УДК 004.89

Є.В. БУРОВ

ЕФЕКТИВНІСТЬ ЗАСТОСУВАННЯ ОНТОЛОГІЧНИХ МОДЕЛЕЙ ДЛЯ ПОБУДОВИ ПРОГРАМНИХ СИСТЕМ

Анотація. *У статті наведено результати аналізу ефективності застосування онтологічного моделювання для побудови програмних систем. Задачу аналізу ефективності розглянуто в контексті підвищення якості програмного продукту на всіх етапах його життєвого циклу. Запропоновано метрики для вимірювання ступеня підвищення якості для різних джерел та етапів життєвого циклу програмного продукту.*
Ключові слова: *онтологічна модель, програмна система, онтологія, якість програмного продукту.*

Аннотация. *В статье приведены результаты анализа эффективности использования онтологического моделирования для построения программных систем. Задача анализа эффективности рассмотрена в контексте повышения качества программного продукта на всех этапах его жизненного цикла. Предложены метрики для измерения степени повышения качества для разных источников и этапов жизненного цикла продукта.*
Ключевые слова: *онтология, онтологическая модель, программная система, качество программного продукта.*

Abstract. *The results of effectiveness analysis of ontology models usage for software systems design are presented. The effectiveness is considered as software quality increase due to ontological models usage on all parts of its life-cycle. Metrics for measuring software quality increase for different sources and stages of life-cycle of software product are suggested.*
Keywords: *ontology model, software system, ontology, software product quality.*

1. Вступ та загальна характеристика проблеми

Збільшення ступеня інтеграції бізнес-процесів підприємств, розвиток інформаційних технологій та глобалізація світової економіки приводять до збільшення темпів змін бізнес-середовища і, як наслідок, необхідності постійної адаптації програмних систем до змінного середовища їх функціонування.

Традиційні методи проектування та архітектурні принципи побудови програмних систем орієнтовані на створення програмного продукту на основі фіксованого набору вимог, що призводить до створення систем, які погано реагують на зміну вимог та зовнішніх чинників, які дорогі у підтримці та експлуатації і в яких нерідко трапляються збої та аварії [1]. Врахування змінених вимог, як правило, призводить до необхідності випуску нової версії програмного продукту, що передбачає виконання тривалих та ресурсоємних етапів аналізу, дизайну, кодування, тестування та впровадження нової версії продукту.

Адаптація програмної системи до змін бізнес-середовища вимагає врахування знань експертів відповідних предметних областей. Як правило, такі експерти не являються спеціалістами у галузі програмування. Звідси виникає необхідність чіткого формулювання вимог експертами для розробників системи. Помилки та непорозуміння у керуванні вимогами до системи під час її концептуалізації становлять значну частину серед причин невдач у розробці програмних продуктів [2].

Одним із шляхів вирішення зазначеної проблеми є відокремлення логіки функціонування програмної системи від механізму її опрацювання та реалізації. При цьому логіка функціонування подається у вигляді певної формальної моделі.

Одним із способів реалізації такого підходу є MDD/MDA (Model – driven design/model-driven architecture), започаткований у роботах Шлаера (Shlaer) та розвинений Меллором (Mellor) [3], зокрема, як xUML-технологія створення програмних систем. Ця

технологія полягає у формалізації програми у вигляді моделей та наступної їх компіляції у код. Багаторічне використання цього підходу, крім переваг, виявило і його суттєві недоліки, зокрема, складність створення та модифікації комплексу моделей, яка є порівняною зі складністю створення системи традиційним шляхом.

Альтернативним способом реалізації MDD є використання онтологій для побудови програмних систем. За визначенням Грубера [4], онтологія є формальною моделлю концептуалізації предметної області. Така модель містить визначення сутностей предметної області та залежностей між ними. Методи онтологічного інжинірингу на сьогодні широко застосовуються для вирішення задач змістовного пошуку інформації, реферування документів, підтримки прийняття рішень [5].

Використання онтологій для побудови програмних систем приводить до уникнення повторної концептуалізації предметної області, що дозволяє скоротити використання ресурсів на етапі аналізу та дизайну системи, зменшує кількість помилок концептуалізації. Перепоною до використання онтології при побудові програмних систем є їх значна складність, декларативний характер онтологій, відсутність подання в них процедурних знань. Подолати зазначені проблеми та обмеження доцільно шляхом побудови та використання виконувальних онтологічних моделей багаторазового використання. Така модель є фрагментом загальної онтології і містить елементи онтології, релевантні для вирішення конкретної задачі, доповнені формальним описом операцій та логіки вирішення цієї задачі.

