УДК 004.415.2.045 (076.5)

О.С. Нечай, асп.
М.О. Сидоров, д.т.н., проф.

МЕТОДИ ТА ЗАСОБИ ВИЯВЛЕННЯ ДЕФЕКТІВ ПРОЕКТУВАННЯ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Уведено поняття дефекту проектування програмного забезпечення та ступеню його розвитку. Визначено причини появи та розвитку дефектів проектування та їх класифікацію. Викладено аналітичний огляд і класифікацію існуючих методів і засобів діагностики дефектів проектування та шляхи подальшого їх удосконалення.

Software perfective maintenance needs design flaw identification techniques and tools. In the paper the notion of design flaw and its progress degree are introduced. Reasons of design flaw appearance and progress are defined. Also design flaw classification is developed. Paper presents analysis and classification of existing design flaw diagnostics methods and tools and ways of its further improvements.

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

Постановка проблеми

Для автоматизації процесів поліпшуючого супроводу потрібні методи і засоби ідентифікації дефектів проектування. У традиційних інженерних дисциплінах поняття дефекту давно відомо [1]. Відповідно до ГОСТ 15467-79 дефектом є кожна окрема невідповідність продукції встановленим вимогами. Якщо розглянута одиниця продукції має дефект, то це означає, що хоча б один з її показників якості або параметрів вийшов за граничне значення, або не виконується одна з вимог нормативної документації до продукції. Згідно з ISO 9000:2007 дефектом є невиконання вимоги, пов'язаної з передбаченим або встановленим використанням [2].

Визначення дефекту проектування

В інженерії програмного забезпечення для дефекту не існує єдиного визначення, однак найбільш часто дефект визначають як відхилення від специфікацій або очікувань, що може викликати збій або відмову програмного забезпечення [3]. Тому визначимо дефект як невідповідність робочого продукту (частини процесу виробництва програмного забезпечення) встановленим вимогам.

Дефект проектування – це невідповідність структурних характеристик елемента або фрагмента конструкції програми правилу проектування [4]. Це визначення обмежує об'єкт дослідження від безлічі існуючих дефектів.

Так, невідповідність структурних характеристик дефектів, які можуть виникнути в програмному забезпеченні від помилкової реалізації вимог до дефектів в документації, виділяє лише дефекти, що впливають на структуру програмного забезпечення, і виправляються його реструктуризацією.

Тому не розглядаються дефекти, пов'язані з порушенням стандартів кодування, процесами розробки, продуктивністю алгоритмів або помилками в документації і т.д.

Сьогодні об'єктно-орієнтоване програмування є основною парадигмою розробки, тому як елементи конструкції розглядаються методи, класи і підсистеми об'єктно-орієнтованого програмного забезпечення. Оскільки дефекти проектування можуть охоплювати відразу кілька елементів конструкції, то під фрагментом конструкції мається на увазі кластер взаємодіючих елементів.

Невідповідність правилу проектування виділяє нефункціональні дефекти, тому не розглядаються дефекти проектування, пов'язані з помилками в інтерфейсах, синхронізацією, обробкою помилок і т.д. У роботі [4] пропонується класифікувати дефекти на функціональні і нефункціональні за їх можливістю викликати збій.

Визначимо ступінь розвитку дефекту проектування як кількісну характеристику відхилення структурних характеристик елемента або фрагмента конструкції програми від правила проектування.

В онтології знань об'єктно-орієнтованого проектування під правилами проектування розуміють принципи, евристики, кращі практики (best practices) та зразки проектування (design patterns) [5].

Програмне забезпечення, сконструйоване за проектом, що ігнорує правила проектування, нездатне до масштабування, супроводу та повторного використання [6]. Тому вартість супроводу значно підвищується, а без можливості регулярної реструктуризації супровід та експлуатація програмного забезпечення економічно невигідні.

Поняття «дефект проектування» охоплює проблеми різного рівня деталізації – від низькорівневих проблем, таких, як поганий аромат (bad smell), до архітектурних проблем, таких як, антизразки проектування (anti-patterns).

