УДК 004.415.2.045 (076.5)

О.С. Нечай, асп.

МЕТОД ДІАГНОСТИКИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Національний авіаційний університет
E-mail: alexander.nechay@livenau.net

Запропоновано метод діагностики об'єктно-орієнтованого програмного забезпечення, який дозволяє інженеру відстежувати еволюцію дефекту проектування і в результаті виявити найбільш небезпечні дефекти або дефекти на ранній стадії розвитку, які можуть стати небезпечними в майбутньому.

дефекти проектування, об'єктно-орієнтоване проектування, правила проектування, якість програмного забезпечення

Вступ

Під час супроводу в програмне забезпечення вносяться зміни, часто в жорстких умовах обмежених термінів та ресурсів. У результаті, його конструкція деградує і, як наслідок, стає складною для розуміння та модифікації. Це явище відоме як розпад програмного забезпечення [1; 2] і є надзвичайно шкідливим, оскільки має тенденцію спочатку бути непоміченим, але згодом наростати з часом [3]. Тому розроблення методів і засобів діагностики програмного забезпечення є актуальним завданням.

Аналіз останніх досліджень та публікацій

Протягом останніх років було розроблено низку підходів до виявлення дефектів проектування [4]. У роботі [5] використовуються запити до моделі програмного забезпечення мовою Prolog для виявлення дефектів.

У роботі [6] запропоновано застосування шаблонів коду для виявлення дефектів проектування.

Застосування правил, що грунтується на метриках для виявлення дефектів проектування запропоновано в роботі [7]. Окрім цього, застосовувались методи візуалізації для розуміння конструкції програмного забезпечення.

У роботі [8] запропоновано предметно орієнтовану мову програмування для опису дефектів проектування. На основі програм, написаних цією мовою, автоматично генеруються алгоритми виявлення дефектів проектування мовою програмування Java.

У цих підходах використано аналіз лише однієї версії програмної системи, тому вони не можуть оперувати корисною інформацією, пов'язаною з історію системи.

Метод, запропонований у роботі [9], дозволяє аналізувати еволюцію елементів конструкції, уражених дефектами проектування, а також використовувати історичну інформацію для підвищення точності виявлення дефектів проектування.

Дефекти проектування виникають і розвиваються кожен по-своєму в результаті внесення змін до програмного забезпечення. І саме спосіб їх розвитку визначає їх небезпечність для еволюції програмного забезпечення. Якщо дефект не змінюється з часом, то він менш небезпечний ніж дефект, ступінь розвитку якого збільшується.

Однак жоден з розглянутих методів діагностики об'єктно-орієнтованого програмного забезпечення не спрямований на виявлення дефектів проектування на етапі їх зародження та спостереження за їх розвитком з метою своєчасного планування робіт з реструктуризації.

Постановка завдання

Мета роботи − розроблення методу діагностики об'єктно-орієнтованого програмного забезпечення, спрямованого на виявлення дефектів проектування на етапі їх зародження та спостереження за їх розвитком.

Для досягнення мети поставлено та вирішуються такі завдання:

– розробити метамодель, яка дозволяє на її основі виконувати аналіз історії дефектів проектування;

– розробити набір графічних зображень, які спрощують відстеження розвитку дефектів у різних аспектах;

– установити, які завдання можна виконати, вивчаючи набір графічних зображень.

Сутність запропонованого методу діагностики об'єктно-орієнтованого програмного за без-печення полягає у моніторингу дефектів проектування.

Метод реалізується шляхом використання запропонованої метамоделі історії дефектів проектування (DDHM – Design Flaws History Model) об'єктно-орієнтованого програмного забезпечення та багатоаспектної візуалізації дефектів проектування елементів конструкції різного рівня абстракції.

У DDHM дефект уперше моделюється як окрема сутність, здатна змінювати свої характеристики з часом. Розглянуто такі кількісні характеристики дефекту: ступінь розвитку дефекту, інтенсивність ознак дефекту та середня інтенсивність ознак дефекту [10]. Для моніторингу дефектів за допомогою запропонованого методу необхідно виконати таке:

– побудувати за допомогою засобів зворотної інженерії екземпляр DDHM;

– побудувати на основі DDHM, використовуючи запропоновані алгоритми, графічні зображення, потрібні для моніторингу дефектів;

– виконати візуальний аналіз отриманих зображень.

Метамодель історії дефектів проектування

Метамодель об'єктно-орієнтованого програмного забезпечення − це опис сутностей програмного забезпечення, їх властивостей і відношень [7].

Для аналізу програмного забезпечення застосовують кілька метамоделей, наприклад, Datrix [11], Famix [12], Dinamix [13], Hismo [14].