В [6] був запропонований підхід до побудови програмних систем з використанням інтерпретованих онтологічних моделей, подані архітектура та порядок роботи системи моделювання. При цьому невирішеною залишилася задача аналізу ефективності застосування онтологічного підходу для побудови програмних систем, виявлення джерел ефективності та розроблення відповідних метрик.

Метою цієї роботи є аналіз ефективності застосування онтологічних моделей для побудови програмних систем.

2. Подання знань, структура та порядок роботи системи моделювання з використанням онтологічних моделей

Значна частина розробок у галузі представлення знань у системах штучного інтелекту зосередилася на вивченні представлення декларативних знань. Розроблені математичні формалізми для подання декларативних знань, зокрема, Description Logic, яка використовується для подання декларативної складової з використанням семантики, що базується на логіці першого порядку. З використанням цих формалізмів було розроблено ряд фреймових систем, зокрема, KL-ONE [7].

Декларативні системи подання знань покладені в основу сучасних напрямів розробки семантичного веб, семантичних веб-сервісів та інших семантично-орієнтованих технологій [8].

Перевагами декларативних систем подання знань є можливість створення цілісної формальної моделі предметної області, досягнення однозначного трактування концептів предметної області у різних задачах, ідентифікація та формальне представлення простих залежностей між концептами. Водночас декларативні системи подання знань та пов’язані з ними механізми логічного виводу мають і ряд суттєвих недоліків, зокрема, велику складність самої онтології та пов’язані з цим значні витрати на створення та супровід декларативних онтологічних систем, процедурна складова у мережевих моделях має другорядне значення та досить спрощена, вони непридатні до подання знань, орієнтованих на вирішення конкретних задач, у них відсутнє подання мети, вони непристосовані до подання та використання процедурних абстракцій.

Декларативний підхід до представлення та використання знань значно відрізняється і від розуміння когнітивного процесу людини у когнітивній психології, поданого теорією

схем [9]. Згідно з цією теорією, прийняття рішення людиною здійснюється шляхом пошуку, активації та використання певної достатньо простої схеми. Схеми можуть містити інші схеми та бути пов'язані між собою різноманітними типами зв'язків. Прийняття рішення завжди здійснюється у певному контексті, який визначається множиною активованих у даний момент схем та метою вирішення задачі.

Декларативні знання відображають концепти та залежності між ними. Вони моделюють відношення загальне-детальне, реалізують наслідування, дозволяють групувати концепти. Водночас вони не дозволяють указати, як вирішити конкретну задачу, як досягнути вказаної мети, які проміжні задачі потрібно вирішити та які цілі досягнути. Значна частина знань людства – це знання про способи вирішення задач та досягнення цілей, а також знання, що містяться у шаблонах, стереотипах, архетипах. Для створення інтелектуальних систем необхідно формально подавати такі знання та опрацьовувати їх машинно.

Для відображення процедурних знань було запропоновано [6] використати онтологічні моделі, що відображають сутності, відношення та обмеження онтології в контексті конкретної задачі та доповнені операціями над цими елементами.

Система моделювання, що реалізує запропонований підхід, складається з таких компонент (рис. 1). База фактів містить факти про об'єкти та події зовнішнього світу, необхідні для вирішення задач системою. Всі факти семантично інтерпретовані, тобто подані як об'єкти певних класів, визначених онтологією. Інформація в базі фактів впорядкована за часом та змінами, що дозволяє відслідкувати стан бази на довільний момент часу або на довільну зміну. База фактів, онтологія та моделі у сукупності утворюють базу знань системи.

Онтологія містить модель предметної області, подану як таксономія класів. Це створює можливість однозначного трактування усіх об'єктів з бази фактів, визначення єдиної структури слотів та типів властивостей для них. Крім того, онтологія містить відношення, правила та обмеження, які мають загальний, системний характер.

Моделі визначають, які задачі вирішуються в кожний момент у системі і як вони вирішуються. Посилання на моделі використовуються в онтологіях для зберігання складних правил та обмежень, зокрема, динамічних обмежень, значення яких залежить від стану бази знань або зовнішнього світу.

