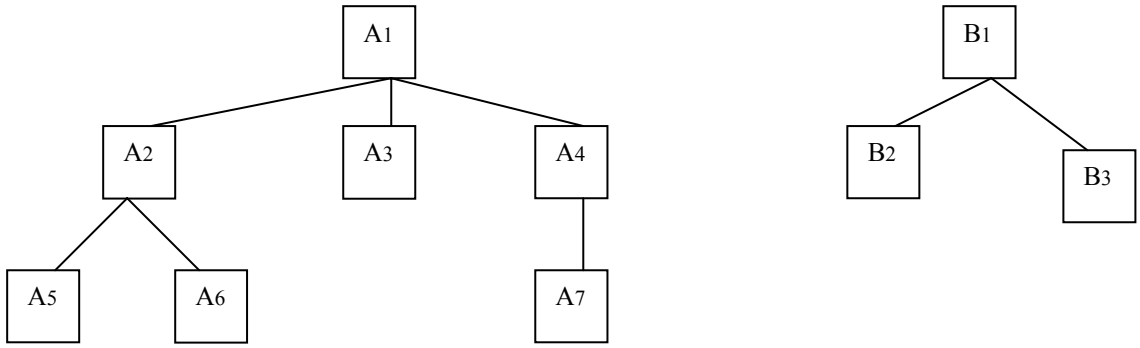

Рис. 5. Приклади структур діаграм класів

На рис. 6 наведено приклад використання об'єктів у програмі, яку відображено горизонтальною лінією.

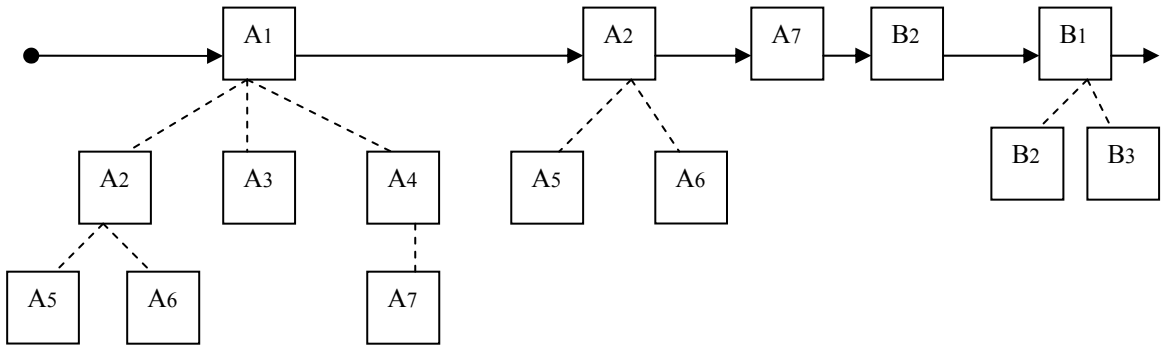

Рис. 6. Приклад схеми використання об'єктів у програмі

Якщо треба переробити наведені на горизонтальній лінії (у програмі) об'єкти, то, можливо, прийдеться переробляти й коди нижчих за ієрархією об'єктів, тому що не виключено, що вони використовують змінні й методи вищих за ієрархією класів. При цьому, в силу властивості інкапсуляції, змінні й методи вищих за ієрархією класів не корегуються. Тільки за рахунок цього вартість заміни коду при об'єктній архітектурі у декілька разів менша, ніж при блочно-модульній, що була описана вище, не говорячи вже про те, що код програми при об'єктній архітектурі за обсягом набагато менший при створенні великого за обсягом коду програмної системи. Для оцінки трудомісткості заміни при використанні технології ООП пропонується така модель. Хай програмний продукт має множину дерев, що відображають ієрархії діаграм класів `D={D1,D2,...,Dn}`, кожне з яких представляє дерево `Dj = (Vj,Ej)`, де: `Vj` – множина вершин дерева; `Ej` – множина ребер (дуг) дерева. Для подальших розміркувань можна поки що опустити нижній індекс (`j`) і розглянути приклад з одним деревом `D=(V,E)`, вершини дерева якого відповідають певним об'єктам.