За основу запропонованої DDHM обрано метамодель Hismo (рис. 1), оскільки, на відміну від інших розроблено як базову метамодель для побудови на її основі метамоделей історії елементів програмного забезпечення.

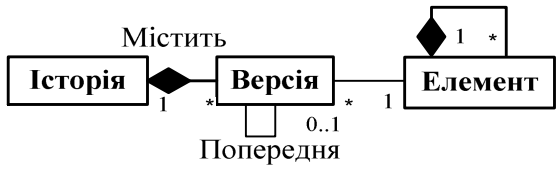

Рис. 1. Метамодель Hismo

Метамодель Hismo складається з таких частин:

– елемент замінюється елементом конструкції, історія якого вивчається, наприклад пакетом, класом або методом;

– версія елемента конструкції програмного забезпечення додає поняття часу до елемента, відносячи його до історії, версія ідентифікується міткою часу та містить ідентифікатор історії, частиною якої вона є, версія містить посилання на нуль або одну попередню версію і нуль, або одну наступну версію;

– історія елемента конструкції програмного забезпечення містить множину версій, версія може належати лише одній історії, тому відношення між історією і версією показано як композицію.

DDHM, побудована шляхом розширення Hismo: як сутності метамоделі, крім версій та історій елементів конструкції програмного забезпечення (метод, клас, пакет), є також версії та історії дефектів (рис. 2).

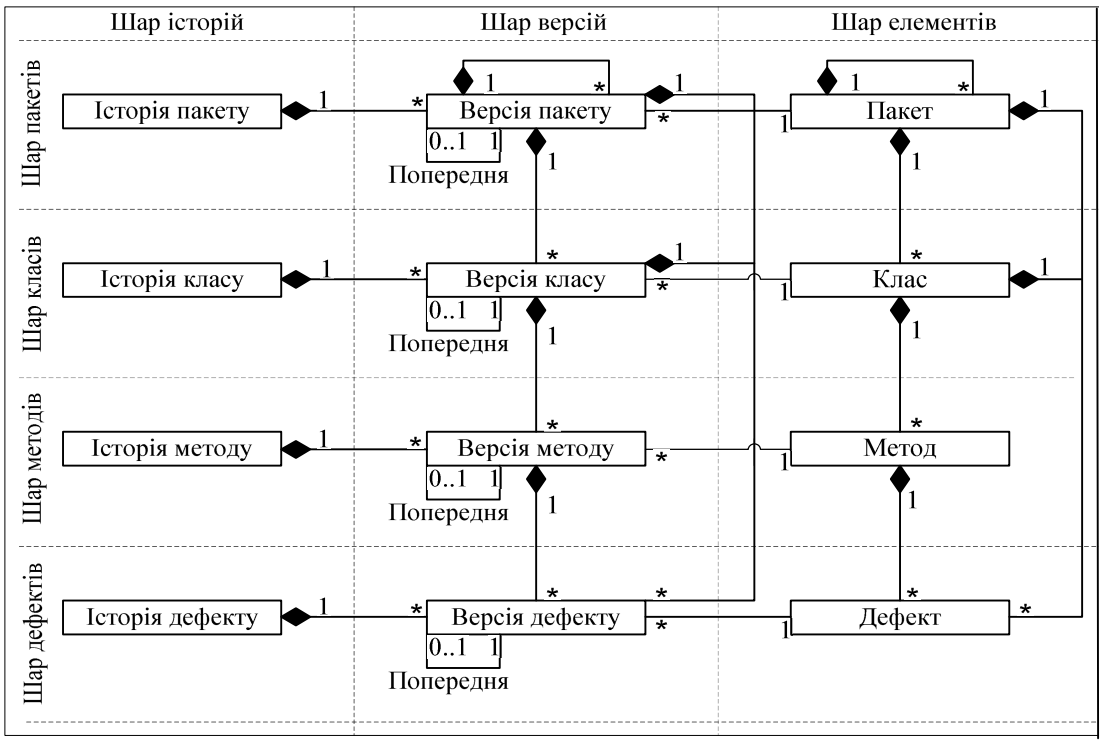

Рис. 2. Метамодель історії дефектів проектування

DDHM має забезпечувати такі можливості:
– автоматичної побудови на її основі графічних зображень;
– взаємодії користувачів з графічними зображеннями;
– обчислення на її основі метрики історій дефектів.

Тому визначимо DDHM як типізований, навантажений орієнтований мультограф

$$MModel = (N, E),$$

де $N = \{n_1, n_2, n_3, ..., n_m\}$ − вузли, які описують сутності моделі;

$E = \{e_1, e_2, e_3, ..., e_k\}$ − ребра, що описують відношення між сутностями моделі.

