УДК 004.415.2.045 (076.5)

МЕТОД ПОБУДОВИ МОДЕЛЕЙ ДЕФЕКТІВ ПРОЕКТУВАННЯ ОБ'ЄКТНООРІЄНТОВАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Сидоров М.О., Нечай О.С.

Національний авіаційний університет

nikolay.sidorov@livenau.net

У статті введено поняття ступеня розвитку дефекту проектування об'єктно-орієнтованого програмного забезпечення та запропоновано метод побудови моделей дефектів, що дозволяють його оцінити. Визначено причини виникнення та розвитку дефектів проектування. Опис стану елемента програмного забезпечення, ураженого дефектом, здійснюється шляхом аналізу правил проектування, нормування метрик та побудови функцій за допомогою t-норми та t-конорми. Побудовано модель дефекту God Class та перевірено її адекватність шляхом ручного аналізу отриманих результатів її роботи.

In the paper the notion of object-oriented design defect progress degree is introduced and the method for defect models construction is proposed. Constructed models allow defect degree assessment. Reasons of design defect appearance and development are defined. State description of software element affected by design defect is accomplished by design rules analysis, metrics normalizing and function building on the base of t-norm and t-conorm. God Class defect model is build and its adequacy is approved by manual code inspection.

Витрат

Витрати на супровід програмного забезпечення сягають 85—90 % бюджету інформаційних систем [1]. Значна частина зусиль під час супроводу спрямована на відновлення працездатності програмного забезпечення [2], а одним із явищ яке, відіграє при цьому важливу роль, є розпад програмного забезпечення (*software decay*), або архітектурний дрейф (*architectural drift*) [3]. Для запобігання розпаду необхідно проводити систематичну реструктуризацію програмного забезпечення, що вимагає методів та засобів виявлення відповідних частин програмного забезпечення що її потребують. Оскільки розпад програмного забезпечення виявляється через дефекти проектування, актуальною є задача їх виявлення та визначення ступеня розвитку.

Постановка завдання

У праці розглядаються дефекти проектування об'єктноорієнтованого програмного забезпечення, тому приймемо, що дефект проектування — це невідповідність проектного рішення правилу об'єктно-орієнтованого проектування програмного забезпечення [4].

Причинами виникнення та розвитку дефектів проектування об'єктноорієнтованого програмного забезпечення є:

- обмеження термінів та ресурсів виконання проектних робіт — програмне забезпечення вступає у фазу супроводу, маючи добре спроектовану структуру, але інженери із супроводу стикаються з жорсткими термінами внесення змін і змушені обирати найпростіші проектні рішення, замість тих, що необхідні для збереження цілісності структури та запобігання розпаду програмного забезпечення;
- брак знань архітектури — інженери із супроводу не мають достатньо знань початкової архітектури проекту, а внесення змін до програмного забезпечення вимагає модифікації архітектури;
- супровід успадкованого програмного забезпечення — успадковане програмне забезпечення написане в часи, коли парадигма об'єктно-орієнтованого програмування щойно з'явилась і більшість розробників не мали достатнього розуміння принципів його проектування.

Дефекти проектування виникають і розвиваються під час виконання робіт із супроводу, тому аналіз еволюції дефектів проектування не повинен обмежуватись їх виявленням в одній чи кількох версій програмного забезпечення. Дефекти проектування, у разі їх розвитку, повинні розглядатись як загроза дегенерації до більш небезпечних дефектів, сприяння появи та розвитку інших дефектів. Крім того, необхідно мати можливість відстежувати під час проведення реструктуризації однієї частини системи збільшення рівня розвитку дефектів в інших, тому для контролю супроводу програмного забезпечення необхідні такі моделі дефектів проектування, які були б здатні зафіксувати збільшення або зменшення ступеня їх розвитку. Визначимо ступінь розвитку дефекту проектування як кількісну характе-ристику відхилення проектного рішення від правила проектування.

