УДК 681.3.06

А.Л. Колесник

МЕХАНІЗМИ ЗАБЕЗПЕЧЕННЯ ВАРІАБЕЛЬНОСТІ В СІМЕЙСТВАХ ПРОГРАМНИХ СИСТЕМ

Визначено основні кроки та проблеми процесу забезпечення варіабельності сімейств програмних систем. Запропоновано загальну модель внутрішньої структури готового ресурсу повторного використання, за допомогою якої виділяються базові компоненти механізмів варіації. Проаналізовано множину зовнішніх і внутрішніх механізмів варіації та надано рекомендації щодо їхнього застосування розробниками готових ресурсів повторного використання в сімействах програмних систем.

Вступ

Інженерія сімейств програмних систем (СПС) є наразі важливим і широко використовуваним підходом до розробки програмних продуктів. Основна ідея даного підходу полягає у створенні програмного продукту з базових елементів, які є готовими ресурсами (ГОР) СПС і можуть використовуватися багаторазово при створенні програмних систем (ПС). Підхід спрямований на покращення якості, прискорення випуску програмної продукції, зниження вартості і збільшення продуктивності праці розробників ПС (у порівнянні з послідовною розробкою ПС) за рахунок використання базових елементів.

Витоки парадигми СПС – у сфері промисловості, де вперше з'явилися лінійки продуктів, орієнтовані на конкретні ринкові сегменти або призначені для вирішення конкретних задач виробництва. Прикладом лінійки продукту може служити літак Боїнг, список деталей моделі 767 якого на 60% складається з деталей моделі 757, що дозволяє значно економити на його виробництві та супроводженні [1].

В такій сфері діяльності, як програмування, побудова ПС з множини базових елементів у парадигмі СПС стала чи не найефективнішим засобом для забезпечення еволюційності ПС, в основі якого лежить варіабельність базових елементів.

Варіабельність, у загальному випадку, це властивість об'єкта до розширення, змінювання, пристосування або конфігурування з метою використання у визначеному контексті та забезпечення подальшого його еволюціонування.

В програмній інженерії питання забезпечення варіабельності ПС належать до області досліджень методів конфігурування продукту (product configuration) та процесів керування конфігурацією (configuration management), спрямованих на визначення і впровадження ефективних процедур його зміни або налаштування з метою отримання програмного продукту з певною функціональністю і характеристиками якості [2].

Зазвичай розглядають три аспекти забезпечення варіабельності:

– моделювання варіабельності на найвищому рівні характеристик об'єктів, які мають варіанти (варіантні характеристики). Метою підходів моделювання є подання варіабельності у такий спосіб, який покращує розуміння проблем забезпечення варіабельності та керування нею;

– реалізацію варіабельності на наступних рівнях архітектури об'єктів за допомогою механізмів варіації. Проблемою забезпечення варіабельності є місце (рівень) її реалізації: чи буде вона реалізована в момент визначення архітектури СПС, чи це буде зроблено під час фази розробки індивідуального продукту ПС. Відповідне рішення приймається розробником СПС з урахуванням власного досвіду створення ефективних механізмів варіації;

– керування варіабельністю, пов'язане з її плануванням, контролюванням та регулюванням у ході життєвого циклу (ЖЦ) об'єктів, яким властива варіабельність.

Базовими елементами або готовими ресурсами СПС, які можуть варіюватися, є складові архітектури, документація, спе-

цифікація, програмні компоненти, моделі виконання, графіки розробки, описи процесів, плани тестування і робіт тощо. При побудові кожного базового елемента треба визначати, яка частина базового елемента буде однакова для всіх систем, в які вона буде включена, а яка змінюватиметься; вибирати механізм варіації; надавати відповідні інструкції для розробників щодо конфігурування ПС з базових елементів або побудови її засобами генерувального програмування [3].

Переваги застосування базових елементів у СПС полягають у тому, що вони можуть бути приєднані до розроблюваної системи без попередньої адаптації, що досягається саме завдяки використанню механізмів варіації у ГОР. Це, в свою чергу, дає можливість «дешево» долучати нову функціональність до ПС у відповідності з вимогами.

У даній роботі стисло охарактеризовано основні кроки та проблеми процесу забезпечення варіабельності СПС, детальніше розглянуто аспект реалізації варіабельності та запропоновано підходи до забезпечення варіабельності базових елементів, які є програмними артефактами СПС, розробленими в парадигмі об'єктно-орієнтованого програмування.

Кроки процесу забезпечення варіабельності СПС

Варіабельність – здатність сімейства ПС, окремої системи або артефакту до розширення, змінювання, пристосування або конфігурування з метою використання у конкретній предметній області.

Варіабельність має забезпечуватися на рівнях вимог до ПС, моделі характеристик СПС, архітектури, коду, документації, тестів тощо. Загалом, вона може бути реалізована як у СПС, так і в конкретних ПС. У першому випадку забезпечується варіабельність виробничої лінії, яка підтримує існування сімейства як множини ПС. Вона стосується множини базових елементів будь-якої природи (програмних і не програмних), які містяться у репозитарії виробничої лінії. У другому випадку забезпечується варіабельність складових ПС (включаючи документацію), що обумовлює еволюційність ПС після «відлучення» від СПС. В обох випадках реалізація варіабельності спрямована на забезпечення життєздатності СПС і ПС [4].