Мультиграф вказує на те, що два вузли можуть поєднувати кілька ребер.

Типізація використовується власне для задання метамоделі.

Вузли можуть бути вкладені в інші вузли, що вказується ребром типу contains.

Орієнтованість використовується для вказівки того, що кожне ребро має виняткова один вузол-джерело і один вузол-мета.

Оскільки граф навантажений, то кожному вузлу або ребру може бути поставлено у відповідність будь-яку кількість властивостей.Нехай η − множина всіх типів вузлів, η_i − довільний тип вузла (табл. 1), ρ − множина всіх властивостей вузлів, ρ_i − довільна властивість вузла, а ε – множина всіх типів ребер, ε_i − довільний тип ребра (табл. 2).Тоді для скорочення назвемо вузол типу η_i як η_i −вузол, а ребро типу ε_i – ε_i − ребро.

Наприклад system-вузол являє собою цілу об'єктно-орієнтовану програмну систему, а method-вузол − метод класу.

Композицію або агрегацію, спрямовану від контейнера до вмісту, який представляє contains-ребро.

Для позначення ребер напрямлених у зворотний бік використовується стрілка поверх назви типу ребра. Наприклад, $\overleftarrow{\text{contains}}$-ребро означає contains-ребро, напрямлене від умісту до контейнера.

Оскільки використовується навантажений граф, то до будь-якого вузла або ребра можуть бути додані специфічні властивості.

У цій роботі властивості ребер не використовуються. Деякі з властивостей вузлів, що використовуються далі, наведено в табл. 3.

Для виконання маніпуляції над моделлю, необхідно декілька функції, як наведені в табл. 4.

Для подальшої роботи з моделлю визначимо також множину

$$V = \{n \in N \mid type(n) \in \{packV, classV, methodV\}\},$$

$$DT = \{x \mid x - \text{тип дефекта}\};$$

предикат $isEdge : E \times N \times \varepsilon \times N \to Boolean$, який використовується для перевірки наявності ε_1− ребра між вузлами n_1 і n_2, де $n_1, n_2 \in N$, $\varepsilon_1 \in \varepsilon$:

$$isEdge\ (e, n_1, \varepsilon_1, n_2) : source(e) = \\ = n_1 \wedge target(e) = n_2 \wedge type(e) = \varepsilon_1;$$

функцію $Dpd : V \times D \to [0,1000]$, яка використовується для находження ступеня розвитку вказаного дефекту $d \in D$, що міститься у версії елемента конструкції $v \in V$:

$$Dpd\ (v, dt) = \begin{cases} property(dv, dpd), \text{якщо} \\ (\exists dv \in \{n \in N \mid type(n) = defecV \wedge \\ \wedge\ property(dv, defType) = dt\}) \\ (\exists e \in E)\ isEdge(e, v, contains, dv); \\ 0,\ otherwise; \end{cases}$$

функцію $Maxdpd : V \to [0,1000]$ (рис. 3), яка використовується для находження ступеня розвитку найбільш розвинутого дефекту, що міститься у версії елемента конструкції $v \in V$.

Таблиця 1

Типи вузлів моделі

Вид вузла	Опис
system, *systemV*, *systemH*	Програмна система, її версія та історія
pack, *packV*, *packH*	Пакет, його версія та історія
class, *classV*, *classH*	Клас, його версія та історія
method, *methodV*, *methodH*	Метод, його версія та історія
attribute, *attributeV*, *attributeH*	Атрибут класа, його версія та історія
defect, *defectV*, *defectH*	Дефект, його версія та історія

Таблиця 2

Типи ребер моделі

Тип ребра	Опис	Можливі пари типів <джерело, мета>
contains	Джерело містить мету	< *system* , *pack* >, < *system* , *class* >, < *pack* , *pack* >, < *pack* , *class* >, < *class* , *method* >, < *class* , *attribute* >, < *pack* , *defect* >, < *class* , *defect* >, < *method* , *defect* >, < *packV* , *packV* >, < *packV* , *classV* >, < *classV* , *methodV* >, < *classV* , *attributeV* >,< *packV* , *defectV* >,< *classV* , *defectV* >, < *methodV* , *defectV* >, < *packH* , *packV* >, < *classH* , *classV* >, < *methodH* , *methodV* >, < *attributeH* , *attributeV* >, < *defectH* , *defectV* >, < *systemH* , *systemV* >, < *systemV* , *packV* >
version	Джерело є версією мети	< *packV* , *pack* >, < *classV* , *class* >, < *methodV* , *method* >, < *attributeV* , *attribute* >, < *defectV* , *defect* >, < *systemV* , *system* >
prev	Джерело є версією, наступною після версії-мети довільній історії	< *packV* , *packV* >, < *classV* , *classV* >, < *methodV* , *methodV* >, < *defectV* , *defectV* >, < *systemV* , *systemV* >
call	Джерело викликає мету	< *method* , *method* >, < *methodV* , *methodV* >
inherit	Мета є суперкласом історії	< *class* , *class* >, < *classV* , *classV* >

