DOI 10.31891/2307-5732-2022-315-6-77-81
УДК 004.9

ДЗЮРБАН Едуард
Хмельницький національний університет
e-mail: eduard.dziurban@gmail.com
ЯШИНА Оксана
Хмельницький національний університет
https://orcid.org/0000-0001-7816-1662
e-mail:oksana.yashyna@ukr.net

МЕТОД ОЦІНКИ ОБ'ЄКТНО-ОРІЄНТОВАНИХ ПРОГРАМНИХ СИСТЕМ НА ОСНОВІ АНАЛІЗУ ЗМІНИ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

Добре відомий факт, що технічне обслуговування програмного забезпечення відіграє важливу роль і набуває важливого значення в життєвому циклі програмного забезпечення. Оскільки об'єктно-орієнтоване програмування давно вже стало стандартом, дуже важливо розуміти проблеми підтримки об'єктно-орієнтованих програмних систем та спосіб виявлення їх потенційних місць виникнення. Ця стаття спрямована на оцінку об'єктно-орієнтованих систем за допомогою аналізу зміни вимог до програмної системи. Основні проблеми порушені в статті: покращення алгоритму аналізу впливу зміни не функціональних вимог до програмної системи на функціональні та їх наслідування.

Попит на ефективне програмне забезпечення зростає з кожним днем, і впровадження об'єктно-орієнтованого проектування програмних систем здатне задовольнити цей попит, оскільки це, мабуть, найпотужніший механізм розробки ефективних програмних систем. Це може допомогти не тільки зменшити витрати, але й розробити високоякісне системне програмне забезпечення. Розробникам програмного забезпечення потрібні відповідні показники для розробки ефективної системи програмного забезпечення. Ця стаття спрямована на дослідження методів оцінки об'єктно-орієнтованої програмної системи за допомогою аналізу впливу змін функціональних вимог до програмного забезпечення за допомогою не функціональних вимог.

Незважаючи на те, що об'єктно-орієнтований підхід має багато переваг, а також він є найпоширенішим зараз та буде таким у майбутньому, його практичність буде доведена лише тоді, коли аспекти управління процесом розробки програмного забезпечення за допомогою цієї методології буде ретельно розглянуто. Саме тут показники програмного забезпечення відіграють важливу роль, забезпечуючи краще планування, зменшення ризиків, раннє виявлення потенційних проблем, оцінку якості та ефективності. У цій статті пропонується набір показників, які найкраще підходять для оцінки використання основних концепцій об'єктно-орієнтованої парадигми, таких як наслідування, інкапсуляція, поліморфізм та повторне використання коду, які однозначно відповідають за підвищення якості програмного забезпечення та продуктивності розробки.

Ключові слова: об'єктно-орієнтована архітектура, аналіз, оцінка, програмна система, зміна алгоритму аналізу впливу, наслідування функціональності.

DZIURBAN Eduard, YASHYNA Oksana
Khmelnytskyi National University

METHOD OF EVALUATION OF OBJECT-ORIENTED SOFTWARE SYSTEMS BASED ON THE ANALYSIS OF CHANGES IN THE SOFTWARE SYSTEM REQUIREMENTS

It is a well-known fact that software maintenance plays an important role and becomes important in the software life cycle. Since object-oriented programming has long become the standard, it is very important to understand the problems of maintaining object-oriented software systems, and how to avoid them by identifying potential gaps in the software system as early as the design analysis. This article is aimed at evaluating object-oriented systems using the analysis of changes in the requirements for the software system. The main problems raised in the article are the change of the algorithm for analyzing the impact of changing non-functional requirements on functional ones and their inheritance.

The demand for efficient software is increasing day by day, and the adoption of object-oriented design of software systems is able to satisfy this demand, as it is perhaps the most powerful mechanism for developing efficient software systems. This can not only help in reducing the cost but also helps in developing high quality system software. Software developers need appropriate metrics to develop an effective software system. This practice is aimed at researching methods for evaluating an object-oriented software system using software impact analysis based on tracking requirements to changes in functional requirements using non- functional requirements.