Дефект Blob [7] (God Class [5]) є типовим прикладом дефекту проектування. Клас, що має цей дефект, великий клас-контролер, який монополізує значну частину обчислень програми. У ньому оголошено багато полів-даних і методів, зв'язаність між якими слабка. Клас-контролер залежить від даних, які зберігаються в простих класах-даних. Класи-дані містять лише дані та методи доступу до них.

Дефекти проектування можуть бути внесені на фазу реалізації у результаті помилок, допущених на фазі проектування. Виникнення дефекту на цій фазі можливо, але, як правило, розробники проводять проектування дуже ретельно, тому що помилки на фазі проектування дорого коштують. Імовірність виникнення дефекту підвищується під час використання гнучких (agile) методів розробки, коли на перший план виходять конструювання, тестування і переробка, а початковому проектуванню приділяється мало уваги.

Ризик виникнення дефекту проектування різко підвищується на фазі супроводу. У процесі супроводу відбувається внесення змін, що призводить до появи нових класів, зв'язків між класами та змін вже існуючих. У підсумку конструкція системи може суттєво змінитися порівняно зі своїм початковим станом, часто не в кращу сторону. Структура стає заплутаною, насичується зайвими зв'язками і, як наслідок, стає складною в розумінні і модифікації.

Класифікація дефектів проектування

Дефекти проектування пропонується класифікувати за впливом на програмне забезпечення, типом ураженого елемента конструкції і за вимірюваністю (рис. 1).

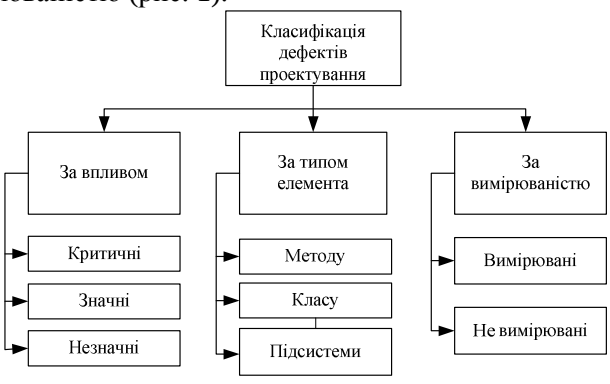

Рис.1. Класифікація дефектів проектування

За впливом на програмне забезпечення пропонується класифікувати дефекти проектування відповідно до ГОСТ 15467-79:

– критичні дефекти роблять супровід програмного забезпечення практично неможливим;

– значні дефекти справляють істотний вплив на можливість або довговічність супроводу програмного забезпечення;

– незначні дефекти практично не впливають на показники якості програмного забезпечення.

Визначення належності дефекту однієї з зазначених категорій виконується інженером залежно від ступеня розвитку дефекту або від зусиль, необхідних для супроводу ділянки конструкції програмного забезпечення, що містить дефект.

За типом ураженого елемента конструкції дефекти проектування класифікуємо так:

– дефект методу – методи реалізують абстракцію управління, тому до них застосовують правила структурного програмування, але оскільки в роботі розглядаються дефекти об'єктно-орієнтованого проектування, до цього типу належать дефекти, пов'язані або з неправильним розподілом функціональності за методами класу, або неправильним розміщенням методу;

– дефект класу – більшість відомих дефектів проектування належать до класу або кластеру класів;

– дефект підсистеми – клас має занадто детальний рівень деталізації, що б бути корисним елементом організації великих програмних систем, тому вони будуються з підсистем, які теж можуть бути уражені дефектами проектування.

За вимірюваністю дефекти проектування пропонується класифікувати так:

– вимірювані дефекти, які в процесі супроводу програмного забезпечення крім появи та зникнення можуть змінювати ступінь свого розвитку;

– невимірювані дефекти, які в процесі супроводу програмного забезпечення можуть тільки з'являється і зникати в результаті реструктуризації.

