**УД К 004.624**

*Федорченко В. М.*

# КОНЦЕПТУАЛЬНЕ ЗЛИТТЯ ЯК МЕТОД ОРГАНІЗАЦІЇ МІЖПРОЕКТНОГО ПОВТОРНОГО ВИКОРИСТАННЯ АРТЕФАКТІВ

*Повторне використання готових рішень є типовим для інженерної діяльності, і програмна інженерія не є виключенням. Вирішення завдання повторного використання є загальновизнаним підходом до підвищення ефективності створення програмних продуктів. У цій статті описано технологію організації ефективного способу міжпроектного повторного використання артефактів у вигляді множини текстових файлів (вихідні тексти на будь-яких мовах програмування, XML-декларації, темплейти, HTML / CSS-розмітка тощо) шляхом структуризації споріднених програмних проектів у формі дерева узагальнень.*

Сучасні тенденції на ринку програмної про- боку, час створення продукту має бути якодукції дедалі частіше формулюють досить супе- мога менший, з іншого – якість (стабільність, речливі вимоги до постачальників ПЗ: з одного швидкодія, підтримка) на рівні міжнародних

стандартів. І все це на фоні вимоги мати фантастичні можливості налаштування продукту на рівні окремих інсталяцій, які фактично розмивають межу між програмістами та системними аналітиками. Методологічний підхід до вирішення цього надзвичайно складного архітектурного завдання в програмній інженерії такий самий, як і в інших галузях – організація процесів повторного використання будь-яких артефактів, що створюються (та використовуються) в процесах життєвого циклу ПЗ [2]. Але специфіка галузі програмної інженерії ( та проблем, що вирішуються) значно ускладнює використання підходу повторного використання на практиці: адже артефакти (фактично, «дещо») не мають чітко визначених або обмежених фізично форм чи формалізацій, а проведення межі «загальне-часткове» фактично є завданням, вирішення якого іноді лежить у філософській площині.

Більш конкретно підхід до повторного використання в контексті програмної інженерії сформульовано вже давно: «Розроблення систем із компонентів необхідного розміру та повторного їх використання. Розгорнуте розуміння "компонентної системи" через площини коду, вимог, аналітичних моделей, архітектури та тестування. Ус і стадії процесу створення ПЗ мають бути предметом для "повторного використання"» (McIlroy, 1969). Упровадження цієї простої ідеї на практиці є достатньо складним: позитивний ефект вимагає суттєвих змін у корпоративній культурі, процесах створення ПЗ, специфічних інструментів підтримки, а також неабияких професійних навичок у розробників. Навіть коли йдеться про визнану індустрією методологію компонентної розробки (CBD), існують як позитивні приклади впровадження, так і негативні – залежно від того, наскільки вдало було інтегровано концепцію повторного використання в конкретній команді розробників. Економічний ефект використання зазначеного підходу залежить не тільки від способу повторного використання, а й від того, що саме використовується, – це може бути як вдале рішення, так і насичене помилками, недостатньо обмірковане розробником рішення.

У праці основну увагу буде зосереждено на висвітленні технології організації ефективного способу міжпроектного повторного використання артефактів; дослідження проблеми визначення об'єктів для повторного використання залишиться за її межами.

Розглянемо проблему організації міжпроектного повторного використання в такій постановці: нехай деяка програмна система представлена у вигляді множини текстових файлів (вихідні тексти на будь-яких мовах програмування, XML-декларації, темплейти, HTML / CSS-розмітка тощо). Ті частини системи, що мають виключно бінарне представлення (зовнішні бібліотеки, графічні зображення та інші ресурси), винесемо за межі розгляду (вони, по-перше, все одно так чи інакше інтегровані в систему за допомогою дескрипторів, що мають символьне представлення, а по-друге – самі по собі зазвичай мають обмежені можливості для маніпуляцій). Потрібно вирішити задачу організації повторного використання таким чином, щоб *ступінь* повторного використання між «спорідненими» проектами був якомога вищим (що фактично визначатиметься кількістю повторно використаних артефактів у символьному представленні), але при цьому складність підтримки (модифікації, розширення) повторно використаних артефактів має залишатися в прийнятних межах (ідеться про «надлишкові» витрати, пов'язані з додатковими залежностями, які неминуче виникають за повторного використання).