Although there are many advantages to the object-oriented approach, and the fact that this approach is the most widespread now and will be in the future, it will be truly recognized, proven and practical only when the management aspects of the software development process using of this methodology will be carefully considered. This is where software metrics play an important role, enabling better planning, evaluating improvements, reducing unpredictability, early detection of potential problems, and evaluating performance. This paper proposes a set of metrics best suited to evaluate the use of core concepts of the object-oriented paradigm, such as inheritance, encapsulation, polymorphism, and a strong emphasis on code reuse, which are uniquely responsible for increasing software quality and development productivity.

Keywords: object-oriented architecture, analysis, evaluation, software system, change of impact analysis algorithm, imitation of functionality.

Постановка проблеми

Існує декілька стандартів відстеження змін до вимог програмної системи. Це, зокрема ISO15504 та CMMI. Також за останні десятиліття було розроблено декілька методик для забезпечення вимог до власне відстеження цих змін. Більшість традиційних методів схожі на два найбільш відомих, це Trace-based Impact Analysis Methodology (TIAM), та Work Product Model (WoRM). Згадані методології дають передбачуване

значення для знаходження класів зі схожими змінами. Методологія TIAM призначена більше для планування, ніж для прямого впровадження змін. TIAM потенційно може бути використана для оцінки ризиків щодо нестабільності вимог. У випадку проектних змін існують певні наслідки застосування об'єктно-орієнтованого підходу. На практиці було виявлено, що архітектори-початківці мають проблеми зі створенням класів та кругообігом між декларативними та процедурними аспектами рішення. Відповідно, має місце впровадження таких патернів або методів відстеження змін до вимог як «вимога-компонент», які можуть бути застосовані як до традиційних, так і до найбільш сучасних процесів розробки програмного забезпечення. Такий підхід дозволив досягти відповідності структури вихідного коду патернам та методам, які забезпечують легкість відслідковування змін у системи відповідно до нових вимог. У життєвому циклі програмне забезпечення зазнає змін на всіх етапах. Програмний продукт є успішним, якщо зміни в програмному забезпеченні ідентифікуються або здійснюється їх управління протягом усіх фаз життєвого циклу програмного забезпечення.

Для отримання програмного продукту повинна бути чітко встановлена межа, і вона має ставати більш точнішою задля того щоб програмний продукт пройшов всі етапи життєвого циклу. Обслуговування програмного забезпечення споживає приблизно сорок відсотків витрат на програмне забезпечення, оскільки є нетривіальною фазою в життєвому циклі розробки програмного забезпечення. А оскільки це нетривіальна фаза, то відстеження зв'язку між кодом та елементами в програмному забезпеченні може полегшити виконання багатьох завдань. Розуміння програми, супровід, вимоги трасування, аналіз впливу та повторне використання існуючого програмного забезпечення, це все важливі елементи про які не слід забувати.

Аналіз останніх джерел

У житті програмного продукту є кілька етапів. Модель водоспаду, як описано у [1], має п’ять основних фаз. Це аналіз вимог і специфікації, кодування та тестування модулів, інтеграційне тестування, системне тестування та обслуговування. Це дослідження стосується лише аспекту заключної фази, обслуговування. Етап обслуговування є найдовшою фазою життєвого циклу. Обслуговувати програмне забезпечення з часом стає все складніше, оскільки система прогресує і розвивається. У дослідженні [2] показали алгоритми для обчислити транзитивного замикання кожного з потенційно зачеплених класів і методів. За допомогою них можна значно покращити інформацію, яку надають алгоритми розпізнавання шаблони проектування, ефекти змін типу даних, а також ефекти додавання та видалення класів. У роботі [3] представлено інтегроване середовище для обслуговування програм C++, яке описує три нові графи залежностей, що характерні для об'єктно-орієнтованих програмних систем: повідомлення, клас і оголошення залежності у моделі під назвою C++ DG. Описані залежності, зокрема щодо ефекту пульсації та регресійного тестування. Застосування виявлених залежностей і розподіл програми призводить до рекурсивного аналізу ефекту хвиль, спричинених модифікацією коду. У міру розташування цих ефектів їх місця можна «позначити» для тестування або повторного виконання на етапі тестування.

У дослідженні [4] пояснили чотири алгоритми, які вимірюють вплив запропонованих змін на об’єктно-орієнтовані системи. Ефект пульсації розраховується шляхом застосування таких алгоритмів як:

1. Обчислення впливу змін всередині класу.
2. Розрахунок впливу змін серед клієнтів.
3. Розрахунок впливу серед підкласів.
4. Вимірювання загального ефекту, керуючи алгоритмами 1, 2 та 3.