Методи виявлення дефектів проектування

Аналіз робіт із діагностики дефектів проектування і реструктуризації виявив основні категорії методів виявлення за рівнем автоматизації:

− неавтоматичні;

− напівавтоматичні;

− автоматичні.

Приклад неавтоматичного методу наведено в роботі [8]. Описаний процес, заснований на інспекції робочих продуктів програмного забезпечення вручну з метою виявлення дефектів проектування. Неавтоматичні методи не можуть бути легко масштабовані на системи великих розмірів. Як засоби ручного виявлення дефектів використовують звичайні інструменти для перегляду коду та графічних зображень моделей програмного забезпечення.

Напівавтоматичні методи, як правило, є спеціалізованими методами зворотної інженерії [9], які забезпечують інженера спеціальними моделями програмного забезпечення, вивчаючи, які він може виявити аномалії і дефекти проектування в їх конструкції. Однак складність і розміри програмного забезпечення перетворюють вивчення моделей, поданих як правило UML діаграмами, в рутинну та ненадійну роботу.

У роботах [6; 10] пропонується вирішення цієї проблеми шляхом візуалізації програмного забезпечення відображаючи властивості його елементів на властивості елементів зображення. Цей підхід називається поліметричними представленнями (Polymetric Views). Наприклад, на рис. 2 показано програму, на якій кожен прямокутник зображує клас.

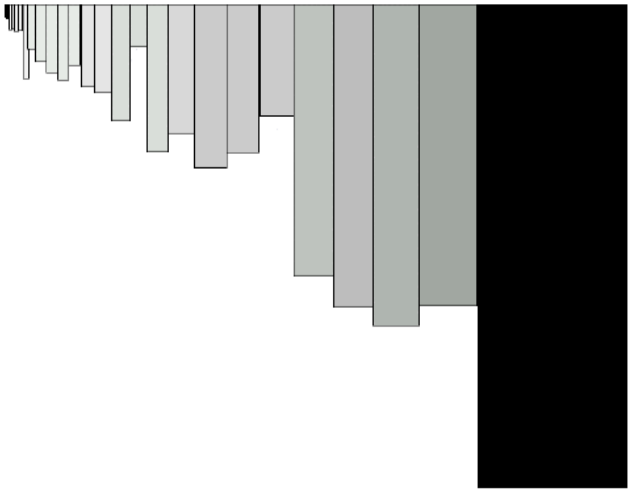

Рис. 2. Візуалізація програмного забезпечення для виявлення дефектів проектування

Властивості кожного класу оцінені за допомогою метрик, таких, як кількість методів у класі, кількість рядків коду в класі, кількість атрибутів класу відображається на такі характеристики прямокутника, як висота, ширина, колір, розміщення. На такому зображенні візуально можна виявити класи з аномальними параметрами та розпочати роботу щодо їх реструктуризації.

Розглянутий підхід підвищує ефективність роботи інженера, але має істотні недоліки. По-перше, відсутні засоби фільтрації і не дано рекомендації щодо критерію відбору, що змушує вивчати всі зображення у пошуках підозрілих прямокутників і робити суб'єктивні висновки щодо того чи іншого елемента програмного забезпечення. По-друге, на зображенні може бути показана лише обмежена кількість параметрів класу, що робить неможливим виявлення дефектів, що мають чотири і більше симптомів.

Метод реалізований в інструменті CodeCrawler як частина проекту MOOSE [11]. CodeCrawler – незалежний від мови програмування інструмент для зворотного програмного забезпечення, який комбінує метрики та візуалізацію. Інструмент написаний на мові SmallTalk і доступний на будь-якій популярній платформі.