Необхідною умовою можливості повторного використання артефактів між двома проектами є розроблення їх в одному концептуальному просторі (тобто вони мають бути визначені в рамках спільної онтології на деякому рівні абстракції. Дійсно, якщо ця умова не виконується, то неможливо формально визначити, які частини двох проектів є однаковими, а які – різними). На практиці це означає, що ці два проекти повинні мати спільну платформу (інфраструктуру), а рівень абстракції спільної платформи буде природним обмеженням для рівня абстракції артефактів, які ефективно можуть бути повторно використані.

Зрозуміло, якщо два або більше проектів визначені в спільному концептуальному просторі, то їх завжди можна розподілити за рівнями узагальнень у деревоподібну структуру. Коли таке дерево узагальнень сформоване, до нього можна застосувати так званий метод *концептуального злиття* для організації міжпроектного повторного використання артефактів (оскільки цей метод базується на властивостях дерева узагальнень).

Під концептуальним злиттям розумітимемо специфічний спосіб компіляції конкретного проекту на основі мета-інформації з дерева узагальнень; при цьому кожен конкретний артефакт має єдине визначене місце в цьому дереві. Проілюструємо ідею злиття на прикладі (рис. 1).

Подібне розподілення означає, що всі артефакти, які стосуються DB-Layer, містяться в «Проекті з використанням DB-Layer», артефакти з Web-проекту – відповідно у «Web-проекті» тощо.

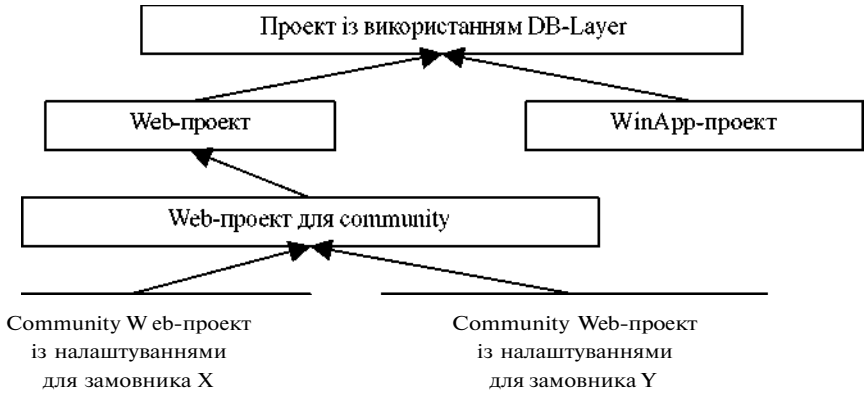


**Рис. 1.** Дерево узагальнень

Наприклад, компіляція «Community Web-проект із налаштуваннями для замовника X» складатиметься з таких кроків:

1. Основа: «Проект із використанням DB-Layer».
2. Злиття «Web-проект» з «батьківським» проектом.
3. Злиття «Web-проект для community» з «батьківським» проектом.
4. Злиття «Community Web-проект із налаштуваннями для замовника X».