Для того щоб забезпечити бажану варіабельність СПС, треба виконати наступну послідовність кроків [5].

Крок 1. Ідентифікація варіабельності СПС. Найпростіше ідентифікувати варіабельність через моделювання систем в термінах характеристик і визначення характеристик, які матимуть варіанти, так званих варіантних характеристик (ВХ). Тобто варіантна характеристика – абстракція, що передбачає наявність сукупності варіантів і точок варіації. Для моделювання можуть застосовуватися методології FODA, FORM, RSEB, FAST тощо [6].

Ідентифікована варіантна характеристика має статус неявної, не реалізованої у СПС, існуючої лише як концепт у моделі характеристик. Впродовж свого життєвого циклу вона багаторазово трансформується до тих пір, поки не буде реалізована програмно у ПС.

Крок 2. Обмеження варіабельності СПС. Спрямоване на забезпечення достатньої гнучкості СПС з урахуванням поточних потреб і перспектив еволюції у найбільш ефективний за витратами спосіб. На цьому кроці ВХ проходить процедуру введення в СПС і перестає бути неявною. Введена в СПС характеристика має подання в архітектурі і реалізації СПС у формі множини точок варіації – місць в архітектурі і реалізації ГОР, які разом надають механізми, необхідні для забезпечення варіабельності характеристики. На момент введення у СПС конкретні варіанти характеристики можуть бути відсутніми.

Обмеження варіабельності стосується прийняття рішень з таких питань:

– час введення варіантної характеристики в архітектуру і реалізацію СПС,

– час і спосіб приєднання варіантів до ПС,

– вибір моменту зв'язування варіанта для кожної точки варіації, тобто моменту, коли точка варіації конкретизується певним варіантом ВХ (і перестає існувати як така у ПС).

Таким чином, на цьому кроці визначаються всі особливості варіантних характеристик, які впливають на спосіб їхньої реалізації.

Крок 3. Реалізація варіабельності. Залежно від прийнятих обмежень щодо варіабельності може бути вибрана відповідна (найбільш ефективна) технологія реалізації точок варіації, пов'язаних з певною варіантною характеристикою. На цьому кроці для введеної ВХ визначаються долучені варіанти, для кожного з яких реалізуються програмні сутності, які разом здатні «закрити» всі точки варіації. Залежно від того, як реалізована точка варіації, вона буде відкритою для долучення варіантів на різних стадіях розроблення ПС, або закритою. Тобто, нові варіанти вводитимуться лише на певних етапах розроблення ПС.

На наступній стадії приймається рішення стосовно того, який варіант ВХ буде використовуватися, і ПС конкретизується до вибраного варіанта через посилання на конкретні програмні сутності.

Таким чином, ЖЦ варіантної характеристики, як об'єкта СПС, включає наступний перехід її станів (рис. 1): концептуальне подання (неявна ВХ), множина точок варіації у СПС (введена ВХ), множина варіантів програмних сутностей (ВХ як множина варіантів реалізації), конкретний фрагмент ПС, який реалізує задану характеристику (зв'язана ВХ).

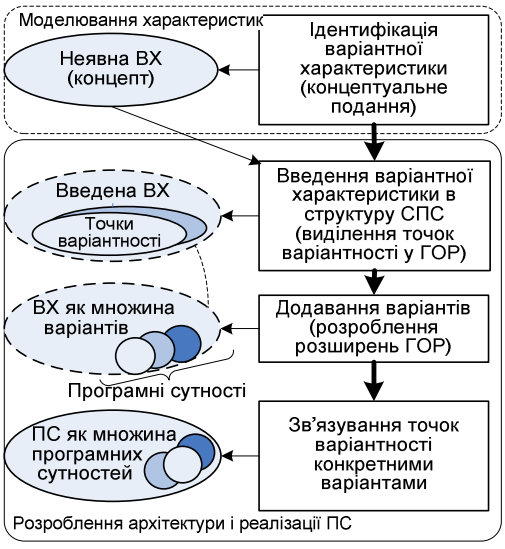


Рис. 1. Процес забезпечення варіабельності

Крок 4. Керування варіабельністю. В результаті виконання кроків 1 – 3 із забезпечення варіабельності у середовищі СПС накопичуються артефакти, створювані впродовж ЖЦ ВХ для всіх розроблюваних ПС. Щодо них мають застосовуватися супроводжувальні дії, як-от поповнення варіантних характеристик новими варіантами, видалення не використовуваних варіантів тощо, а також, видалення ВХ разом з усіма варіантами при зміні вимог, видалення «слідів» старих ПС у СПС тощо. Разом ці дії стосуються регулювання рівня варіабельності СПС і пов'язані з іншими діями з керування варіабельністю – плануванням, обліком і контролем варіабельності.