Таблиця 3

Властивості вузлів моделі

Тип вузла	Властивість	Опис
systemV , *packV* , *classV* , *methodV* , *defectV*	*date* *rank*	Дата створення версії Номер верстки в історії
defectV	*dpd* *msi* *defType*	Ступінь розвитку дефекта (defect progress degree) Середня інтенсивність ознак дефекта (mean sign intencity) Тип дефекта

Таблиця 4

Функції маніпулювання над моделлю

Функція	Опис
$source : E \to N$	Для отримання вузла, джерела ребра
$target : E \to N$	Для отримання вузла, мети ребра
$type : N \to \eta$	Для отримання типу вузла
$type : E \to \varepsilon$	Для отримання типу ребра
$property : N, \rho \to PROPV$	Для отримання заданної властивості вузла, *PROPV* – множини значень властивостей вузла

```
procedure Maxdpd(v)
    Maxdpd(v) := 0
    for each
    dv ∈ {n ∈ N | type(n) = defecV ∧
    ∧ ∃e ∈ E  isEdge(e, v, contains, n)}
    do
        if Maxdpd(v) < property(dv, dpd)  do
            Maxdpd(v) := property(dv, dpd)
        end if
    end for
end procedure
```

Рис. 3. Процедура обчислення функції *Maxdpd*

Засоби зворотної інженерії, потрібні для відновлення DDHM, крім відомих методів відновлення моделей об'єктно-орієнтованого програмного забезпечення [15], мають реалізовувати алгоритми пошуку ідентичних сутностей.

Ідентичними називаються сутності, що є в різні моменти часу двома версіями з однієї і тієї ж історії.

Найпростіший метод встановлення ідентичності сутностей полягає у використанні імен.

Якщо існують дві сутності одного типу з однаковим іменем у двох різних версіях системи, то вони розглядаються як дві версії з однієї історії. Цей метод не дозволяє виявити ідентичністьперейменованих або переміщених сутностей. Кілька методів, що вирішують цю проблему, запропоновано в літературі [16; 17] і можуть використовуватися для вдосконаленого відновлення DDHM.

Графічні зображення для моніторингу дефектів

Людина краще сприймає інформацію, у графічному вигляді, ніж текстовому.

Візуалізація програмного забезпечення − це використання типографії, графічного дизайну, анімації, кінематографії та сучасних технологій комп'ютерної графіки для спрощення розуміння та використання програмного забезпечення [18].

Ключову роль у побудові зображень для моніторингу дефектів відіграє ступінь розвитку дефекту [10].

У контексті запропонованого методу зображення показують дефекти проектування елементів конструкцій таких рівнів абстракції: рівня методу, рівня класу, рівня підсистеми, і в таких аспектах:

− історія розпаду програмного забезпечення;

− історія розвитку дефекту проектування;

− історія розвитку ознак дефекту проектування.

Для відображення дефектів проектування в аспекті історії розпаду програмного забезпечення пропонується зображення «Рентгенограма». Це зображення призначено для вирішення таких завдань:

– моніторингу розподілу дефектів по елементах конструкції;

– формування загальної картини розпаду програмного забезпечення.

За основу зображення «Рентгенограма», взято зображення, запропоноване в роботі [19], яке призначено для моніторингу результатів тестування. Розглянемо правила побудови зображення «Рентгенограма» (рис. 4).

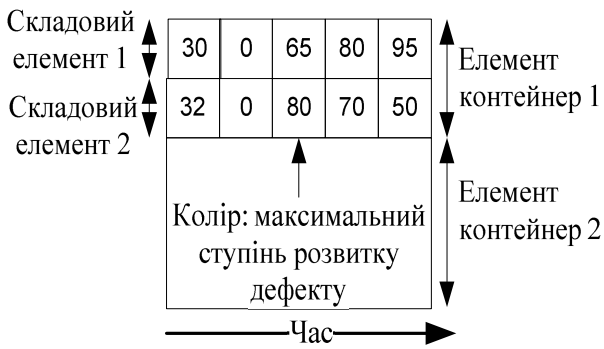

Рис. 4. Правила зображення «Рентгенограма»

Зображення будується на основі матриці. Рядок матриці відображає складовий елемент конструкції програмного забезпечення.