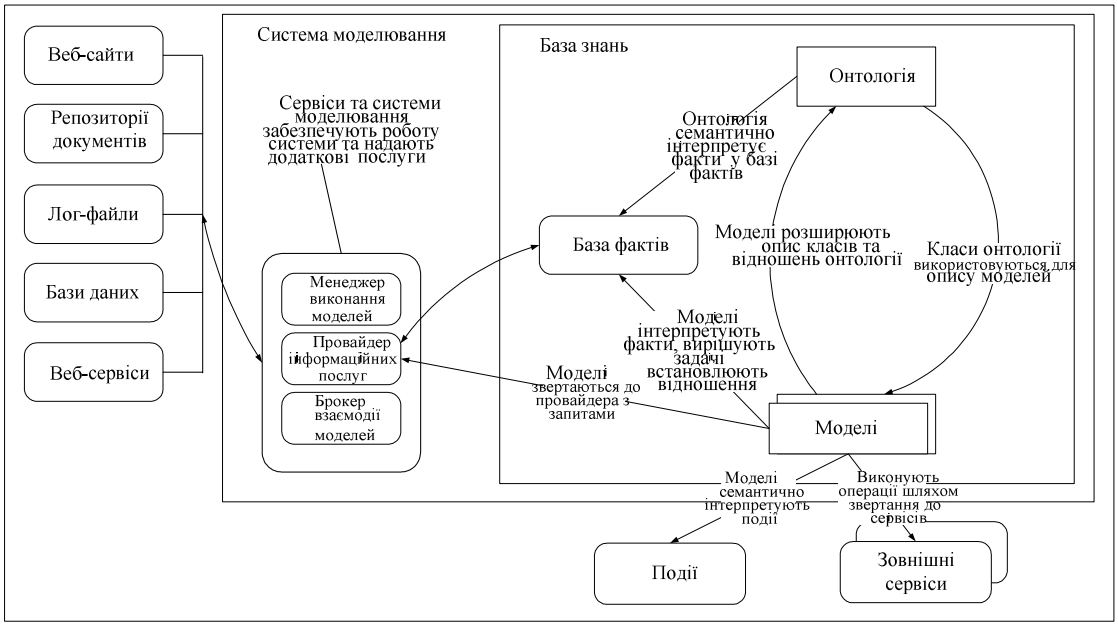

Рис. 1. Структура системи моделювання

Система реагує на визначене коло подій зовнішнього світу та опрацьовує їх, створюючи нові або модифікуючи існуючі факти. Для опрацювання цих подій використовуються відповідні моделі. Важливими компонентами системи є сервіси системи моделювання, які забезпечують виконання моделей, їх взаємодію, пошук потрібної інформації у зовнішніх джерелах. Так, Менеджер виконання моделей реалізує запуск або зупинку моделей, відслідковує використання ресурсів моделей. Брокер взаємодії моделей реалізує пошук релевантних моделей для вирішення задач, ініціалізує та запускає обрані моделі. Провайдер інформаційних послуг за запитом моделі системи звертається до зовнішніх джерел, реалізує пошук необхідних даних та їх семантичну інтерпретацію. Такими зовнішніми джерелами є, наприклад, бази даних, документи, веб-сервіси та ін.

Моделі виконують операції відповідно до відображеної в них логіки, звертаючись з запитами до зовнішніх відносно системи моделювання сервісів. Такими сервісами є сервіси операційної системи, сервіси інформаційної системи підприємства, побудованої з дотриманням вимог SOA [10], або довільні веб-сервіси.

3. Аналіз ефективності застосування онтологічних моделей для покращання якості програмних систем

Використання онтологічних моделей впливає на різні аспекти та етапи процесу створення й експлуатації програмних систем. Задачу аналізу ефективності будемо розглядати як задачу оцінки ступеня підвищення якості програмного забезпечення в результаті використання онтологічних моделей на всіх етапах його життєвого циклу.

3.1. Підходи до визначення та вимірювання якості програмної системи

Якість програмного забезпечення є комплексним поняттям, яке відображає різні аспекти процесів його створення та експлуатації. В літературі відображено такі визначення цього терміна.

• «Якість програмного забезпечення – це сукупність властивостей, які визначають корисність програми для користувачів у відповідності з функціональним призначенням та сформульованими вимогами» [11].

• «Увесь обсяг ознак та характеристик продукту або послуги, що стосується їх можливості задовольняти визначеним або передбачуваним потребам» [12].