Проблеми забезпечення варіабельності

Введення варіантної характеристики в архітектуру. Варіантні характеристики можуть бути введені на всіх етапах ЖЦ системи, від архітектурного та детального проекту, до реалізації, компіляції і компонування (лінкінгу, від linking) [5]. Одна ВХ може відбиватися на множину різнотипних програмних сутностей, які разом забезпечують бажану функціональність. Вона проявляє себе у СПС множиною точок варіації на різних рівнях абстракції. Рішення щодо введення ВХ, приймаються з огляду на таке:

– доступність (готовність) технології реалізації, яка задовольняє потребам щодо моментів зв'язування та долучення варіантів;

– розміри програмних сутностей, яких стосується введення ВХ;

– кількість точок варіації;

– вартість супроводження введених точок варіації.

Якщо точка варіації введена на ранніх стадіях ЖЦ ВХ, вона потрапляє під контроль на всіх наступних етапах розроблення, в іншому випадку – вона контролюється лише протягом нетривалого часу.

Якщо точка варіації відкрита для долучення нових варіантів до їх колекції у цій точці, в будь-який момент часу варіанти можуть бути долучені або вилучені. Якщо точка варіації закрита, колекція варіантів зафіксована і не може змінюватися.

Долучення варіантів у точках варіації. Рішення щодо того, коли і як долучати варіанти, приймається з огляду на бізнес стратегію і модель виробництва ПС на лінії. Якщо бізнес стратегія передбачає можливість пізнього долучення варіантів, наприклад, сторонніми постачальниками, це обмежує вибір технологій реалізації точок варіації, оскільки може виникнути потреба залишати їх (точки) відкритими для долучення нових варіантів після компіляції або навіть у період виконання ПС.

На те, як і коли долучати варіанти, впливає також процес розроблення ПС і інструменти, використовувані для розроблення [2].

Долучення варіантів може виконуватися двома шляхами, залежно від того, як реалізовано точку варіації. Варіанти можуть бути долучені неявно, і це означає, що не існує подання колекції варіантів у ПС. Колекцією керують за межами системи з використанням, наприклад, простих переліків наявних варіантів. У цьому випадку забезпечення належного варіанта з колекції неявних варіантів покладається на розробників або користувачів. При явному долученні варіантів колекція варіантів присутня у складі початкового коду ПС, і це означає, що в системі достатньо інформації для того, щоб за потреби самостійно знайти необхідний варіант.

Зв'язування варіанта в точці варіації. Головна мета введення варіантних характеристик – відкладення на подальші стадії розроблення ПС рішення щодо конкретної характеристики, яку треба реалізувати. Конкретизація СПС до необхідного варіанта асоціюється зі зв'язуванням варіанта з незмінною складовою ПС у точці варіації.

Розрізняють внутрішнє і зовнішнє зв'язування. При внутрішньому зв'язуванні програмна система включає функціональність, асоційовану із зв'язуваним варіантом. Таке зв'язування відбувається у період виконання. Зовнішнє зв'язування передбачає, що існує особа або інструмент, які фактично виконують зв'язування.

Рішення щодо того, яким має бути зв'язування – внутрішнім або зовнішнім, обумовлюється тим, виконують його розробники ПС чи кінцеві користувачі, а також тим, чи має воно бути прозорим для користувачів.

Як і з долученням варіантів, час, коли потрібне зв'язування, обмежує вибір можливих шляхів реалізації точки варіації. Якщо варіантна характеристика вводиться через багато точок варіації, можуть виникнути проблеми, оскільки точки варіації мають зв'язуватися або одночасно (як у випадку зв'язування в період виконання), або синхронізовано (коли, наприклад, точка варіації, зв'язувана при компіляції, асоціюється з варіантом, який був зв'язаний у точках варіації на рівні архітектури).

При прийманні рішень щодо часу зв'язування слід керуватися правилом: чим пізніше виконується зв'язування, тим воно витратніше. Відкладання зв'язування з часу побудови архітектури на час компіляції зумовлює необхідність для розробників керування всіма варіантами під час реалізації, а відкладання зв'язування з часу компіляції на час виконання зумовлює необхідність включати функціональність зв'язування і це коштує дорожче в термінах ефективності виконання зв'язування.

Внутрішня структура повторно використовуваного ресурсу

Рекомендації щодо побудови зовнішніх специфікацій повторно використовуваних інформаційних ресурсів надає OMG у документі «Reusable Asset Specification (RAS)» [7]. Призначення цих специфікацій – спрощення процедури об'єднання (пакетування) інформації щодо ГОР для її розповсюдження і пошуку через уніфікацію структури, вмісту і опису ГОР різних категорій.

Для усвідомленого вибору і впровадження механізмів реалізації варіабельності у програмних ГОР СПС, виконуваного розробниками цих ГОР, пропонується використовувати іншу модель (рис. 2).

Модель внутрішньої структури ГОР показана на рис. 2 діаграмою класів в об'єктно-орієнтованому стилі, але це не означає, що ГОР має бути написаний об'єктно-орієнтованою мовою. На цій діаграмі базовий елемент є основним класом, який представляє ГОР і взаємодіє з ПС.