Кожен об'єкт будемо характеризувати трійкою <Vi, Pi, Mi>, де ідентифікатор об'єкту, `Pi` = {pi1,pi2,...,piqi} – множина змінних, `Mi = {mi1,mi2,...,mi}` – множина методів `i`-го об'єкту. Хай потрібно буде переробити об'єкт `Vi`. Тоді необхідно проаналізувати, чи використовуються змінні й методи цього об'єкту в об'єктах, що

знаходяться на нижчих рівнях ієрархії. Використовуючи дерево ієрархії класів, як наприклад на рис. 4, можна побудувати ланцюжки із вершин, що лежать на шляхах від даної до листів дерева:

$$\{v^1 I, v^1 i+1,...,v^1 i+k_1\}, \{v^2 i, v^2 i+1,...,v^2 i+k_2\},...,\{v^c i, v^c i+1,...,v^c i+k_c\},$$

яким відповідають множини змінних

$$\{P^1 i, P^1 i+1,...,P^1 i+k_1\},\{P^2 i, P^2 i+1,...,P^2 i+k_2\},...,\{P^c i, P^c i+1,...,P^c i+k_c\}$$

і методів

$$\{M^1 i, M^1 i+1,...,M^1 i+k_1\},\{M^2 i, M^2 i+1,...,M^2 i+k_2\},...,\{M^c i, M^c i+1,...,M^c i+k_c\}.$$

Хай для об'єкта vi виконується заміна змінних

$$\{p1,p2,...pn\} \subset \{P^1 i, P^1 i+1,...,P^1 i+k_1\},$$

тоді треба виконати переробку в тих об'єктах, де наведені змінні також використовуються. Для виявлення таких об'єктів треба виконати операції перерізів:

$$\{p1,p2,...,pn\} \cap \{P^2 i, P^2 i+1,...,P^2 i+k_2\},...,\{p1,p2,...,pn\} \cap \{P^c i, P^c i+1,...,P^c i+k_c\}.$$

Аналогічно при заміні методів для виявлення об'єктів, де, можливо, теж потрібна переробка коду, виконуються операції перерізу множин

$$\{m1,m2,...,mn\} \cap \{M2i, M2i+1,...,M2i+k_2\},...,\{m1,m2,...,mn\} \cap \{Mci, Mci+1,...,Mci+k_c\}.$$

Хай треба перепрограмувати модуль vi, тоді трудомісткість можна оцінити як

$$B = B1i + B2i,$$

де: $B1i$ – трудомісткість заміни (перепрограмування та тестування) самого об'єкта vi;

$B2i$ – трудомісткість перепрограмування ділянок коду об'єктів нижчих рівнів, для яких перерізи із змінюваними змінними й методами (див. вище) є непустими. Якщо порівняти з оцінками трудомісткості для модульно-блочної архітектури, то вона може бути у багато разів меншою.

Крім того, що набагато зменшуються обсяг та, відповідно, вплив змін, побудова ПЗ на основі об'єктів має ще ряд переваг:

- доступ системи до ресурсів і робота з ними стають уніфікованими. Наприклад, в ПЗ САПР [7] створення, видалення та посилання на об'єкт-конструкцію здійснюється таким же чином, як і для об'єкта-точки: з використанням *описувачів* об'єктів.
- Спрощується захист, так як для усіх об'єктів він здійснюється однаково. Так, для наведеного прикладу із САПР при спробі доступу до об'єкта підсистема захисту втручається і перевіряє припустимість операції незалежно від того, чи є об'єкт конструкцією, точкою чи конструктивною лінією. Тому перенесення такого програмного продукту на іншу платформу стає набагато простішою задачею.