Метод побудови моделей дефектів. Сутність запропонованого методу побудови моделей дефектів проектування на відміну від інших [4; 5], полягає в тому, що опис стану частини (елемента) програмного забезпечення, ураженого дефектом, здійснюється шляхом нормування метрик та побудови функцій за допомогою *t*-норми та *t*-конорми на основі знань об'єктно-орієнтованого проектування, що дає змогу визначати ступінь розвитку дефекту. Для побудови моделі шляхом застосування запропонованого методу необхідно виконати такі:

– сформулювати правило проектування частини (елемента) програмного забезпечення;

– виконати аналіз правила для визначення ознак його порушення;

– визначити нормовану метрику для обчислення інтенсивності кожної простої ознаки;

– встановити базове значення для кожної з нормованих метрик;

– побудувати функцію для обчислення ступеня розвитку дефекту шляхом комбінування нормованих метрик.

За типом ураженого елемента програмного забезпечення пропонується розрізняти дефекти проектування методу класу, класу та підсистеми. Тому нехай $E_\tau = \{e_{\tau,1}, e_{\tau,2}, e_{\tau,3}, ..., e_{\tau,k}\}$ — множина елементів програмного забезпечення типу $\tau \in T$, а $T = \{method, class, subsystem\}$. Тоді $D_\tau = \{d_{\tau,1}, d_{\tau,2}, d_{\tau,3}, ..., d_{\tau,p}\}$ — множина дефектів проектування що можуть спостерігатись у елемента типу $\tau \in T$. Модель дефекту проектування для оцінювання ступеня розвитку дефекту будується на основі функції тако вигляду: $\varphi_{d_\tau} : E_\tau \to [0,1]$.

Формулювання правила проектування. Онтологія знань об'єктноорієнтованого проектування складається з таких основних груп [6] (рис. 1): *декларативні знання* — концепції які описують правила проектування (евристики, шаблони проектування, принципи проектування, кращі практики (*best practices*) і т. ін.); оперативні знання — описують операції або процеси внесення змін до програмного забезпечення (наприклад перетворення програми, зі збереженням її функціональності).

Аналіз показує, що принципи, евристики кращі практики не мають значних не мають значних відмінностей між собою і мають структуру та форму правила, тобто містять умову і реко-мендацію, що має виконуватись. Надалі під правилом проектування розумітимемо будь-який принцип, евристику чи кращу практику, приведену до вигляду «якщо *умова*, то *рекомендація*». Наприклад, відомий принцип інверсії залежності подається у вигляді такого правила [7]: «якщо e_τ *має залежність від конкретного класу*, то *ця залежність має бути організована через інтерфейс*». Визначимо взаємно однозначну відповідність $violate : R_\tau \to D_\tau$, де $R_\tau = \{r_{\tau,1}, r_{\tau,2}, r_{\tau,3}, ..., r_{\tau,p}\}$ — правила проектування. Якщо $d_\tau = violate(r_\tau)$, $r_\tau \in R_\tau$, то порушення правила r_τ спричиняє дефект проектування d_τ. Згідно з онтологією для виправлення дефекту (виконання правила) може бути використано 0 або кілька шаблонів проектування (рис. 1). Процеси виправлення дефекту визначаються оперативним знанням про реструктуризацію архітектури.

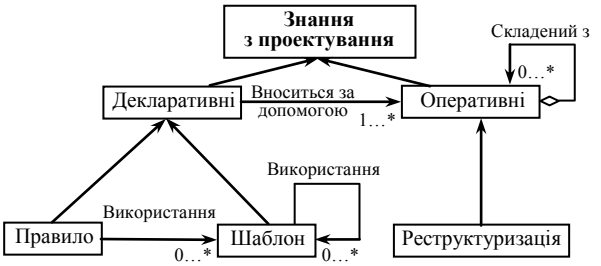

Рис. 1. Онтологія знань у сефрі об'єктно-орієнтованого програмування

Аналіз правила для визначення ознак його порушення. Завданням цього кроку є таке: шляхом покрокової деталізації правила r_τ виділити для кожного $d_\tau = violate(r_\tau)$ множину ознак $S_{d_\tau} = \{s_{d_\tau,1}, s_{d_\tau,2}, s_{d_\tau,3}, ..., s_{d_\tau,m}\}$ того, що елемент e_τ уражений дефектом d_τ. Нехай S_τ є множина усіх ознак, що можуть мати елементи e_τ, тоді $S_\tau = \bigcup_{d_\tau \in D_\tau} S_{d_\tau}$.