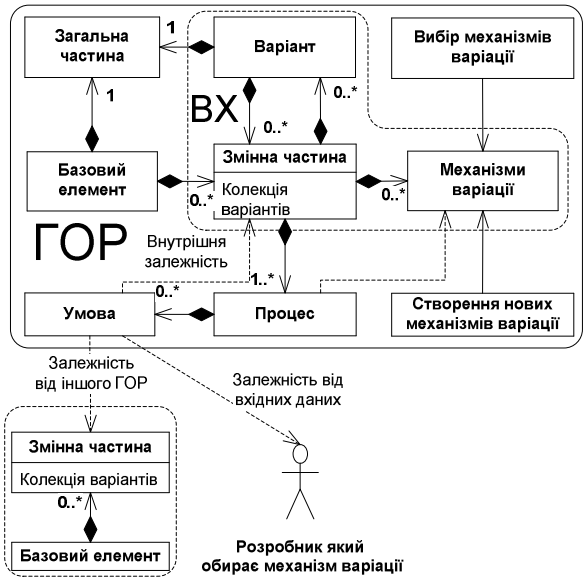


Рис. 2. Внутрішня структура ГОР

Іншими словами, об'єкт класу базовий елемент і є ГОР. Базовий елемент включає в себе загальну частину і (необов'язково) змінну частину. Якщо базовий елемент не має змінної частини, його використовують безпосередньо шляхом включення до конкретної ПС. Призначенням змінної частини є локалізація змін у конкретному місці ГОР. Відсутність загальної частини означає, що всі ГОР мають бути розроблені для кожної ПС індивідуально, що суперечить концепції ГОР.

Подальші пояснення щодо внутрішньої структури ГОР, а також механізмів забезпечення його варіабельності, супроводжуватимемо прикладом, в якому як ГОР виступатимуть елементи управління (ЕУ) графічного інтерфейсу користувача (GUI), які за зовнішнім виглядом і змістом виконуваних дій можуть адаптуватися до вимог користувача. Отже, для ЕУ загальною частиною може виступати механізм графічного виводу ЕУ на екран, оскільки для всіх варіантів ЕУ він буде однаковим. Змінну частину представлятиме набір параметрів, які модифікують поведінку і вигляд ЕУ.

Кожна змінна частина може містити декілька механізмів варіації, які підтримують створення або вибір варіантів. Механізм варіації може бути інтегрований безпосередньо в базовий елемент. Прикладом такої інтеграції є спосіб зміни параметрів ЕУ, його розміру, кольору тощо. Змінна частина представляє точку варіації, а механізми варіації виступають засобом конкретизації ГОР.

У багатьох випадках змінна частина включає два механізми: один для створення нового варіанта, а інший – для підтримки вибору існуючого.

Механізм варіації включає все необхідне розробникам ПС для внесення будь-яких важливих змін. У разі його відсутності, відповідні ГОР не будуть широко використовуватися в СПС, а це свідчитиме про не координованість дій процесу розроблення ГОР і процесу розроблення ПС з використанням ГОР [2]. У прикладі з ЕУ механізм варіації надає можливість для розробників ПС змінювати його властивості, а для розробників ГОР – долучати нові.

Результатом застосування механізму варіації до змінної частини ГОР є варіант, який міститься в загальній частині і може бути включений до змінної частини. Зазвичай, поява нового варіанта зумовлюється застосуванням нового механізму варіації. Наприклад, додаючи новий механізм заповнення даними (binding – механізм зв'язування) ЕУ типу список, отримуємо новий варіант поведінки базового елемента. В подальшому ЕУ здатен працювати у всіх ПС, які використовують цей механізм.

Щойно створений варіант для однієї ПС може бути використаний в інших продуктах через застосування процедури вибору механізму варіації.

Для підтримки використання механізмів варіації при побудові ПС на основі СПС пропонується в опис змінної частини включати інформацію щодо процесу використання та умов її адаптації до потреб ПС. Процес використання пояснюватиме, як використовувати механізми варіації для створення або вибору варіанта для розроблюваної ПС. Опис процесу може містити умови використання. Якщо умов немає, то він є безумовним і виконується кожного разу, коли створюється нова ПС із базових елементів. Умови використання змінної частини визначають залежність від колекції варіантів, які впливають на конкретний продукт. Колекція може бути доповнена варіантами з іншої змінної частини або задана розробниками ПС і відображатиме,

яку альтернативу було обрано для вирішення проблеми в конкретній ПС.

Можливі різні види залежності змінних частин, а саме: внутрішня залежність – це залежність однієї змінної частини ГОР від його іншої змінної частини, якщо така є, і зовнішня залежність – це залежність від змінної частини іншого ГОР.

Основні підходи до реалізації варіабельності програмних ГОР

З точки зору розробника ПС підхід до реалізації варіабельності вибирається виходячи з потреб зовнішнього або внутрішнього зв'язування варіантів, а спосіб впровадження конкретного механізму варіації – з огляду на застосовні парадигми, стилі і навіть мови програмування.

1. Підходи і механізми зовнішнього зв'язування варіантів. Для зовнішнього зв'язування варіантів можуть використовуватися такі механізми варіації, як макроси, умовна компіляція коду, конфігурування та генерування коду, організація статичних бібліотек та динамічне приєднання бібліотек. Розглянемо їх детальніше.