Автор також представив деталі того, як різні типи змін впливають на систему. Зміни широко класифікуються, а потім уточнюються до більш детальної інформації, наприклад додавання учасника або зміна атрибута. Відповідно до роботи [5] візуальний аналіз впливу покращив розпізнавання додаткових залежностей. У дослідженні [6] автор вказав, що практика розробки програмного забезпечення розвивається відповідно до вимог розподілених програми на різнорідних платформах; зміна програмного забезпечення все більше впливає на проміжне програмне забезпеченням і компоненти. Відносини залежності взаємодії тепер вказують на більше відповідні наслідки зміни програмного забезпечення та обов’язково приводять до обов’язкового їх аналізу. Сучасні моделі аналізу впливу змін програмного забезпечення не враховують ці тенденції належним чином.

У статті [7] автор пояснив підхід, який застосовується до написання програмного забезпечення на об’єктно-орієнтованій мові для відстеження об’єктно-орієнтованого коду та реалізації функціональних вимог. Тут розглядається проблема встановлення зв’язків та їх аналізу між вільною текстовою документацією, пов’язаною з циклом розробки та обслуговування програмної системи, а також її кодом.

Моделі відстеження змін до вимог

Відстеження вимоги означає здатність описувати та стежити за життям вимоги як у прямому, так і у зворотному напрямку. Це здатність відстежувати вимоги до компонентів проекту або їх впровадження. Зворотне відстеження – це можливість відстежити вимогу до її джерела, тобто до особи, установи або ж закону чи аргументу тощо. Відслідковування вимог стосується також зв’язків між ними, або ж їх зв’язку з атрибутами програмної системи [8].

Аналіз впливу на основі відстеження змін до вимог

Аналіз впливу на основі відстеження змін вимог до програмної системи – це нефункціональне та неформальне відстеження. Функціональне відстежування пов'язане з добре встановленим відображенням між типами моделі об'єктів і типами відображення. Це дозволяє аналізувати моделі проектування, моделі процесів, організаційні моделі. Нефункціональне відстеження пов'язане з відстеженням нефункціональних аспектів розробки програмного забезпечення. Зазвичай, вони пов'язані з аспектами якості та є результатом зв'язку з нематеріальними концепціями. Нефункціональне відстеження класифікується за чотирма категоріями, такими як: причина, контекст, рішення та технічний аспект.

У цій статті оглянута методологія аналізу впливу на основі відстеження змін вимог до програмної системи, яка призначена для розширення можливостей відстеження потенційних проблем а також проведення об'єктно-орієнтованого аналізу та аналізу деяких аспектів архітектури. Запропоновано розглянути кілька етапів, а саме:

Фаза перша включає в себе такі пункти:

A. Перевірка нових вимог від будь-якої із зацікавлених сторін.
B. Класифікація вимог, функціональних чи нефункціональних.
C. Матриця відстеження може допомогти відстежити вимогу.
D. Огляд вимог.
E. Оцінка вимог.
F. Вимоги до документації.
G. Приймальні випробування.

Після проходження першої фази, відповідно до визначених показників буде розділено аналіз по них у другій фазі. Це такі показники:

A. Стабільність: нестабільні вимоги.
B. Повнота: неповні вимоги.
C. Чіткість: незрозумілі вимоги.
D. Дійсність: Недійсні вимоги.
E. Здійсненність: нездійсненні вимоги.
F. Прецедент: безпрецедентні вимоги.

Стабільність

Цей показник вказує на вразливість системи до змін. Було помічено, що підтримка програмного забезпечення погіршується, оскільки до нього вносяться зміни, що збільшує складність програмного забезпечення. В такому випадку стабільність системи розраховується за формулою:

$$S(\#NORS + \#NOCNR + \#NOCUR + \#NOCDR) / (\#NORS),$$

де S – стійкість, NORS – кількість вхідних вимог систему, NOCNR – кількість сукупної кількості вимог, NOCUR – сукупна кількість запитів, оновлених у системі, NOCDR – сукупна кількість запитів, видалених із системи.