У роботі [12] описано напівавтоматичний метод, суть якого полягає у використанні мови специфічного домену (у даному випадку домен – аналіз якості проектування) для опису дефектів проектування у вигляді формальної специфікації, в якій указуються симптоми дефектів, метрики для оцінки цих симптомів, а також порогові значення цих метрик (рис. 3).

```
CODESMELL define LongMethod as METRIC LOC
METHOD with VERY HIGH and 10.0;
CODESMELL define NoParameter as METRIC
NMNOPARAM with VERY HIGH and 5.0;
CODESMELL define NoInheritance as METRIC DIT
with 1 and 0.0;
CODESMELL define NoPolymorphism as STRUC NO
POLYMORPHISM;
CODESMELL define ProceduralName as LEXIC CLASS
NAME with (Make, Create, Exec);
CODESMELL define UseGlobalVariable as STRUC
USE GLOBAL VARIABLE;
CODESMELL define ClassOneMethod as METRIC
NMD with VERY LOW and 10.0;
```

Рис. 3. Специфікація дефектів проектування

Код специфікації далі інтерпретується і на його основі генерується алгоритм пошуку дефектів на мові програмування Java. Результатом роботи алгоритму є список виявлених дефектів на основі необмеженої кількості симптомів. Основним недоліком є проблема всіх методів, що застосовують метрики – вибір порога. Поріг розбиває область значень метрики на дві підобласті. Залежно від підобласті, в яку потрапляє значення метрики, робиться висновок про стан вимірюваної сутності. Наприклад, якщо вимірюється здатність до повторного використання елемента конструкції, можливі значення вимірювання

знаходяться в інтервалі [0,1] і поріг визначено 0,7, то елементи, що мають значення параметра вище цього порога, вважаються добре пристосованими до повторного використання. Однак виникають питання:

− чому обрано поріг саме 0,7;

− чому не 0,5;

− чи дійсно елемент конструкції програми, що має значення здатності до повторного використання 0,68 не пристосований до повторного використання порівняно з елементом, що має значення цього параметра 0,7;

− чи буде мати сенс такий поріг, якщо аналізована генеральна сукупність має максимальне значення здатності до повторного використання 0,5.

Ще одним недоліком методу є використання жорстко закодованих метрик. Описуючи в специфікації дефект проектування, інженер може використовувати лише передбачені метрики і не може поставити нову метрику або змінити спосіб обчислення вже існуючої. Даний метод реалізований в інструменті DÉCOR [13].

До напівавтоматичних методів можна віднести метод, запропонований у роботі [14], який полягає у використанні нечіткої кластеризації для виявлення елементів конструкції програми, що найбільшою мірою порушують правила об'єктно-орієнтованого проектування. Хоча метод і вирішує проблему з вибором порога, він не надає механізму обробки отриманих кластерів, тому не ясно, які кластери містять дефектні елементи, а які ні.

Автоматичний метод використання евристик об'єктно-орієнтованого проектування для завдання метричних правил виявлення дефектів проектування запропонований у роботі [15].

Метричні правила називаються стратегіями виявлення і мають графічне зображення (рис. 4).

Кожен вхід являє собою логічний вираз порівняння метрики та її порогового значення. Результати обчислення виразів потім стають операндами логічних операторів І (AND) та АБО (OR).

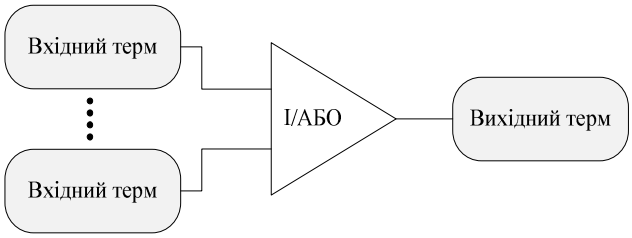

Рис. 4. Стратегії виявлення дефектів проектування у загальному вигляді

У результаті отримане логічне значення виходу визначає наявність (значення TRUE) або відсутність дефекту (значення FALSE) елемента конструкції, що аналізується. Метод не надає достатньо деталей для реалізації пошуку дефектів в інших інструментах та для завдання нових стратегій виявлення. Крім того, залишається проблема визначення порогів значень метрик та адекватність їх завдання. Метод реалізований в інструменті під назвою iPlasma [16] – інтегроване середовище для аналізу якості об'єктно-орієнтованого програмного забезпечення, яке включає всі необхідні фази аналізу: від витягу моделей (для C + + і Java) до високорівневого метричного аналізу та виявлення дублікатів коду.