Макрос – це програмний об'єкт, який при обробці «розгортається» в послідовність дій або команд. Макроси повністю або частково підтримуються в Ada, C, C++, D, Erlang, Haskell, Common, Lisp, Nemerle, Ruby, Ocaml. Функціональність макросу розділена на дві частини. Загальна знаходиться в тілі визначення макросу, а варійовані частини можуть перебувати в різних частинах коду та містити варійовані параметри, як у наступному прикладі:

```
// Macros.h Macros попередження
#define РеєстрПопереджень(повідомлення)
  cout << " Попередження у строчці: " <<
  __LINE__ << ": " << (повідомлення) << endl;

// Будь який клас в ПС. *.cpp
#include "Macros.h"  // Включаємо макрос.
БудьЯкийКлас::Метод() {
  РеєстрПопереджень("Повідомлення");};
```

Умовна компіляція коду. Цей механізм варіації забезпечує можливість контролю за включенням або виключенням сегмента коду із збірки, використовуючи параметри компіляції. Приклад:

```
string s;
#ifdef WIN32
s = …;
#ifdef VXWORKS
s = …;
#endif
return s;
```

Тут представлена змінна частина і варіаційний механізм. Варіант, результат варіації, зберігається у змінній «s».

Конфігурування. Вихідний код кожного варіанта знаходиться в окремому файлі, а для вибору альтернатив використовується інструмент керування конфігурацією.

Генерування. Генератор конвертує задану специфікацію, написану мовою предметної області або компонентною мовою, у вихідний код, а вихідний код, у свою чергу, може бути використаний як частина продукту. Приміром, графічний інтерфейс користувача може бути згенерований з графічної (*Silverlight*) або текстової специфікації.

Організація статичних бібліотек. Статичні бібліотеки містять набір зовнішніх функцій, які можна використовувати в програмному застосуванні після його компіляції. При цьому код з бібліотеки інтегрується в виконуваний файл. Варіабельність досягається шляхом вибору відповідної бібліотеки перед її приєднанням. У Unix-системах, наприклад, бібліотеки в основному розташовуються в директоріях /lib, /usr/lib, /usr/local/lib, а у Windows-системах – в \System32, і мають тип файлу "ocx" або "lib".

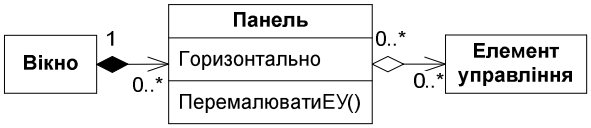
Динамічно приєднувані бібліотеки. Даний механізм варіації заснований на використанні DLL-бібліотек, які можна помістити в адресний простір програми під час її роботи. Керування варіабельністю досягається шляхом реалізації різних варіантів коду в окремих бібліотеках та написання вихідного коду, який завантажує певні варіанти в програмне застосування під час його роботи. Отже, немає необхідності знати всі можливі варіанти під час розробки програми. У Unix-системах, наприклад, вони називаються спільно використовуваними об'єктами (shared objects) і мають тип файлу "so". Mac OS X також успадкували від BSD роботу з бібліотеками. Вони знаходяться в директоріях "bundles" і мають тип "dylib".

2. Підходи і механізми внутрішнього зв'язування варіантів. При описі механізмів, призначених для внутрішнього зв'язування, вважатимемо, що ПС розроб-

ляється в парадигмі об'єктно-орієнтованого програмування, де кожний компонент ПС представлений класом із власними даними і методами їх обробки, а отже, всі варіації, з якими користувач має справу, при реалізації являють собою маніпуляції класами або іншими конструкціями використовуваної мови програмування.

Пропонуються наступні підходи до реалізації варіабельності: композиція, успадкування, параметризація, перевантаження операторів і методів, динамічне завантаження класів, рефлексія, шаблони (паттерни) архітектури і проектування. Стисло охарактеризуємо кожний з них.

Композиція. Механізм варіабельності засновується на встановленні зв'язку між об'єктами типу «part of» і розміщенні загальної частини в делегуючому класі, а змінної – в делегованому. На рис. 3 показана UML схема та відповідний фрагмент коду, що описує поведінку системи у випадку зміни розташування панелі з елементами управління у вікні.




```
class Вікно {
    List<Панель> панелі = new List<Панель>();
    public Вікно() {
        foreach (Панель панель in панелі) {
            панель.Горизонтально = true;}}}
class Панель {
    List<ЕлементУправління> ЕлементиУправління
        = new List<ЕлементУправління>();
    public bool Горизонтально {
        get { return горизонтально; }
        set {горизонтально = value;
             ПеремалюватиЕУ();}}
    private bool горизонтально;
    private void ПеремалюватиЕУ(){/*...*/}}
class ЕлементУправління {/*...*/}
```

Рис. 3. Композиція

Вікно не містить запрограмованої логіки щодо відображення і групування різних ЕУ. Вікно делегує цю функцію панелі. Команда `панель.Горизонтально = true;` є загальною частиною, яка виконується однаково не залежно від типу панелі й ЕУ. Змінна частина знаходиться у методі `ПеремалюватиЕУ()`, який містить відповідну логіку розташування і відображення для різних ЕУ.