Таким чином, якість формулюється через корисність та відповідність явним вимогам або неявним очікуванням.

Існуючі методології вимірювання якості програмного забезпечення, сформульовані у стандартах [11, 13], розглядають якість програмного продукту на різних етапах його життєвого циклу за посередництвом моделей якості. Такі моделі [11] використовуються для визначення критеріїв якості для різних типів програмних продуктів і різних предметних областей їх використання.

ISO 9126 розрізняє внутрішні, зовнішні параметри якості, а також якість на етапі використання. Параметри якості, які вимірюють на етапі розробки, називають внутрішніми (internal), параметри, які вимірюють на етапі тестування, – зовнішніми (external), а якість програмного продукту для кінцевого користувача відображена у параметрах етапу використання (рис. 2).

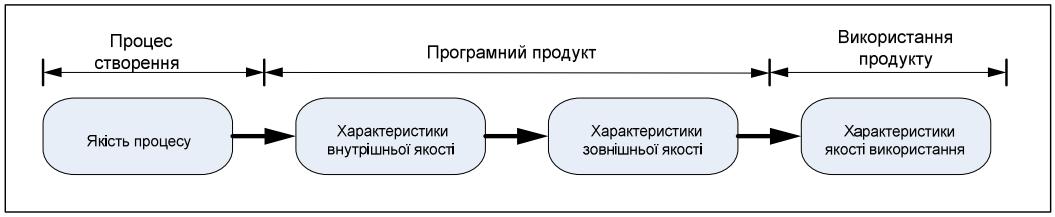

Рис. 2. Модель якості програмного продукту стандарту ISO 9126

Стандарт ISO/IEC 25010 [13] розширює модель стандарту ISO 9126 у контексті середовища, в якому функціонує програмний продукт, розглядаючи поняття якості програмної системи. Він вводить також поняття якості даних.

Кожен із стандартів якості будує таксономію характеристик якості. ISO 9126 визначає для внутрішньої та зовнішньої якості шість характеристик якості, у свою чергу, поділених на підхарактеристики.

Стандарт ISO 9126 визначає такі головні характеристики якості, як функціональність (відповідність вимогам, точність, інтероперабельність, безпека), надійність (зрілість, стійкість до відмов, поновлюваність), зручність у використанні (легкість розуміння та освоєння, зручність в експлуатації, естетичність), ефективність (часові аспекти, використання ресурсів), супровід (придатність до аналізу, зручність впровадження змін, стабільність, придатність до тестування), переносимість (зручність адаптації, придатність до інсталяції, здатність співіснувати з іншими продуктами, можливість заміни).

Стандарт ГОСТ 28195-89 [14] визначає такі характеристики якості програмного забезпечення, як надійність (працездатність та стабільність), супровід (структурність, простота, наочність, повторюваність), зручність використання (легкість освоєння, зручність експлуатації, наявність документації), ефективність (рівень автоматизації, використання ресурсів, часові параметри), універсальність (гнучкість, мобільність, придатність до модифікації), коректність (повнота, узгодженість, логічна коректність).

Як видно з порівняння стандартів ISO/IEC 9126-1 та ГОСТ 28195-89, характеристики якості у значній мірі співпадають. Водночас у стандарті ISO/IEC 9126-1 відсутні такі важливі характеристики, як структурність, простота конструкції, рівень автоматизації, перевіреність.

Для визначення ефективності застосування онтологічних моделей для побудови програмних систем необхідно визначити джерела ефективності та проаналізувати характеристики якості, на які впливає використання онтологічних моделей, розробити метрики для вимірювання ступеня покращання якості.

Розглянемо можливі джерела ефективності застосування онтологічних моделей для побудови програмних систем для різних етапів життєвого циклу програмного продукту.

3.2. Повторне використання знань на етапі передпроектного аналізу та розробки архітектури

Вагомим джерелом збільшення якості програмного забезпечення, побудованого з використанням онтологічних моделей, є повторне використання концептів, знань, структур даних. При цьому повторне використання має місце на різних етапах життєвого циклу програмного продукту.