Повнота

Цей показник визначає повноту вимоги та обчислюється за формулою:

$$CMP = NARS - NIR$$

CMP – ступінь завершеності системи, NARS – кількість фактичних/початкових вимог до системи, NIR – кількість незавершених вимог у системі.

Ясність

Цей показник відображає чіткість вимог до системи

$$CL = NARS - NIR - UCLR$$

CL – ступінь чіткості системи, NARS – кількість фактичних/початкових вимог до системи, NIR – кількість незавершених вимог у системі, UCLR – кількість незрозумілих вимог.

Здійсненність

Цей показник відображає ступінь здійсненності системи, тобто фактичне число її реалізованості.

$$FR = IFR - UCLR$$

FR – показник ступеня всіх техніко-економічних вимог системи, IFR – кількість нездійсненних вимог у системі, UCLR – кількість незрозумілих вимог.

Прецедент

Це показник, який відображає наскільки система завершена на даний момент.

$$PR = CMP + CL + FR$$

PR – показних попередніх вимог до системи, CMP – завершеність системи, CL – чіткість системи, FR – здійсненність системи.

Результати

У цьому розділі буде проведено ідентифікацію, візуалізація та аналіз відстеження змін до вимог на об'єктно-орієнтованій програмній системі. Тут як обов'язкову вимогу було прийнято практичне дослідження системи бронювання авіаквитків. Виходячи з рівня вимоги, необхідно розділити вимоги на певні рівні, після чого робити подальший аналіз. В результаті було виділено такі рівні:

1. Потреби замовника.
2. Функціонал.
3. Варіанти використання.
4. Додаткові вимоги.
5. Варіанти тестування.

Вимоги на верхньому рівні (потреби замовника) збираються за допомогою різних методів виявлення вимог, такі як опитування, дзвінки, зустрічі тощо.

Бізнес-аналітик виводить другий рівень (функціонал) із запитів зацікавлених сторін, корегуючи вимоги переводить їх із проблемної області в область вирішення. Функціонал повинен мати всі атрибути адекватних вимог до програмної системи.

Третій рівень містить варіанти використання програмної системи, які виділяються на першому рівні аналізу системи.

Додаткові вимоги охоплюють переважно нефункціональні вимоги. Вони також можуть фіксувати деякі загальні функціональні вимоги, не пов'язані з жодними конкретними випадками використання.

Варіанти тестування створені для перевірки вимог третього рівня.

Як приклад, можна привести алгоритми для системи бронювання авіаквитків.

Крок 1: Запуск алгоритму.
Крок 2: Введення URL-адреси.
Крок 3: Введення даних про рейси для пошуку рейсів.
Крок 4. Вибір рейсу.
Крок 5. Системне відображення зворотних рейсів.
Крок 6: Системне відображення деталей рейсів
Крок 7. Підтвердження рейсу.
Крок 8: Реєстрація нового користувача.
Крок 9: Вхід.
Крок 10: Надання інформацію про пасажира.
Крок 11: Відображення вільних місць.
Крок 12: Вибір місця.
Крок 13: Введення платіжної інформації.
Крок 14: Введення номеру підтвердження.
Крок 15: Звершення алгоритму.

На рисунку 1 показано діаграму варіанту використання для користувача.