У даному прикладі загальною частиною є інтерфейс роботи з властивістю `Панель.Горизонтально` і методом `ПеремалюватиЕУ()`, які містять механізм варіації. Ці компоненти реалізовані в усіх інших породжених типах панелей і ЕУ відповідно, а отже змінною частиною є вміст кожного з цих методів і властивостей. Варіантами є об'єкти, що зберігаються в змінних: `List<ЕлементУправління> ЕлементиУправління` і `List<Панель> панелі`.

На рис. 3 зв'язок між класами панель і ЕУ є прикладом агрегації об'єктів – панель може містити декілька ЕУ; зв'язок між класами вікно і панель – прикладом композиції об'єктів, де до відношення «part-of» додається умова, що панель належить тільки одному вікну і знищується разом з ним. Узагальненням обох відношень (агрегації і композиції) є асоціація.

Успадкування. Механізм базується на такому принципі об'єктно-орієнтованої парадигми, як успадкування через встановлення відношень «нащадок-предок» з можливістю породжувати один клас об'єктів від іншого із збереженням всіх властивостей і методів класу-предка, а, отже, і керувати об'єктами, не маючи повної інформації про них.

Цей механізм розділяє загальну функціональність у базовому класі і доповнену в класі спадкоємця. Як мінімум успадкування поділяють на наступні категорії:

– *Стандартне успадкування*. Передбачає створення ієрархічної структури коду, що містить більш загальну функціональність на верхньому рівні (у базовому класі) для подальшого використання її на нижньому рівні, доповненому варійованою функціональністю (класи спадкоємці). Таким чином, класи стають більш спеціалізованими. Приклад стандартного успадкування класів показано на рис. 4.

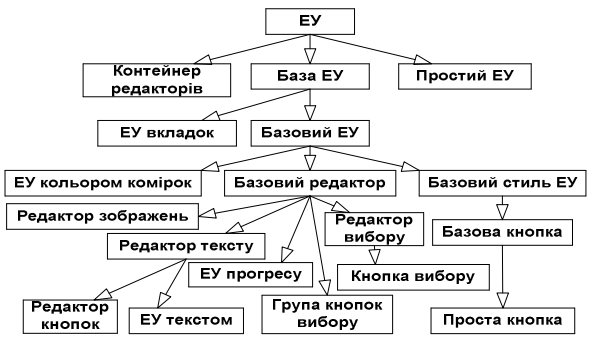


Рис. 4. Приклад успадкування класів в GUI

– У*спадкування з віртуальними методами*. Механізм, подібний до стандартного успадкування, але методи базового класу можуть бути перевантажені класом спадкоємцем.

– *Множинне успадкування*. При такому механізмі реалізації варіабельності новий клас успадковується від декількох базових.

У прикладі на рис. 5 клас керування містить алгоритми сортування та пошуку і не залежить від елементів, над якими виконується операція. Такий тип успадкування підтримується в мовах програмування: Eiffel, C, Dylan, Python, Perl, Curl, Common Lisp (через CLOS), OCaml.

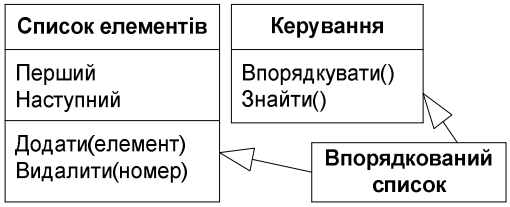
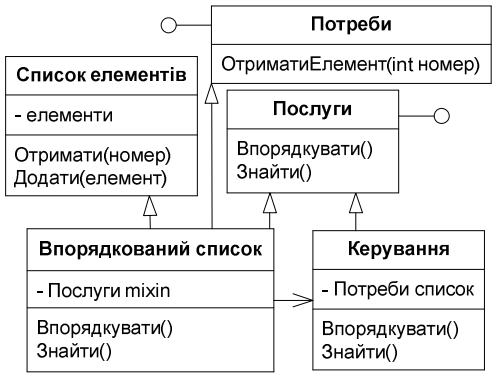


Рис. 5. Множинне успадкування

– У*спадкування, засноване на домішках*. Домішка (mix) – елемент мови програмування (абстрактний підклас або модуль), який реалізує варіант поведінки батьківського класу або модуля. При реалізації варіабельності він використовується для уточнення поведінки інших класів і не призначений для породження самостійно використовуваних об'єктів. Мови програмування, які підтримують домішки: Flavors, Smalltalk, Beta, CLOS, Ruby, D, Python, Scala.

На рис. 6 показана UML-схема реалізації успадкування з домішкою та відповідний фрагмент коду. На схемі інтерфейс потреби описує, через які методи домішка отримує інформацію від предка. Будь який клас, в якому буде використовуватись домішка, має реалізовувати методи інтерфейсу послуги. Класи Алгоритми1 і Алгоритми2 у даному випадку є домішками. У класі впорядкований список змішуються обидва класи.