Ознака може бути простою або складеною з інших ознак. Простою вважатимемо ознаку, інтенсивність якої можна обчислити за допомогою однієї прямої метрики. Умова правила є ознакою його порушення, з якої починається процес покрокової деталізації. На кожному кроці процесу ознаки, що визнаються складеними, діляться на складові ознаки так, що складена ознака визначається або одночасним проявом складових ознак, або проявом хоча б однієї складової ознаки. Прості ознаки надалі не підлягають поділу. Покрокова деталізація закінчується, коли не залишається неподілених складених ознак. Наприклад, нехай деяка ознака $s_{\tau,1}$ = «дуже складний та слабо зчеплений e_τ» є складеною, оскільки не може бути оцінена однією метрикою. Поділимо її на дві ознаки, що мають одночасно проявлятись: $s_{\tau,2}$ = «дуже складний e_τ» та $s_{\tau,3}$ = «слабко зчеплений e_τ», які є простими, оскільки можуть бути оцінені метриками *Weighted Method Count* (WMC) [8] та Tight Class Cohesion (TCC) [9] відповідно. Результатом покрокової деталізації є граф ознак дефекту d_τ $SG = (S_{d_\tau}, IS\_COMPONENT\_OF)$, де $IS\_COMPONENT\_OF \subseteq S_{d_\tau} \cdot S_{d_\tau}$. Для будь-якого $s_{\tau,i}, s_{\tau,j} \in S_{d_\tau}$ вираз $s_{\tau,i} IS\_COMPONENT\_OF s_{\tau,j}$ означає, що $s_{\tau,i}$ визначається кількома ознаками, одна з яких $s_{\tau,j}$. Граф ознак зображується у вигляді дерева (рис. 2). Вузли, що означають прості ознаки — зафарбовані. Однією аркою показано, що складена ознака визначається одночасним проявом складових ознак, двома арками — проявом хоча б однієї складової ознаки.

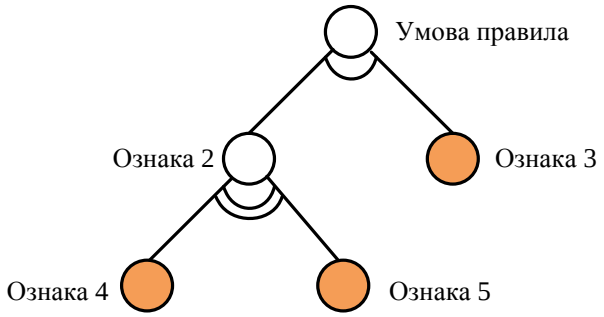

Рис. 2. Граф ознак

Визначення нормованої метрики для обчислення інтенсивності кожної простої ознаки. Для кожної простої ознаки $s_\tau \in S_\tau$ обрати метрику $M'_{s_\tau} : E_\tau \to \Re$, за допомогою якої можна обчислити інтенсивність ознаки s_τ. Якщо оцінюється ознака, для якої неможливо підібрати існуючу метрику, тоді необхідно визначити нову метрику. Для ненормованих метрик потрібно визначити максимальне значення $\max_{s_\tau}$ — деяке значення метрики, ймовірність появи якого дуже низька (< 0.001). Мінімальне значення ненормованої метрики має дорівнювати нулю $\min_{s_\tau} = 0$. Якщо $M'_{s_\tau}(e_\tau) > \max_{s_\tau}$, то значення метрики встановлюється $\max_{s_\tau}$. Для всіх нормованих метрик $\max_{s_\tau} = 1$. Нормовану метрику $M_{s_\tau} : E_\tau \to [0,1]$ знаходимо наступним чином:

$$M_{s_\tau}(e_\tau) = \frac{M'_{s_\tau}(e_\tau)}{\max_{s_\tau}}.$$

Встановлення базового значення для кожної з нормованих метрик. Базове значення Z_{s_τ} нормованої метрики M_{s_τ} відповідає стану елемента програмного забезпечення, коли ознака s_τ не проявляється. Як базові значення рекомендується обирати граничні значення нормованих метрик із множини {0, 1}, тому що вони відображають повну присутність або відсутність ознаки. Наприклад для ознаки «дуже складний клас», інтенсивність якої обчислюється за допомогою метрики WMC, базовим значенням необхідно встановити 0, що відповідає класу з нульовою складністю (дана ознака не проявляється). Інтенсивність ознаки знайдемо за формулою:

$$\alpha_{s_\tau}(e_\tau) = \left| Z_{s_\tau} - M_{s_\tau}(e_\tau) \right|.$$

Побудова функції для оцінювання ступеня розвитку дефекту. Комбінування метрик можна виконати різними шляхами. Це може бути неформальний підхід, наприклад, з використанням лінійчатої діаграми, що показує відразу кілька метрик (рис. 3). У випадку аналізу складних та великих програм, що містять кілька сотень класів, указаний неформальний підхід може виявитись дуже трудомістким, тому необхідний формальний підхід, наприклад, визначення непрямої метрики за допомогою формули. Так, для отримання функції φ_{d_τ} необхідно визначати формули для обчислення інтенсивності складених ознак дефекту d_τ шляхом комбінування формул для обчислення інтенсивності складових ознак. Пропонується використовувати *t*-норму, якщо складена ознака дефекту d_τ визначається одночасним проявом складових ознак, та *t*-конорму, якщо складена ознака дефекту d_τ визначається проявом хоча б однієї з ознак.

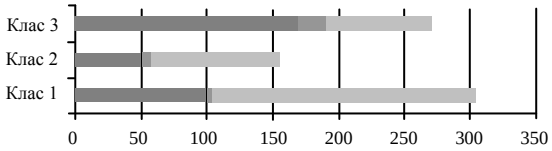

Рис. 3. Рис. 3. Комбінування значень метрик з використанням лінійчатої діаграми: — метрика складності; — метрика зчеплення; — метрика розміру

У праці [10] *t*-норма *T* та *t*-конорма *S* визначаються як функції $T, S : [0,1]\times[0,1] \to [0,1]$, такі, що для усіх $x, y, z \in [0,1]$ виконуються такі умови:

– граничні — $T(x, 1) = x, S(x, 0) = x$;

– комутативність — $T(x,y) = T(y,x)$, $S(x,y) = S(y,x)$;

– асоціативність — $T(T(x,y),z) = T(x,T(y,z))$, $S(S(x,y),z) = S(x,S(y,z))$;

– монотонність — $T(x,y) \le T(x,z)$, $S(x,y) \le S(x,z)$ для $y \le z$.

Побудовані моделі планується використовувати для аналізу еволюції дефектів проектування, тому обрані *t*-норма та *t*-конорма мають відповідати таким вимогам:

– $\forall a,b,c \in [0,1]$ $a \ne c, b \ne 0 \Rightarrow T(a,b) \ne T(c,b)$; $a \ne c, b \ne 1 \Rightarrow S(a,b) \ne S(c,b)$ — необхідна для фіксування змін будь-якої з метрик, що використовуються для моделювання дефекту;

– мінімальна кількість арифметичних операцій — необхідна для оптимізації тривалості обчислень.

Встановленим вимогам відповідають алгебричні *t*-норма $T(a,b) = ab$ та *t*-конорма $T(a,b) = a + b - ab$. Наприклад, якщо деякий дефект $d_\tau \in D_\tau$ визначається проявом ознак $s_{\tau,1} \in S_{d_\tau}$ або $s_{\tau,2} \in S_{d_\tau}$, а $s_{\tau,2}$ — одночасною присутністю ознак $s_{\tau,3} \in S_{d_\tau}$ та $s_{\tau,4} \in S_{d_\tau}$, то об'єд-нуючи їх інтенсивності за допомогою *t*-норми та *t*-конорми маємо:

$$\alpha_{s_{\tau,2}}(e_\tau) = T(\alpha_{s_{\tau,3}}(e_\tau), \alpha_{s_{\tau,4}}(e_\tau)),$$

$$\varphi_{d_\tau}(e_\tau) = S(\alpha_{s_{\tau,1}}(e_\tau), \alpha_{s_{\tau,2}}(e_\tau)) =$$

$$S(\alpha_{s_{\tau,1}}(e_\tau), T(\alpha_{s_{\tau,3}}(e_\tau), \alpha_{s_{\tau,4}}(e_\tau))),$$

$$\varphi_{d_\tau}(e_\tau) = \alpha_{s_{\tau,1}}(e_\tau) +$$

$$+ \alpha_{s_{\tau,3}}(e_\tau)\alpha_{s_{\tau,4}}(e_\tau) - \alpha_{s_{\tau,1}}(e_\tau)\alpha_{s_{\tau,3}}(e_\tau)\alpha_{s_{\tau,4}}(e_\tau).$$