Група рядків відображає контейнерний елемент конструкції, що містить елементи, які подано рядками.

У стовпчику подано версію програмного забезпечення, більш ранні версії розміщено зліва.

Колір прямокутника відображає ступінь розвитку найрозвиненішого дефекту проектування відповідної версії елемента конструкції.

Використовуються відтінки сірого кольору − чим темніший колір, тим більш ступінь розвитку дефекту. На рисунку замість кольору показано ступінь розвитку дефекту. Як контейнерний елемент конструкції користувач може вибрати клас чи підсистему, а як складніший − клас чи метод.

В алгоритмі побудови матриці для зображення «Рентгенограма» використано:

– DDHM;

– вибір зроблений користувачем типу контейнерного елемента конструкції;

– вибір зроблений користувачем типу складового елементу конструкції.

Алгоритм візуалізації містить правила зображення «Рентгенограма» та отриману матрицю.

Зокрема, застосувавши до кожного елемента матриці функцію *Maxdpd* , алгоритм отримує ступінь розвитку найрозвиненішого дефекту. Далі, застосувавши до результату функцію *SelectColor* (задається в табличному вигляді та залежить від параметрів пристрою графічного виведення, настроювань користувача і кольорової моделі), алгоритм отримує код кольору, потрібний для з'ясування прямокутника, що відображає елемент конструкції програмного забезпечення.

Для відображення дефектів проектування в аспекті історії розвитку дефекту проектування пропонується зображення «Історія дефекту».

Виходом алгоритму є матриця, елементами якої є вузли графу, що зображують версії елементів конструкції програмного забезпечення.

Це зображення призначено для моніторингу дефектів проектування певного типу.

Тип елементів конструкції (підсистема, клас, метод) і відповідний тип дефектів проектування вибирається користувачем. Як основу для цього подання вибрано зображення «Evolution Matrix» [20], метою якого є підтримання розуміння еволюції класів об'єктно-орієнтованого програмного забезпечення.

Розглянемо правила побудови зображення «Історія дефекту» (рис. 5).

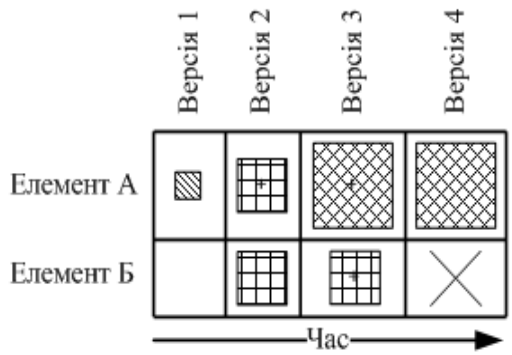

Рис. 5. Правила зображення «Історія дефекту»

Зображення теж будується на основі матриці. Кожен рядок матриці відображає елемент конструкції програмного забезпечення, стовпці − час зліва направо.

Кожен стовпець відображає версію програмного забезпечення.

Прямокутники на перетинах відображають дефекти проектування, а вражені ними версії елементів конструкції програмного забезпечення відповідають перетинам рядків і стовпців матриці.

Три можливі кольори прямокутників відображають ступінь розвитку дефекту відповідно до підділянки, в яку потрапляє її значення.

На ділянці значень ступеня розвитку виокремлюються такі ділянки:

– зелена підділянка − інтервал (0,75], в якому значення ступеня розвитку дефекту далеке від перевищення порога;

− жовта підділянка − інтервал (75,100], в якому значення ступеня розвитку дефекту близьке до перевищення порога;

− червона підділянка – інтервал (101,1000], в якому значення ступеня розвитку дефекту перевищило поріг.

Довжина діагоналі прямокутника відповідає ступеню розвитку дефекту. Якщо прямокутника немає на перетині, то дефекту в цій версії елемента конструкції теж немає.

Знак «+» всередині прямокутника означає, що ступінь розвитку дефекту збільшилася порівняно з попередньою його версією, або залишилася незмінною, але збільшилася середня інтенсивність ознак дефекту.

Таким чином, невеликі зміни ступеня розвитку, які візуально не помітні, все одно будуть зафіксовані.

Наприклад, ступінь розвитку дефекту елемента А (рис. 5) у третій версії збільшився і відповідний прямокутник позначено знаком «+», а в четвертій версії не змінювався, тому знаку немає.

Ступінь розвитку дефекту елемента Б у третій версії не змінився, про що свідчить не змінена, діагональ прямокутника, але знак «+» означає, що середня інтенсивність ознак дефекту збільшилася.