В цьому фрагменті точка варіації представлена в коді командою `this.mixin = new Алгоритми1(this);`. Якщо виникне необхідність змінити алгоритм сортування або пошуку, який реалізовано в класі Алгоритми1, достатньо змінити Алгоритми1 на інший клас, успадкований від інтерфейсу послуги.




```
interface Послуги {
    void Впорядкувати();
    object Знайти(/*елемент*/);}
interface Потреби {
    object ОтриматиЕлемент(int номер);}
class Алгоритми1 : Послуги {
    private readonly Потреби список;
    public Алгоритми1(Потреби список) {
        this.список = список;}
    public void Впорядкувати() {
        /*Алгоритм впорядкування списку який
        знахо-диться в parent*/}
    public object Знайти(/*елемент*/) {
        /*Алгоритм пошуку в
        список.ОтриматиЕлементи();*/}}
class Алгоритми2 : Послуги {
    private readonly Потреби список;
    public Алгоритми2(Потреби список) {
        this.список = список;}
    public void Впорядкувати() {}
    public object Знайти(/*елемент*/){/*...*/}
}
class СписокЕлементів {
    private object[] елементи;
    public СписокЕлементів() {
        елементи = new object[10]; }
    public object ОтриматиЕлемент(int номер) {
        return елементи[номер]; }
    public void ДодатиЕлемент(object елемент){
        /*Додаємо елемент */ }}
class ВпорядкованийСписок:
        СписокЕлементів, Потреби, Послуги {
    private readonly Послуги mixin;
    public ВпорядкованийСписок(): base() {
        if(/*обмежена пам'ять*/){
            this.mixin = new Алгоритми1(this);
        } else if (/*повільний процесор*/) {
            this.mixin = new Алгоритми2(this);
        }}
    public void Впорядкувати() {
        mixin.Впорядкувати();}
    public object Знайти(/*елемент*/) {
        object елемент =
            mixin.Знайти(/*елемент*/);}}
```

Рис. 6. Імітація множинного успадкування

Змінна частина представлена класами Алгоритми1 і Алгоритми2, загальна – інтерфейсами потреби і послуги та класами Список елементів і ВпорядкованийСписок. Вибраний варіант зберігається у змінній `private readonly Послуги mixin`, а механізмом варіації є оператор if

в конструкторі `public Впорядкований Список()`.

– У*успадкування, засноване на об'єктах, або шаблон делегування (delegation pattern*). Це механізм варіації, який, на відміну від інших, застосовується на рівні об'єктів, а не класів. Це спосіб, яким об'єкт зовні демонструє деяку поведінку, але фактично передає відповідальність за виконання відповідної дії іншому об'єкту, як у наступному прикладі, де `Принтер` делегує її об'єкт типу `СправжнійПринтер`.

```
class СправжнійПринтер { //делегований
    void друкувати() {/*..*/}}

class Принтер { // делегуючий
    СправжнійПринтер принтер =
        new СправжнійПринтер();
    void print() {
        принтер.друкувати();}}
```

Параметризація. В об'єктно-орієнтованому програмуванні параметризованим може бути клас, структура, інтерфейс або метод, у якому є посилання на місце конкретизації параметрів одного або декількох типів, якими він оперує або які зберігає (конструкція на зразок *placeholders (type parameters)*). Цей механізм варіації передбачає наявність загального коду для роботи з різними типами даних. Яскравими представниками є шаблони та універсальні шаблони. Вони є синтаксично і функціонально подібними, але відрізняються внутрішнім механізмом реалізації. Приклад використання можна знайти на рис. 3, це змінні: `List<Панель>` і `List<ЕлементУправління>`. Вони можуть зберігати масиви об'єктів будь-яких типів, породжених від типів панелі і ЕУ відповідно. При цьому алгоритм роботи залишається не змінним.

Перевантаження. Сутність цього механізму полягає у варіації типів, над якими виконується оператор, або за якими викликається функція. Тобто мовою програмування надається можливість повторного використання імені для оперування іншими типами. Вибір варіанта здійснює програмне середовище мови програмування. Застосовується до процедур, функцій або операторів. Мови, що підтримують перевантаження: Ada ,C++, C#, D, Erlang, F#, Groovy, Java, JavaScript, Haskell, Common Lisp, Nemerle, Scala, VB.NET, Delphi, Ocaml. Мова Java не підтримує перевантаження операторів, але виконує внутрішнє перевантаження оператора "+" для поєднання рядків. В C#, наприклад, можна перевантажувати методи, унарні і бінарні оператори.

Використання цього механізму можна розглянути на прикладі методу `public object Знайти(/*елемент*/)`(рис. 6). Система сама вирішить, що повертати як результат: знайдений елемент чи його позицію у списку, якщо розробник додасть ще один метод з такою ж назвою але іншим вихідним параметром, `public int Знайти(/*еле-мент*/)`.

Динамічне завантаження класів. Сутність механізму полягає у тому, що клас завантажується в пам'ять лише тоді, коли він дійсно починає використовуватися, як реалізовано, наприклад, у віртуальній машині мови Java. Для керування цією функціональністю використовується `(MainClass.class.getClass Loader ();)`

Відображення. Механізм заснований на здатності програми модифікувати власну поведінку. Такі маніпуляції досягаються завдяки поділу ПС на базовий і мета рівні. Мета рівень надає інформацію про обрані системні властивості. Оскільки базовий рівень, що включає логіку програми, будується на мета рівні, зміни мета рівня впливають на поведінку базового рівня. У СПС загальна функціональність розташована на базовому рівні і може мінятися під час роботи залежно від конфігураційної інформації, яка використовується на мета рівні.