Розуміння дії «злиття» залежить від форми представлення артефактів, що повторно використовуються. У цьому разі (проект характеризується множиною текстових файлів) операція «злиття» (merge) в найпростішому випадку може бути визначена так: контекстом операції є множина файлів «основи» ($B$) та проекту, що «зливається» ($T$), результатом є множина файлів ($R$), сформована на основі правил злиття, що застосовуються до кожного файлу $f$ із множини $B \cup T$ ($f$ – файл, $f_X$ – файл із ім'ям $f$ та контентом, що відповідає файлу $f$ у множині $X$, $f^M$ – файл із макроінструкціями для модифікації файлу $f$, $f^I$ – файл із контентом, що посилається на частини контенту файлу $f$):

якщо $f \in B \wedge f^M \in T$, тоді $R = R \cup Modify\ (f, f^M)$;
якщо $f \in B \wedge f^I \in T$, тоді $R = R \cup Inherit\ (f_B, f^I)$;
якщо $f \in B \wedge f \notin T$, тоді $R = R \cup f_B$;
якщо $f \in B \wedge f \in T$, тоді $R = R \cup f_T$;
якщо $f \in T \wedge f \notin B$, тоді $R = R \cup f_B$.

Операція $Modify(f, f^M)$ можлива для файлів, у контенті яких містяться маркери контенту. Ці маркери визначають іменовану частину файлу, вміст якої може бути змінений, або просто конкретну позицію у файлі. Файл із довизначеннями $f^M$ містить додатковий контент для файлу $f$ з метаінформацією про те, де саме розширити або замінити контент файлу $f$ (шляхом посилань на відповідні іменовані маркери). Результатом операції є файл із ім'ям $f$ та контентом, що сформується після всіх операцій вставки та заміни над вихідним контентом файлу $f$.

Операція $Inherit\ (f, f^I)$ також базується на маркерах контенту. На відміну від операції модифікації, вихідним контентом є вміст файлу $f^I$. Він містить один чи більше іменованих маркерів, що визначають, куди потрібно вставити частини контенту, помічені відповідними іменованими маркерами у файлі $f$. Результатом є файл із довільним ім'ям та контентом, що сформується після операцій вставки.

Визначена таким чином операція злиття дає змогу в довільному проекті-спадкоємці довизначити або перевизначити артефакти з батьківського проекту (наскільки вдало це можна зробити, суттєво залежить від розподілення маркерів та власне організації артефактів). Але можлива ситуація, коли артефакт, що може бути повторно використаний у кількох проектах, недоцільно визначати в «батьківському» проекті (оскільки цей артефакт може бути небажаним у інших проектах, що також успадковуються від спільного для них усіх «батьківского» проекту). У цьому разі можна застосувати «множинне» наслідування, а дерево залежностей проектів перетвориться на ациклічний граф залежностей (рис. 2).

Елементарність атомарних операцій (вставка та заміна в текстових файлах) має одну суттєву перевагу: описану вище технологію можна застосовувати майже до будь-якого проекту, незалежно від часу розробки та типу платформи. Але найбільший ефект спостерігається для проектів, що побудовані за принципами компонентної розробки (CBD), оскільки мінімізація взаємозалежностей компонентів дає змогу довизначити або перевизначити фактично будь-який компонент із батьківского проекту з мінімальними зусиллями.

Іншою перевагою запропонованого методу є надзвичайна гнучкість та простота балансування в дереві узагальнень. Для того, щоб «узагальнити»


**Рис. 2.** Граф узагальнень

деякий артефакт, достатньо перемістити деяку підмножину файлів із часткового проекту до одного з «батьківських» проектів; або навпаки – з «батьківского» проекту до одного зі «спадкоємців», якщо виявиться, що «узагальнений» артефакт перестає бути таким. Платою за цю гнучкість є необхідність проводити операції злиття кожен раз, коли змінюється будь-який із «батьківських» проектів. Коли йдеться про відносно невеликі проекти (менше ніж 1000 файлів у результуючому образі), цей процес може займати досить мало часу – кілька секунд; але коли кількість та загальний обсяг файлів зростають, час, що витрачається на «злиття», може спричинити суттєві незручності для розробників. Ця проблема вирішується шляхом створення спеціалізованих інструментів, що аналізують залежності результуючого образу від «батьківських» проектів та реалізують інкрементальний варіант операції злиття.