Закреслена рамка в четвертій версії елемента Б означає, що в четвертій версії програмного забезпечення елемента Б немає в результаті видалення або реструктуризації.

Алгоритм побудови матриці для зображення «Історія дефекту» використовує:

– DDHM;

– вибір користувачем типу елементів конструкції, що підлягають візуалізації;

– предикат, що забезпечує фільтр під час формування списку елементів конструкції, що підлягають візуалізації.

Виходом алгоритму є матриця, елементами якої є вузли графу, що зображують версії елементів конструкції програмного забезпечення.

Алгоритм візуалізації використовує правила зображення «Історія дефекту» та отриману матрицю. За допомогою функції *Dpd* , заданих типу дефекту і межі підділянки визначається значення ступеня розвитку потрібного дефекту і підділянки, якій він належить.

Для відображення дефектів проектування в аспекті історії розвитку ознак дефекту пропонується зображення «Історія ознак дефекту». Це зображення призначено для моніторингу ознак дефекту проектування.

Елемент конструкції (підсистема, клас, метод) визначається вибраним користувачем дефектом проектування. Розглянемо правила побудови зображення «Історія ознак дефекту» (рис.6).

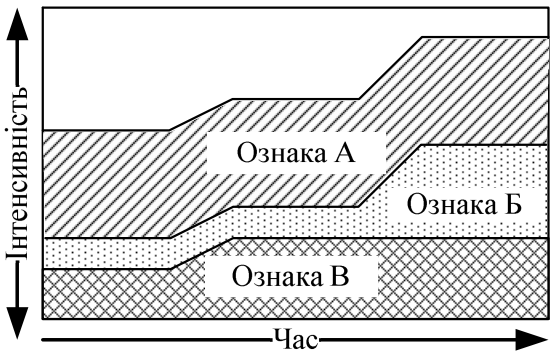

Рис. 6. Правила зображення «Історія ознак дефекту»

Як основу вибрано набір правил інформаційної візуалізації, що успішно використовується як для візуалізації програмного забезпечення [21], так і в інших ділянках [22].

Кожна ознака має призначений їй колір і відображається шаром, товщина якого відповідає інтенсивності прояву ознаки. За горизонтальною віссю відкладається час.

Товщина всіх шарів відповідає сумі інтенсивностей всіх ознак і пропорційна середній інтенсивності ознак дефекту.

В алгоритмі підготовки вхідних даних для зображення «Історія ознак дефекту» використано:

– DDHM;

– вибір користувачем історії дефекту.

Виходом алгоритму є вектор, елементами якого є вузли графу, що являють собою версії дефектів проектування елементів конструкції.

Алгоритм візуалізації, використовує отриманий вектор і за допомогою функції *property* визначає значення інтенсивностей ознак кожного елемента вектора.

Аналіз графічних зображень дозволяє:

– виділити найбільш небезпечні дефекти, що забезпечує більш ефективний розподіл ресурсів, спрямованих на підтримання супроводу програмного забезпечення;

– виявити реструктуризації – інформація про те, де і коли була проведена реорганізація програмного забезпечення, необхідна для розуміння того, як і до певної міри, чому структура програмного забезпечення була змінена;

– виконати ранню діагностику дефектів, використовуючи інформацію про розвиток дефектів від версії до версії, інженери з супроводу можуть діагностувати дефект на ранній стадії розвитку та коригувати роботи з супроводу, так щоб запобігти потребу в реструктуризації в майбутньому;

– визначити обставини появи і розвитку дефекту − сучасні інструменти автоматизації життєвого циклу програмного забезпечення, наприклад Microsoft Visual Studio Team System, зберігають дані про всі зміни вихідного коду, наприклад хто ці зміни вносив і чому, тому є можливість визначити хто, та реалізуючи які вимоги вніс дефект.

Висновки

Одним із завдань супроводу є не тільки виявлення дефектів проектування програмного забезпечення, але й відстеження їх розвитку.

Для вирішення завдання запропоновано метод діагностики програмного забезпечення, суть якого полягає в моніторингу дефектів проектування.

Для реалізації методу запропоновано модель історії дефектів проектування, набір зображень, що відображають дефекти проектування в різних аспектах.

У майбутній роботі передбачається дослідження зразків зміни дефектів і метрик історії як можливих засобів спрощення інтерпретації користувачем зображень, що генеруються.

Для реалізації запропонованого методу розроблено засіб діагностики програмного забезпечення SEM (Software Evolution Miner) [23], який застосовується в навчальному процесі під час виконання лабораторних робіт із дисципліни «Архітектура та проектування програмного забезпечення».