У роботі [17] запропоновано автоматичний метод використання логічного мета-програмування, за допомогою якого виконується маніпуляція інформацією про структуру програмного забезпечення. Вихідний код аналізується, і збирається інформація високорівневої організації програми. З метою інтерпретації інформації, зібраної з вихідного коду, розроблена мета − модель об'єктно-орієнтованого програмного забезпечення. Мета-модель описує елементи і взаємовідносини між ними, які можуть бути в конструкції програми. Модель програми, що відповідає мета-моделі, подано у вигляді предиката, що дозволяє опитувати і маніпулювати моделлю з допомогою мови запитів. Потім за допомогою мови логічного програмування Prolog створюють опис фрагментів конструкції, що відповідають дефекту проектування (рис. 5).

```
% Base classes should not have knowledge about their descendants
knowsOfDerived (Class, DerivedClass) :-
   % Both Class and DerivedClass must be classes
   class (Class), class (DerivedClass),
   % DerivedClass is a direct or transitive descendant of Class
   trans (inheritsFrom, DerivedClass, Class),
   % The base class knows its heir
   knows (Class, DerivedClass).

% A class 'knows' another class, if
knows (Class1, Class2) :-
   % it inherits from that class, or
   class (Class1), class (Class2),
     inheritsFrom (Class1, Class2);
   % it has an attribute of that type, or
   hasAttribute (Class1, Attr), hasType(Attr, Class2);
   % it has a method which returns an object of that type, or
   hasMethod (Class1, Meth1), returns (Meth1, Class2);
   % it has a method which calls a method of that class, or
   hasMethod (Class1, Meth1), calls (Meth1, Meth2),
     hasMethod (Class2, Meth2);
   % it has a method containing a parameter with type of that class.
   hasType(Param, Class2).
```

Рис. 5. Формалізація правила проектування в Prolog

Похідний клас повинен мати знання про базовий клас за визначенням, але базовий клас не повинен нічого знати про свої похідні класи. Розглянутий метод добре працює з невимірюваними дефектами, але не підходить для виявлення та аналізу вимірюваних дефектів. Метод реалізований в інструменті GOOSE, який був розроблений у Forschungszentrum Informatik на платформі проекту FAMOOS для вирішення низки завдань реінженерії програмного забезпечення. Інструмент підтримує візуалізацію програмного забезпечення, підрахунок метрик, створення запитів до конструкції та автоматичну реструктуризацію. Загальний недолік розглянутих методів полягає у тому, що розглядається лише одна версія програми, а отже, ігнорується історична інформація про появу і розвиток дефектів проектування. Ця інформація дуже важлива, оскільки дефекти проектування мають природу, подібну хворобам людини. Цю аналогію вперше використав Парнас [18]. Деякі хвороби присутні під час народження людини, інші ж набуваються.

Лікарі, використовуючи історичну інформацію про хвороби, можуть більш точно встановити діагноз. Більше того існують хвороби, до яких організм звикає, і вони не становлять небезпеки для здоров'я людини. Аналогічно інформація про розвиток дефектів проектування може допомогти визначити дефекти, які становлять небезпеку для подальшого розвитку програмного забезпечення і виключити з розгляду дефекти, які не являють собою такої небезпеки. Наприклад, неякісна структура і заплутаний код автоматично згенерованого коду, що не вимагає змін, ніяк не вплине на подальші витрати по супроводу програмного забезпечення. У той же час клас, розмір і складність якого зростає від версії до версії, має негативний вплив на супровід.

Існує кілька методів, що використовують історичну інформацію. У роботі [19] наведено метод виявлення дефектів проектування на основі логічної зв’язаності модулів. Пошук дефектів виконують у два етапи.