Зокрема, описаний вище метод концептуального злиття було впроваджено для побудови дерева узагальнень Web-проектів на базі технології Microsoft ASP.NET. Файлову структуру репозиторію визначено таким чином (рис. 3):

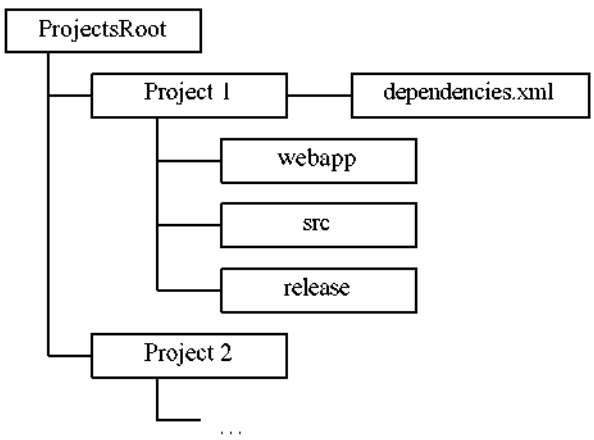


**Рис. 3.** Файлова структура репозиторію проектів

Усі проекти мають власний кореневий каталог: у ньому розміщено каталоги для зберігання вихідних файлів (src) та конфігурації/ресурсів (webapp). Каталог release містить останній зібраний образ проекту. Кореневий каталог проекту може містити файл-дескриптор (dependencies.xml) із описом залежностей від інших проектів; якщо ж цей файл відсутній або не містить посилань на жоден проект - цей проект є основою, кореневим проектом.

Послідовність компіляції проекту може бути описана в будь-якій формі; для проектів, що базуються на платформі Microsoft.NET, для цього доцільно використовувати спеціальні make-інструменти: NANT або MSBuild, що пропонують приблизно однакові можливості та спосіб опису послідовності «компіляції» проектів; обидва інструменти дають змогу без жодних труднощів реалізувати описаний вище метод концептуального злиття. У цьому разі було зручно описати процес злиття за допомогою NANT, а процес компіляції вихідних текстів мовою C# - за допомогою MSBuild.

Маркери контенту є частиною мета-інформації дерева узагальнень, тож їх наявність має бути семантично прозорою відносно первісного контенту. Найлегше цього досягти шляхом визначення синтаксису маркерів на основі «коментарів», що зазвичай доступні як синтаксичний елемент того чи іншого типу контенту. Коли йдеться про контент, типовий для ASP.NET проектів, синтаксис маркерів може бути визначений таким чином (у форматі EBNF):

. XML, ASPNET-файли (aspx, ascx, asax, web.config тощо):

<merge-token> ::= <merge-start> [<content> <merge-end>]
<merge-start> ::= "<!--{{" <token-name> "}}-->"
<merge-end> ::= "<!--{{/" <token-name> "}}-->"
<token-name> ::= {alphabetic character | " " | "-"}

<content> ::= ? any characters sequence excluding <merge-end> ?

• C#, JavaScript-файли:

<merge-token> ::= <merge-start> [<content> <merge-end>]
<merge-start> ::= "/*{{" <token-name> "}}*/"
<merge-end> ::= "/*{{/" <token-name> "}}*/"
<token-name> ::= {alphabetic character | " | "-"}
<content> ::= ? any characters sequence _" before <merge-end> ?

Відповідно файли-модифікатори ($f^M$ та $f^I$) зручно визначити як файли з особливими іменами:

<modify-filename> ::= "#" <base-filename> "#" <token-name>
<base-filename> ::= ? any characters valid for filename except "#" ?
<token-name> ::= {alphabetic character | " " | "-"}

<modify-filename> ::= "@" <base-filename> "@" <destination-filename>
<base-filename> ::= ? any characters valid for filename except "@" ?
<destination-filename> ::= ? any characters valid for filename ?