3. Формальний опис пошарової моделі. Сучасні програмні системи – це досить складні композиції різноманітних функцій, яким відповідають програмні модулі або об'єкти. При цьому примітивні функції можуть складати композиції, які виконують певні узагальнені функції, ті, в свою чергу, можуть об'єднуватися в нові композиції, тощо. Для того, щоб сукупність модулів та/або об'єктів можна було переглянути й зрозуміти, уведено певну пошарову їх структуризацію, при якій модуль, розташований в певному шарі, викликає модуль тільки з нижчого сусіднього шару [11, 12, 9, 10]. Однією з переваг пошарової архітектури є те, що код кожного шару має доступ тільки до необхідних йому інтерфейсів та структур даних нижчих шарів і тим самим зменшується обсяг коду, що має "необмежену владу". Крім того, така структура дозволяє при налагодженні починати із самого нижнього шару та додавати по одному рівню до тих пір, поки уся система не стане працювати правильно. Пошарова структура полегшує також і розширення систем: можна цілком замінити будь-який рівень, не чіпляючи решту частин ПЗ. Структуру такого ПЗ також можна моделювати матрицею

зв'язності, як і блочно-модульну систему. На рис. 7 показано загальний вигляд такої матриці для програмного продукту, розгалуженого на чотири шари S1, S2, S3, S4.