Комбіноване значення функції φ_{d_τ} дає змогу порівнювати елементи програмного забезпечення, оскільки елементи з більшим значенням φ_{d_τ} мають більш розвинений дефект проектування, ніж елементи з меншим значенням φ_{d_τ}.

Застосування методу. Побудуємо модель дефекту *God Class* [11] та визначимо ступінь його розвитку для класів програмної системи *Vuze* (відомий P2P Java клієнт з відкритим вихідним кодом). Технічні характеристики *Vuze* отримані за допомогою інструменту *Plazma* [12] (табл. 1)

Таблиця 1

Технічні характеристики Vuze

Версія	4.1.0.0
Кількість рядків коду	498,108
Кількість методів	37,003
Кількість класів	1,033
Кількість пакетів	478

В успадкованому програмному забезпеченні дефектами *God Class* вражені класи, що концентрують значну частину системної логіки в своїх методах, використовуючи інші класи як постачальники даних, та мають слабке зчеплення. Правило проектування, пов'язане з цим дефектом сформулюємо наступним чином: «якщо клас є складним, слабо зчепленим та отримує доступ до даних інших класів то такий клас необхідно розчепити на менші класи». Виконавши аналіз цього правила, отримуємо такі прості ознаки: s_1 — складний клас, s_2 — слакбо зчеплений клас, s_3 — клас, що отримує доступ до даних інших класів (граф ознак показаний на рис. 4).

Для оцінювання інтенсивності ознак оберемо метрики: s_1 — WMC [8], s_2 — TCC [9], s_3 — ATFD [5]. Метрика TCC — нормована, тому $\max_{s_2} = 1$.

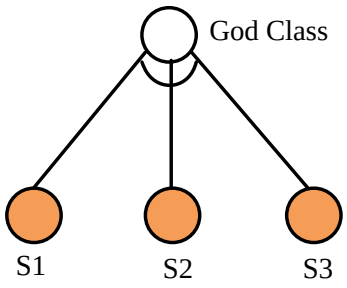

Рис. 4. Граф ознак для God Class

Для того щоб знайти $\max_{s_1}$ та $\max_{s_3}$, було проаналізовано 13 відкритих програмних проектів із загальною кількістю класів 10,543. Системи були обрані різні за розмірами (від 300 класів до 1,500 класів), із різних доменів та розроблені різними групами розробників. У результаті було встановлено $\max_{s_1} = 300$ (значень WMC вище за 300 отримано шість з усієї вибірки), $\max_{s_3} = 20$ (значень TCC вище за 20 отримано дев'ять з усієї вибірки).

Будемо вважати що ознаки s_1, s_2, s_3 не проявляються, якщо $M_{s_1}(c) = 0$, $M_{s_2}(c) = 1$, $M_{s_3}(c) = 0$ де c — досліджуваний клас. Тобто базові значення нормованих метрик мають такі значення: $Z_{s_1} = 0$, $Z_{s_2} = 1$, $Z_{s_3} = 0$.

Функцію для оцінки розвитку даного дефекту *God Class* згідно з графом ознак визначимо наступним чином:

$$\varphi_{godclass}(c) = \alpha_{s_1}(c)\alpha_{s_2}(c)\alpha_{s_3}(c) =$$

$$= \left|0 - M_{s_1}(c)\right|\left|0 - M_{s_2}(c)\right|\left|0 - M_{s_3}(c)\right|.$$