Наприклад, якщо Project1 містить файл webapp/some.xml із описом деякого компонента:

component name="priceCalculator">
<value-name>value</value-name>
<!-{{priceCalculator-expression}}->
<expression>#value*1.2</expression>
<!-{{/priceCalculator-expression}}->
</component>

Нехай Project2 наслідується від Project1 та містить файл webapp/#some.xml#priceCalculator-expression ($f^M$):

<expression>#value*2</expression>

Тоді після компіляції Project2 файл release/some.xml буде таким:

<component name="priceCalculator">
<value-name>value</value-name>
<!--{{priceCalculator-expression}}-->
<expression>#value*2</expression>
<!--{{/priceCalculator-expression}}-->
</component>.

Структура репозиторію для зберігання артефактів, визначення синтаксису маркерів контенту та створення інструменту, що реалізує основні операції для компіляції та злиття проектів у цьому контексті, – це все, що потрібно для визначення інфраструктури, достатньої для визначення дерева узагальнень.

Життєздатність та економічну доцільність методу концептуального злиття (навіть у описаній вище примітивній формі) підтверджено під час розроблення більш ніж 50 комерційних Web-проектів різноманітного обсягу та складності протягом майже двох років. Концептуальна простота та належна інструментальна підтримка дали змогу впровадити метод концептуального злиття для цілих сервісних ліній споріднених продуктів із мінімальними витратами на підготовку розробників.

Подальший розвиток описаної технології полягає у довизначенні операції злиття на основі додаткової інформації про тип контенту. Наприклад, для XML-файлів можна довизначити правило трансформації на основі XSL:
якщо $f^X \in T \wedge f \in B$, тоді $R = R \cup XslTransform(f_B, f^X)$;
якщо $f^X \in T \wedge f \in T$, тоді $R = R \cup XslTransform(f_T, f^X)$,
де $XslTransform(f, f^X)$ – відповідно XSL трансформація XML-файлу $f$ за допомогою шаблону $f^X$. У цьому разі можливості XSLT відкривають якісно інші можливості для повторного використання артефактів, оскільки операція злиття в цьому випадку може розглядатися як перетворення моделі з одного представлення в інше (у контексті модель-орієнтованого підходу, MDA [3]).


1. *Перевозчикова О. Л.* Основи системного аналізу об'єктів і процесів комп'ютеризації.– К.: Видавничий дім «КМ Академія», 2003.
2. ДСТУ 3918–99 (ISO/IEC 12207:1995). Інформаційні технології. Процеси життєвого циклу програмного забезпечення.
3. *Kleppe A., Warmer J., Bast W.* MDA Explained: The Model Driven Architecture™: Practice and Promise.– New York: Addison-Wesley, 2003.
4. *Hunt A., Thomas D.* Pragmatic Programmer, the: From Journeyman to Master.– New York: Addison-Wesley, 1999.
5. *Szyperski C.* Component Software: Beyond Object-Oriented Programming.– New York: Addison-Wesley Proffessional, 1997.
6. *Seacord R., Plakosh D., Lewis G.* Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices.– New York: Addison-Wesley, 2003.
7. *Брукс Ф.* Мифический человеко-месяц, или Как создаются программные системы.– СПб.: Символ-Плюс, 2000 (original: Addison-Wesley, 1995).

*V. Fedorchenko*

## INTER-PROJECTS ARTIFACTS REUSE USING CONCEPT MERGING TECHNOLOGY

*Reuse of existing solutions is typical for engineering and software engineering is not exception. Increasing effectiveness of software products creation and support is an ultimate goal. Nowadays software reuse is common approach for reaching this goal, so almost all modern methodologies and technologies are concentrated on this problem. In this paper is proposed one more effective cross-project reuse technology for artifacts described as text files collections (source code, XML, templates, HTML/CSS, etc); key idea of this technology is storing similar software projects in generalization tree.*