1. Аналіз послідовності змін – у кожному релізі програмного забезпечення, кожному зміненому модулю призначають номер відповідно до послідовності усіх змін даного релізу. Потім модулі, які змінюються в одній і тій самій послідовності у різних релізах, позначаються як логічно зв'язані.

2. Аналіз звітів про зміни проводиться для уточнення логічної зв’язаності шляхом з'ясування, чи викликані зміни логічно зв'язаних модулів однією і тією ж причиною. Модулі з різних підсистем, які виявилися логічно зв’язаними, свідчать про помилки, допущені під час проектування цих підсистем.

Розширенням методу з роботи [15] є робота [20]. Сутність методу полягає в розширенні стратегій виявлення дефектів проектування логічними виразами, операнди яких є історичні метрики. Перша метрика *Stab* − стабільність елемента E щодо вимірювання M визначається за формулою

$$\mathrm{Stab_i(E,M)} = \begin{cases} 1, \text{if } \mathrm{M(E_i)} - \mathrm{M(E_{i-1})} = 0, \\ 0, \text{if } \mathrm{M(E_i)} - \mathrm{M(E_{i-1})} \neq 0 \end{cases} (i > 1);$$

$$Stab_{i..n}(E,M) = \frac{\sum_{i=2}^{n} Stab_i(E,M)}{n-1} \quad (n > 2),$$

де

i – номер версії елемента програмного забезпечення.

Результат вимірювання потрапляє в інтервал [0,1], де значення 0 означає, що елемент змінювався у всіх версіях, а 1 − елемент не змінювався в жодній із версій.

Інша метрика *Pers* – тривалість існування дефекту проектування D у елемента E щодо тривалості існування самого елемента визначається за формулою

$$Pers_i(E,D) = \begin{cases} 1, \text{ if } E_i \text{ містить дефект } D, \\ 0, \text{ if } E_i \text{ не містить дефект } D \end{cases} \qquad (i \geq 1)$$

$$Pers_{i..n}(E,D) = \frac{\sum_{i=1}^{n} Pers_i(E,D)}{n} \quad (n > 2),$$

де

i – номер версії елемента програмного забезпечення.

Результат вимірювання потрапляє в інтервал [0,1], де 0 означає, що елемент всю свою історію до самої останньої версії не має дефектів проектування, а 1 − елемент з самого початку своєї історії мав дефект проектування. Відповідно до цього методу стабільні елементи (*Stab* > 95 %) та ті, що мають дефект з великою тривалістю існування (*Pers* > 95 %) не розглядаються, оскільки не мають суттєвого впливу на розвиток програмного забезпечення. До недоліків методу можна віднести невизначеність метрики, яку необхідно використовувати для знаходження стабільності елемента.

Так само аналіз сконцентрований на еволюції елемента конструкції, а не дефекту проектування, тому в результаті виявлення дефекту відбувається на основі інформації про зміну елемента конструкції, яка могла не тільки не збільшити ступінь розвитку дефекту, але й навіть зменшити її. Крім того, визначення тривалості має все ту ж слабкість, пов'язану з визначенням порогів.

Висновки

У результаті аналізу існуючих методів і засобів діагностики дефектів проектування було визначено їх класифікацію і для кожної категорії класифікації було розглянуто відповідні методи, їх можливості та недоліки. Методи, засновані на аналізі однієї версії програми, досягли зрілості та доповнюють один одного. Однак методи, засновані на еволюційному аналізі, знаходяться на стадії розвитку, і все ще не вирішено ряд проблем, пов'язаних з розвитком дефектів проектування: як інформація про розвиток дефектів може бути використана для розробки превентивних заходів, оптимізації реструктуризації і розуміння розвитку програмного забезпечення.