Література

1. *Lehman M. M.* On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle / M.M. Lehman // *The Journal of Systems and Software*. – 1980. – Vol. 1. – P. 213−221.
2. *Izurieta C.* How Software Designs Decay: A Pilot Study of Pattern Evolution / Clemente Izurieta, James M. Bieman // Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (ESEM'07), September 20−21 2007. – Washington, 2007. – P. 449−451.
3. *Godfrey M.* The past, present, and future of software evolution / M.W. Godfrey, D.M. German // Frontiers of Software Maintenance, 2008. – Beijing, 2008. – P. 129−138.
4. *Нечай О.С.* Методи та засоби виявлення дефектів проектування об'єктно-орієнтованого програмного забезпечення / О.С. Нечай, М.О. Сидоров // Вісник НАУ. – 2009. – №3. – С. 200−205.
5. *Ciupke O.* Automatic detection of design problems in object-oriented reengineering / Oliver Ciupke // Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS'99). – Washington: IEEE Computer Society, 1999. – P. 18–32.
6. *Hovemeyer D.* Finding bugs is easy / David Hovemeyer, William Pugh // ACM SIGPLAN Notices. – 2004. – Vol.39, No.12. – P.92−106.
7. *Marinescu R.* Measurement and Quality in Object-Oriented Design: Ph.D thesis / R. Marinescu. – "Politehnica" University of Timisoara, 2002. – 155 p.
8. *Moha N.* A Domain Analysis to Specify Design Defects and Generate Detection Algorithms / N. Moha, Y. Guéhéneuc, F. Le Meur, L. Duchien // Proceedings of the 11th Intern. Conf. on Fundamental Approaches to Software Engineering. – Springer-Verlag, March-April 2008. – P. 276−291.
9. *Ratiu D.* Using history information to improve design flaws detection / D. Ratiu, S. Ducasse, T. Girba, R. Marinescu // Proceedings of European Conference on Software Maintenance and Reengineering (CSMR'04), March 24−26 2004. – Washington, 2004. – P. 223–232.
10. *Нечай О.С.* Метод побудови моделей дефектів проектування об'єктно-орієнтованого програмного забезпечення / О.С. Нечай, М.О. Сидоров // Наукоємні технології. – 2009. – № 2. – С. 58−64.
11. *Laguë B.* An analysis framework for understanding layered software architectures / Bruno Laguë, Charles Leduc, André Le Bon, Ettore Merlo, Michel Dagenais // Proceedings of the 6th International Workshop on Program Comprehension (IWPC'98), June 24−26 1998. – Washington, 1998. – P. 37−48.
12. *Tichelaar S.* FAMIX and XMI / Sander Tichelaar, Stéphane Ducasse, Serge Demeyer // Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00), November 23−25 2000. – Washington, 2000. – P. 296−306.
13. *Greevy O.* Dynamix − a Meta-Model to Support Feature-Centric Analysis / Orla Greevy // 1st International Workshop on FAMIX (FAMOOSR 2007), June 25 2007. – Zurich, 2007. – P. 25−29.
14. *Girba T.* Modeling History to Understand Software Evolution: Ph.D thesis / T. Girba. – Inauguraldissertation der Philosophisch-naturwissenschaftlichen Fakultat der Universitat Bern, 2005. – 168 p.
15. *Tichelaar S.* Modeling Object-Oriented Software for Reverse Engineering and Refactoring: Ph.D thesis / Sander Tichelaar. – University of Berne, 2001. – 186 p.
16. *Antoniol G.* An automatic approach to identify class evolution discontinuities / Guliano Antoniol, Massimiliano, Di Penta, Ettore Merlo // Proceedings of IEEE International Workshop on Principles of Software Evolution (IWPSE'04), September 06−07 2004. – Washington, 2004. – P. 31−40.
17. *Zou L.* Detecting merging and splitting using origin analysis / Lijie Zou, Michael W. Godfrey // Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03), November 13−17 2003. – Washington, 2003. – P. 146−154.
18. *Stasko J. T.* Software Visualization - Programming as a Multimedia Experience / J. T. Stasko, J. Domingue, M. H. Brown, B. A. Price. – The MIT Press, 1998. – 596 p.
19. *D'Ambros, M.* "A Bug's Life" Visualizing a Bug Database / M. D'Ambros, M. Lanza, M. Pinzger // Proceedings of 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2007), June 24−25 2007. – Banf,2007. – P. 113−120.
20. *Lanza M.* The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques / Michele Lanza // Proceedings of International Workshop on Principles of Software Evolution (IWPSE'01), September 10−11 2001. – New York, 2001. – P. 37−42.
21. *Lungu M.* Reverse Engineering Super-Repositories / Mircea Lungu, Michele Lanza, Tudor Girba, Reinout Heeck // Proceedings of 14th Working Conference on Reverse Engineering (WCRE 2007), October 28−31 2007. – Vancouver, 2007. – P. 120−129.