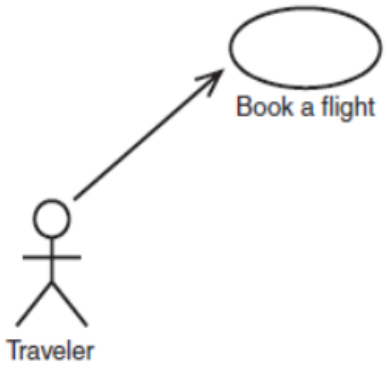

Рис. 1. Користувач та варіант використання

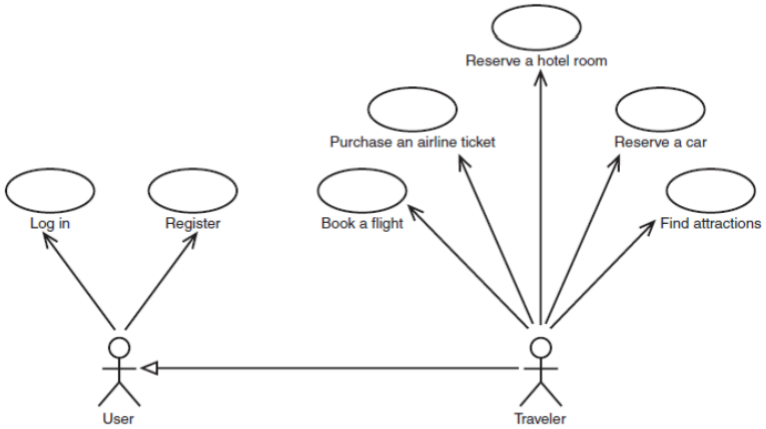

Рис. 2. Варіант використання для нового на зареєстрованого користувача

Основною метою структурування моделей є усунення будь-якої надмірності, що полегшить розуміння та підтримку визначених варіантів використання. Для цього визначається структуру відстеження змін до вимог в програмній системі, що зображено на відповідній діаграмі (рис. 3).

Потреби замовника (STRQ) будуть відстежуватися до функціоналу (FEAT), визначених у документі вимог та додаткових вимог. Між STRQ і FEAT може існувати зв'язок «багато-до-багатьох», але зазвичай це один запит зацікавленої сторони до багатьох функцій. Кожен схвалений запит має стосуватися принаймні однієї функції або додаткової вимоги.

Вимоги до функціоналу (FEAT) будуть відстежуватися або у конкретному випадку використання,

або в додатковій вимозі. Кожен схвалений функціонал має відстежувати принаймні один варіант використання або додаткову вимогу. Між функціями, варіантами використання та додатковими вимогами можуть існувати зв'язки «багато-до-багатьох».

Вимоги до варіантів використання (UC), визначені в специфікаціях варіантів використання, будуть відстежуватися до функціоналу.

Додаткові вимоги (SUPL) будуть відстежуватися до функціоналу.

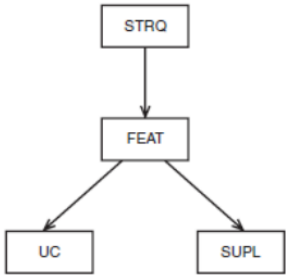

Рис. 3. Структура відстеження змін до вимог для прикладу «забронювати рейс»

Висновки

Отже, у даній статті було здійснено об'єктно-орієнтований аналіз для прикладу «забронювати авіаквиток» і отримано діаграми варіантів використання. Відповідно до визначених варіантів використання було отримано алгоритм та залежності між основними фазами внесення змін до програмної системи. Запропонований аналіз сильно залежить від добре визначених вимог до програмного забезпечення та не функціональних вимог, які можна відстежити. Саме завдяки цьому на основі аналізу змін у вимогах можна визначити вплив на діаграму класів до атрибутів даного випадку, а також визначити тестові варіанти, які слід змінити, щоб врахувати більше потенційних помилок. Це дозволить не нести збитки у майбутньому, що покращить загальну надійність системи та збільшить шанс на успішну її реалізацію.

References

1. Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli, Fundamentals of Software Engineering, Prentice Hall Publishing, 2011.
2. Chandra Shrivastava, Carver D. L. Using Low-Level Software Architecture for Software Maintenance of Object-Oriented Systems. Proceedings of the 1995 Software Engineering Forum, Boca Raton, FL, November, pp. 31-40, 2005.
3. Chen X., Tsai W., Hunag H., Poonawala M., Rayadurgam S., Wang Y. Omega-an Integrated Environment for C++ Program Maintenance. Proceedings of the International conference on software Maintenance, pp. 114-123, 1996.
4. Li L., Offutt A. J. Algorithmic Analysis of the Impact of Changes to Object-oriented Software. Proceedings of the International Conference on Software Maintenance, pp. 171-184, 1996.
5. Hutchins M., Gallagher K. Improving Visual Impact Analysis, Proceedings of the International Conference on Software Maintenance, pp. 294-301, 2016.
6. Bohner S. A. Software change impacts–an evolving perspective, Proceedings of the International Conference on Software maintenance, pp. 263–272, 2020.
7. Peter Zielczynski, IBM, Requirements Manangement Using IBM Rational Requisite Pro, 2013.
8. Sarah Maadawy and Akram Salah, Measuring Change Complexity from Requirements: A proposed methodology, IMACST. Volume 3, 2012.