Приклад реалізації відображення мовою C#:

```
//Без відображення
    Університет унів = new Університет ();
унів.СписокФакультетів();
//Відображення
Type type = Type.GetType(
     "UnivercityNamespace. Університет ");
object univ = Activator.CreateInstance(t);
type.InvokeMember("СписокФакультетів ",
BindingFlags.InvokeMethod, null, univ, null);
```

Шаблони (або Паттерни) архітектури і проектування (*patterns*) можуть використовуватися в СПС, оскільки більшість з них надають готові рішення для управління варіабельністю. Застосування шаблонів покриває деякі попередні механізми, такі як: агрегація, успадкування,

параметризація. У прикладі на рис. 7 показано шаблон міст (*bridge*).

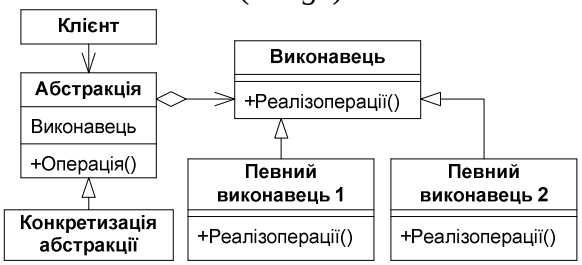


Рис. 7. Шаблон

Клас абстракція визначає абстрактний інтерфейс для клієнта і містить змінну виконавець типу виконавець. Конкретизація абстракції розширює інтерфейс, визначений абстракцією. Виконавець визначає інтерфейс для певного виконавця. Він не повинен повністю відповідати інтерфейсу абстракції, тобто він може бути повністю відмінним. Типовою є ситуація, коли виконавець надає тільки примітивні операції, а абстракція визначає операції високого рівня, що базуються на примітивних. Певний виконавець визначає певні (конкретні) реалізації простих операцій. Накладаючи цей шаблон на вищепредставлену схему ГОР, можемо побачити, що абстракція є загальною частиною, змінна виконавець зберігає варіант, у функції операція() знаходиться механізм варіації, а клієнт є іншим ГОР, залежним від загальної частини, і не залежить від інтерфейсу певних виконавців.

Розглянуті механізми варіації хоча і не покривають всю проблемну область забезпечення варіабельності, але є найбільш використовуваними у парадигмі об'єктно-орієнтованого програмування. Частково вони можуть бути використані в інших парадигмах, а деякі з них навіть концептуально закладені в ці парадигми.

Висновки

Розглянуто кроки процесу забезпечення варіабельності, сформульовано проблеми щодо введення варіантних характеристик в архітектуру СПС, долучення варіантів та їх зв'язування у точках варіації. Запропоновано загальну модель внутрішньої структури ГОР, за допомогою якої виділяються базові компоненти механізмів варіації. Проаналізовано 6 зовнішніх і 7 внутрішніх найбільш поширених механізмів варіації та надано рекомендації щодо їхнього застосування розробниками ГОР СПС. Результати дослідження використовуватимуться при побудові системи підтримки прийняття рішень щодо вибору можливих механізмів варіації у контексті парадигм програмування, операційних систем та мов програмування, а також для автоматизованого аналізу ГОР і оцінювання варіантних характеристик СПС з точки зору наявності реалізованих варіантів, повноти охоплення проблем еволюції ГОР, а також можливості долучення нових варіантів.

1. *Northrop L.M., Clements P.C.* A Framework for Software Product Line Practice, version 5.0 – http://www.sei.cmu.edu/productlines/index.html
2. *Лавріщева К.М., Коваль Г.І., Слабоспицька О.О., Колесник А.Л.* Особливості процесів керування при створенні сімейств програмних систем // Проблеми програмування. – 2009. – № 3 – С. 40 – 49.
3. *Лавріщева К.М.* Генерувальне програмування програмних систем і їх сімейств // Проблеми програмування. – 2009.– № 1. – С. 3 – 16.
4. *Ігнатенко П.П., Бистров В.М.* Особливості забезпечення життєздатності програмних систем в умовах генеруючого програмування // Проблеми програмування. – 2008. – № 2-3. – С. 270 – 278.
5. *Svanberg M., van Gurp J.* A Taxonomy of Variability Realization Techniques // Practice and Experience . – 2006. – N 8. – P. 705–754.
6. *Domain* engineering methodologies survey – http://www.pnp-software.com/cordet/download/ pdf/GMV-CORDET-RP-001_Iss1.pdf
7. *Reusable* Asset Specification (RAS) – http://www.omg.org/cgi-bin/doc?formal/05-11-02.pdf

Отримано 04.01.2010

Про автора:

Колесник Андрій Леонідович,
аспірант Інституту програмних систем НАН України.

Місце роботи автора:
Інститут програмних систем НАН України
03187, Київ-187,
Проспект Академіка Глушкова, 40.
Тел.: +380 (50) 444 2299
e-mail: swabber@gmail.com