22. *Wattenberg M.* Baby Names, Visualization, and Social Data Analysis / Martin Wattenberg // Proceedings of IEEE Symposium on Information Visualization (InfoVis 2005), October 23−25 2005. – Minneapolis, 2005. – P. 1−6.

23. А. с. Комп'ютерна програма «Software Evolution Miner» («SEM») / О.С. Нечай (Україна). – № 29953 ; заявл. 19.06.09 ; опубл. 19.08.09.

Стаття надійшла до редакції 25.11.09.

А.С. Нечай

МЕТОД ДИАГНОСТИКИ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Национальный авиационный университет

дефекты проектирования, качество програмного обеспечения, объектно-ориентированное проектирование, правила проектирования

При сопровождении программного обеспечения изменения зачастую вносятся в жестких условиях ограниченных ресурсов. В результате его структура деградирует и, следовательно, становится трудной в понимании. Предлагаемый метод диагностики объектно-ориентированного программного обеспечения позволяет выявлять дефекты проектирования на стадии их возникновения и контроля развития. Метод реализован с помощью предлагаемой метамодели истории дефектов проектирования (DDHM – Design Defect History Model) объектно-ориентированного программного обеспечения и многоаспектной визуализации дефектов проектирования элементов конструкции на различных уровнях абстракции. Анализ графического изображения позволяет выделять наиболее опасные дефекты, более эффективно распределять ресурсы, направленные на сопровождение программного обеспечения, определять реструктуризацию − информацию о том, где и когда переделано программное обеспечение, необходимое для понимания, каким образом и до какой степени, почему была изменена структура программного обеспечения, осуществлять раннее обнаружение дефектов, используя информацию о развитии дефектов от версии к версии, с помощью которой инженеры могут диагностировать дефекты на ранней стадии развития и адаптировать работы по сопровождению для предупредупреждения необходимости реструктуризации в будущем, выяснять обстоятельства возникновения и развития дефектов. Современные инструменты автоматизации жизненного цикла программного обеспечения, такие, как Microsoft Visual Studio Team System, хранят данные о всех изменениях исходного кода, например, кто внес изменения и почему. Для реализации метода разработано средство диагностики программного обеспечения SEM.

Alexander S. Nechay

METHOD FOR OBJECT-ORIENTED SOFTWARE DIAGNOSTIC

National Aviation University

design flaws, design rules, object-oriented design, software quality

During software maintenance changes are introduced often in harsh terms and conditions of limited resources. As a result, its structure degraded and, consequently, it becomes difficult to understand and modify. This phenomenon is known as software decay and is extremely harmful, as tends to be unnoticed at first, but then grow with time. Therefore the problem of developing methods and diagnostics software is an issue.The purpose of the paper is to develop a method of object-oriented software diagnostic to detect design defects in the phase of their appearance and monitoring their development. For achieve a purpose following tasks are supplied and solved: develop a metamodel that enables to perform analysis of design defects history based on it; develop a set of graphics to facilitate the tracking of defects in various aspects; determine which tasks may be solved by studying graphic images.The essence of method of object-oriented software diagnosing is to monitor the design defects. The method is implemented by using the proposed meta-history of design flaws (DDHM - Design Flaws History Model) object-oriented software and multidimensional visualization of defects of the design elements at various levels of abstraction. In DDHM first defect is modeled as a separate entity, able to change their characteristics with time. Considered the following quantitative characteristics of the defect: the degree of deficiency, symptoms of defect intensity and average intensity of the defect symptoms.Analysis of graphic images give next opportunities: highlight the most dangerous defects, a more efficient allocation of resources aimed at supporting the maintenance of software; identify restructuring - information on where and when the restructuring of the software necessary for understanding how and to some extent, why the structure of the software was modified; implement early detection of defects, using information on the development of defects from version to version, with maintenance engineers can diagnose the defect in the early stages of development and adjustment of working with maintenance, so as to prevent the need for restructuring in the future; determine the circumstances of the emergence and development of the defect - the modern tools of automation software life cycle, such as Microsoft Visual Studio Team System, store data about all the changes the source code, for example, who brought these changes and why, so it is possible to determine who, and because of what requirements made defect.To implement the method we developed diagnostics software SEM (Software Evolution Miner).