Використаємо отриману модель дефекту для кожного класу обраної для аналізу системи. Основне завдання — виявити класи-претенденти для реструктуризації. Десять класів, що мають найбільшу ступінь розвитку дефекту *God Class* наведені в табл. 2. Видно, що всі виявлені класи активно отримують доступ до даних інших класів, мають дуже велику складність та їх зчеплення близьке до нуля. Клас *BuddyPlugin* має найрозвиненіший дефект. При ручному перегляді коду помітні великий розмір та складність методів. Він має 106 методів та 92 атрибути, але реалізує інтерфейс *Plugin*, що потребує лише один метод. Тобто існуюча велика кількість методів не викликана необхідністю реалізації інтерфейсних методів. *BuddyPlugin* має значення ATFD метрики рівне 31 — отримує доступ до 31-го атрибуту інших класів (беруться до уваги лише класи, що не є членами однієї і й тієї ж ієрархії класів), або напряму, або через спеціальні методи доступу. Усі ці факти свідчать про те, що розглянутий клас використовує більше дані, а не логіку інших класів, делегуючи їм лише повноваження зберігання даних. Таке проектне рішення має негативний вплив на повторне використання та на розуміння цього класу [11].

Таблиця 2

Класи-претенденти для реструктуризації Vuze

№	Клас	ATFD	TCC	WMC	$\varphi_{godclass}(c)$
1	BuddyPlugin	31	0,06	289	0,9055
2	SubscriptionManagerImpl	20	0,03	280	0,9053
3	SideBar	20	0,09	256	0,7765
4	DHTTransportUDPImpl	20	0,18	223	0,6095
5	UDPConnectionSet	14	0,23	289	0,5192
6	PiecePickerImpl	11	0,16	300	0,4620
7	Subscription Impl	16	0,06	181	0,4537

№	Клас	ATFD	TCC	WMC	$\varphi_{godclass}(c)$
8	StartStopRulesDefaultPlugin	9	0,1	281	0,3794
9	SpeedManagerPingMapperImpl	12	0,08	196	0,3606
10	DHTPluginStorageManager	15	0,11	160	0,3560

Висновок. Запропоновано метод для побудови моделей дефектів проектування об'єктно-орієнтованого програмного забезпечення, які на відміну від відомих шляхів обчислень за результатами вимірювань дають змогу визначити ступінь розвитку дефектів. Із застосуванням методу побудовано модель дефекту *God Class* та пере-вірено її адекватність шляхом ручного аналізу отриманих результатів її роботи. Побудовані за допомогою запропонованого методу моделі можуть бути застосовані для аналізу еволюції дефектів проектування об'єктноорієнтованого програмного забезпечення.

ЛІТЕРАТУРА

1. *Erlikh L.* Leveraging legacy system dollars for E-business // *(IEEE) IT Pro.* — 2000. — May/June. — P. 17 — 23.

2. *Фаулер М.* Рефакторинг: Улучшение существующего кода / М. Фаулер: пер. с англ. — СПб : Символ-Плюс, 2003. — 432 с.

3. *Moha N.* A Domain Analysis to Specify Design Defects and Generate Detection Algorithms / N. Moha, Y. Guéhéneuc, Le Meur F., L. Duchien // Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering. — Springer-Verlag, March-April 2008. — P. 276—291.

4. *Lanza M.* Object-Oriented Metrics in Practice / M. Lanza, R. Marinescu. — Springer-Verlag Berlin Heidelberg, 2006. — 205 p.

5. *Garzas J.* Object-oriented design knowledge: principles, heuristics, and best practices / J. Garzas, M. Piattini. — Hershey: Idea Group Publishing, 2007. — 376 c.

6. *Martin R.C.* The dependency inversion principle // C++ Report.— 1996. — May. — P. 61—66.

7. *Chidamber S.* A metrics suite for object oriented design / S. Chidamber, C. Kemerer. // IEEE Transactions on Software Engineering. — 1994. — Vol. 20, No. 6. — P. 476—493.

8. *Bieman J.M.* Cohesion and reuse in object-oriented system / J.M. Bieman, B.K. Kang. // Proceedings of ACM symposium on Software Reusability. — April. — 1995. — P. 259—262.

9. *Schweizer B.* Statistical metric spaces / B. Schweizer, A. Sklar. // Pacific Journal of Mathematics. — 1960. — V. 10, No. 1. — P. 313—334.

10. *Dewayne D.E.* Founddations for the study of software architecture / D.E. Dewayne, A.L. Wolf. ACM SIGSOFT Software Tngineering Notes. — V. 17, Is. 4. — P. 40—52.

Стаття надійшла до редакції 03.03.09