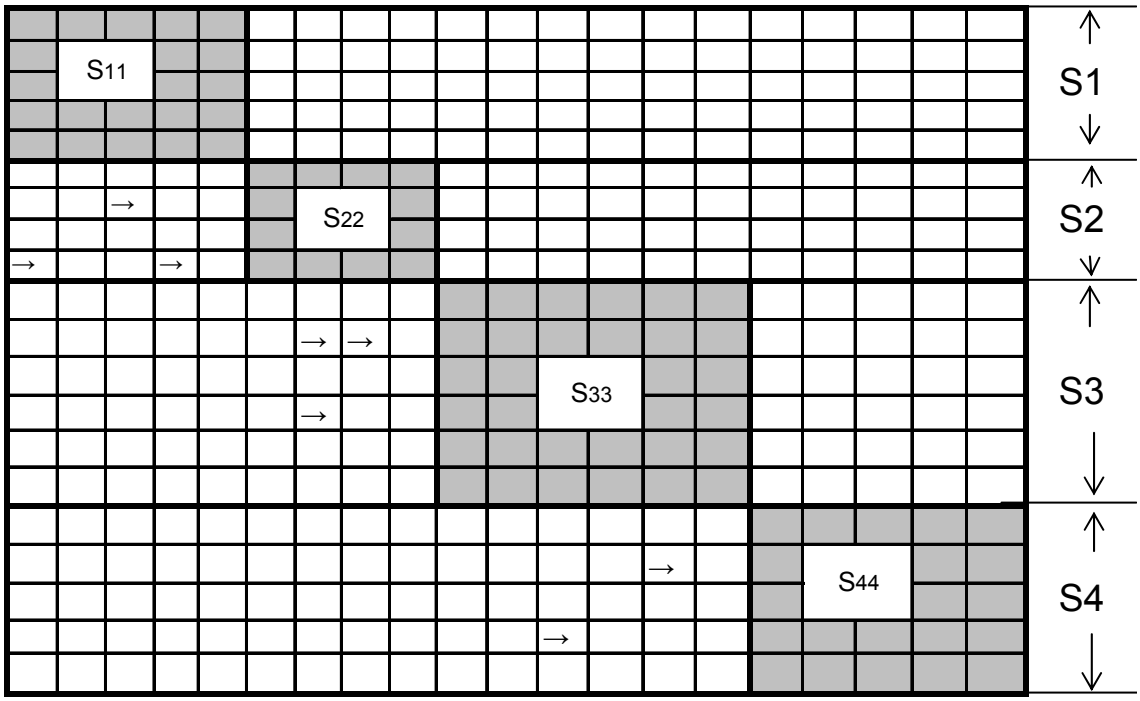

Рис. 7. Приклад загального вигляду матриці зв'язності пошарової структури ПЗ

Кожному шару відповідає своя матриця зв'язності $S_{11}, S_{22}, S_{33}, S_{44}$. Стрілочками вказано, який модуль з Sii викликає модулі з Si+1,i+1. Як правило, в архітектурі програмної системи шари представляють рівні абстракції. Шар n+1 використовує шар n, тому абстракції понять шару n+1, в кращому разі, не нижче, ніж у шару n, а взагалі його рівень абстракції має бути вище. Відповідно, шар n ховає (інкапсулює) логіку роботи, що визначена на цьому шарі, дозволяючи, таким чином, шару n+1 реалізувати роботу з більш складними поняттями, організовувати більш складну логіку, використовуючи засоби шару, що лежить нижче. Як видно з цього рисунку, встановлення самих зв'язків визначається у матрицях зв'язностей Si,i+1, тому кількість зв'язків між модулями є набагато меншою, чим була б, якби не було їх розшарування. Така мінімізація інтерфейсів дозволяє підвищити мобільність ПЗ. Однак, звичайно, зустрічаються ситуації, коли строга пошарова структура системи порушається [10]. Як правило, це є наслідком ерозії архітектури (архітектурним дефектом) і її усунення у більшості випадків здатне принести чутливу вигоду. Але така "не строгість" пошарової архітектури може бути пов'язана з необхідністю внесення змін не тільки в найближчому нижньому шарі, але й в інших нижніх шарах. Так, якщо в базу даних САПР вносяться зміни в таблиці, що моделюють геометричний об'єкт, то це потребує знайти відповідне відображення у кожному нижньому шарі аж до графічного [9].

Перспективи подальших досліджень. Виконане дослідження дозволяють сформулювати задачі мінімізації інтерфейсів при розробці складного й об'ємного програмного забезпечення, а також створювати метабазу або репозиторій систем програмної інженерії.

ЛІТЕРАТУРА

1. Сычов А., Шильников П. CALS-технология как средство интеграции CAD/CAM/CAE/ систем // CAD/CAM/CAE observer. – 2002. – №2. – С. 70-79.
2. Mansurov, D. Campara, Klocwork, Managed Achitecture of Existing Codeas a Practical Transition towards MDA, N 1 Chrysalis Wat, Ottawa, Canada, K2G6P9 – 1999.
3. Соммервил И. Инженерия программного обеспечения. Пер. с англ. – М.: Издательский дом "Вильямс", 2002. – 324 с.
4. Липаев В.В., Филинов Е.Н. Мобильность программ и данных в открытых информационных системах. – М.: Научная книга, 1997.
5. Recommended Practice for Architectural Description of Software-Intensive Systems, ANSI/IEEE Std 1471, 2000.
6. Буч Г. Объектно-ориентированный анализ. – М.: Бином, 1998. – 560 с.
7. Биков Д.П., Фісун М.Т. Застосування об'єктно-орієнтованої методології у САПР корпусів суден. Збірник наукових праць УДМТУ. – Миколаїв: УДМТУ, 2002. – С. 126-132.
8. Лаврищева Е.М. Современные методы программирования: Возможности и инструменты // Проблеми програмування/ – №2-3 спеціальний випуск. – Київ: НАН України, Інститут Програмних Систем, 2006. – 60-74 с.
9. Биков Д.П., Фісун М.Т Пошарова модель розробки програмного забезпечення САПР суден // Наукові записки НУКМА, т.22 у трьох частинах. Частина III : Природничі науки. – К.: Вид. дім "КМ Академія". – 2003. – С. 477-483.
10. Мартин Фаулер. Архитектура корпоративных программных приложений. – Москва: Вильямс, 2004.
11. Многоуровневое структурное проектирование программ. Теоретические основы инструментарий / Е.Л. Ющенко, Г.Е. Цейтлин, В.П. Грицай, Т.К. Терзян. – М.: Финансы и статистика, 1989. – 208 с.
12. Бабенко, Лавріщева К.М. Основи програмної інженерії: Навчальний посібник. – К.: Знання, 2001. – 269 с.
13. Редько В.Н., Сергиенко И.В., Стукало А.С. Прикладные программные системы. Архитектура, построение, развитие. – К.: Наукова думка, 1992. – 480 с.
14. Лаврищева Е.М., Грищенко В.Н. Сборочное программирование. – К.: Наукова думка, 1991. – 216 с.
15. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приёмы объектно-ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2001. – 368 с.
16. Павлов А.А., Теленик С.Ф. Информационные технологии и алгоритмизация в управлении. – К.: Техніка, 2002. – 344 с.