Література

1. *ГОСТ 15467.* Управление качеством продукции. Термины и определения. – М.: Изд-во стандартов, 1979. – 38 с.
2. *ISO 9000:2000.* Quality management systems -- Fundamentals and vocabulary. – ISO, 2000. – 41 p.
3. *Runeson P.* What Do We Know about Defect Detection Methods? / Per Runeson, Carina Andersson, Thomas Thelin, Anneliese Andrews, Tomas Berling // IEEE Software. – 2006. – Vol.23, No.3. – P. 82−90.
4. *Marinescu R.* Measurement and Quality in Object-Oriented Design: Ph.D thesis / R. Marinescu. – ”Politehnica” University of Timisoara, 2002. – 155p.
5. *Garzas J.* Object-oriented design knowledge: principles, heuristics, and best practices / J. Garzas, M. Piattini. – Hershey: Idea Group Publishing, 2007. – 376 c.
6. *Lanza M.* Object-Oriented Metrics in Practice / M. Lanza, R. Marinescu. – Springer-Verlag Berlin Heidelberg, 2006. – 205 p.
7. *Brown W.J.* Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis / W.J. Brown, R.C. Malveau, H.W. McCormick, T.J. Mowbray. – Wiley. – 1998. – 336 p.
8. *Riel A.J.* Object-Oriented Design Heuristics / Arthur J. Riel. – Addison Wesley. – 1996. – 400 p.
9. *Travassos G.* Detecting defects in object-oriented designs: using reading techniques to increase software quality / G. Travassos, F. Shull, M. Fredericks, V. R. Basili // In Proc. of the 14th OOSPLA Conf., 1999. – P. 47–56.
10. *Lanza M.* Object-Oriented Reverse Engineering – Coarse-grained, Fine-grained, and Evolutionary Software Visualization: Ph.D thesis / Michele Lanza. – University of Berne, 2003. – 132 p.
11. *Ducasse S.* Moose: a Collaborative and Extensible Reengineering Environment / Stéphane Ducasse, Tudor Gîrba, Michele Lanza, Serge Demeyer // RCOST Software Technology Series. – Franco Angeli, Milano, 2005. – P. 55–71.
12. *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.
13. *Moha N.* Ptidej and DECOR: Identification of Design Patterns and Design Defects / N. Moha, Y. Guéhéneuc // Tool demo at the International Conference on Automated Software Engineering, November 2007.
14. *Serban C.* Software Quality Assessment Using a Fuzzy Clustering Approach / Camelia Serban, Horia F. Pop // Studia Universitatis Babes-Bolyai Series Informatica. – Babes-Bolyai University, 2008. – Vol. LIII. – P. 27−38.
15. *Marinescu R.* Detection strategies: Metrics-based rules for detecting design flaws // Proc. of Intern. Conf. on Software Maintenance (ICSM’04). – IEEE Computer Society Press, 2004. – P. 350–359.
16. *Marinescu C.* iPlasma: An integrated platform for quality assessment of object-oriented design / C. Marinescu, R. Marinescu, P. Mihancea, D. Ratiu, R. Wettel // Proc. of 21st Intern. Conf. on Software Maintenance. – Tools Section. – 2005.
17. *Ciupke O.* Automatic detection of design problems in object-oriented reengineering / Oliver Ciupke // Proc. of TOOLS’30, 1999. – P. 18–32.
18. *Parnas D. L.* Software Aging / David Lorge Parnas // Proc. of Intern. Conf. on Software Engineering (ICSE’94). – IEEE Computer Society / ACM Press, 1994. – P. 279−287.
19. *Gall H.* Detection of Logical Coupling Based on Product Release History / H. Hall, K. Hajek, M. Jazayeri // Proc. of the Intern. Conf. on Software Maintenance (ICSM ’98). – IEEE Computer Society Press, 1998. – P. 190–198.
20. *Ratiu D.* Using history information to improve design flaws detection / D. Ratiu, S. Ducasse, T. Gˆrba, R. Marinescu // Proc. of European Conf. on Software Maintenance and Reengineering (CSMR’04). – P. 223–232.

Стаття надійшла до редакції 07.09.09.