На етапі концептуалізації розробник програмного продукту вирішує задачу побудови концептуалізації предметної області в контексті задач, які повинен вирішувати програмний продукт. Розробник виділяє у предметній області необхідні сутності, визначає їх

атрибути та обмеження, залежності між ними. Помилки, допущені на етапі концептуалізації вважають найбільш вагомими, і виправлення їх нерідко вимагає перероблення всього програмного продукту [2]. Різні програмні продукти часто вимагають концептуалізації одних і тих самих об'єктів реального світу. При цьому великий обсяг робіт з первинної концептуалізації та аналізу предметної області фактично дублюється. При використанні онтологічного підходу розробник використовує наявну концептуалізацію предметної області, відображену в онтології. Якщо онтологія неповна, він доповнює її новими елементами, які в подальшому можуть бути використані іншими розробниками.

Прикладами концептів, які використовуються різними програмними продуктами та багатократно концептуалізуються у процесі створення програмного забезпечення, є концепти особи, організації, користувача, інформаційного ресурсу та ін.

Уникнення повторної концептуалізації безпосередньо впливає на такі характеристики якості, як відповідність вимогам, точність, інтероперабельність, зрілість, використання ресурсів на етапі проектування, переносимість та можливість співіснування з іншими продуктами. Використання готової концептуалізації у вигляді онтології предметної області спрощує процес початкової концептуалізації при створенні програмного продукту, дозволяє уникнути помилок концептуалізації.

Для різних програмних продуктів, що використовують спільну концептуалізацію, простіше організувати взаємодію. За рахунок багатократного використання онтології у різних, реально функціонуючих, програмних продуктах досягається високий ступінь зрілості концептуалізації, а також стабільності та надійності, що досягається шляхом багатократного та різностороннього тестування, яке проходить онтологія під час створення та експлуатації продуктів на її базі.

Метрику, що вимірює ступінь повторного використання концептуалізації, визначимо як відношення кількості доданих сутностей в онтологію (N_{en-new}) до кількості вже існуючих в онтології сутностей, використаних при побудові даного продукту (N_{en-old}).

$$C_{ent-re} = N_{en-new} \Big/ N_{en-old} . \qquad (1)$$

Метрика (1) розглядає тільки сутності онтології і є досить наближеною. Враховуючи те, що при створенні нового продукту до онтології додають (або модифікують) не тільки сутності, але й відношення, обмеження, правила, точнішою буде метрика, що враховує усі додані або змінені елементи онтології:

$$C_{en-\det} = N_{el-new} \Big/ N_{el-old} ,$$

де N_{el-new} – кількість доданих (або змінених) елементів онтології;

N_{el-old} – кількість використаних (та не змінених) елементів онтології.

Продукти, що базуються на спільній концептуалізації, мають кращу інтероперабельність за рахунок уникнення помилок, пов'язаних з різним поданням предметної області в них.

Подамо метрику $C_{\text{int}}(A,B)$ для оцінки погіршення інтероперабельності, яке виникає через відмінності концептуалізації, для двох продуктів A та B:

$$C_{\text{int}}(A,B) = \frac{N_{diff}^{A} + N_{diff}^{B}}{N_{comm}^{AB}},$$

де N_{diff}^{A} – кількість відмінних елементів в онтології продукту A порівняно з продуктом B;

N_{diff}^{B} – кількість відмінних елементів в онтології продукту В порівняно з продуктом А;

N_{comm}^{AB} – кількість спільних елементів онтології, які використовують продукти А та В.

3.3. Повторне використання структур даних та коду на етапі дизайну та кодування

Використання єдиної онтології на етапі дизайну та кодування дозволяє розширити ступінь повторного використання коду, поданого онтологічними моделями. Моделі, що базуються на спільній онтології, є сумісними і можуть використовуватися разом.

Якщо порівнювати підхід, який базується на онтологічних моделях з об'єктно-орієнтованим програмуванням (ООП), то кожен клас ООП має декларативну частину, подану оголошенням змінних, та процедурну частину, подану специфікацією методів. При використанні онтологічних моделей всі моделі (які є аналогами методів) мають спільну декларативну частину, подану онтологією.

При використанні об'єктно-орієнтованого програмування переносимість обмежена повторним використанням груп взаємозалежних класів [15]. При цьому методи класу не є переносимими і можуть використовуватися тільки з екземплярами даного класу.

Використання онтологій для побудови програмної системи дозволяє уникнути багатократного декларування об'єктів у різних класах, підвищити переносимість коду, зменшити використання ресурсів на стадії кодування за рахунок повторного використання коду. Характеристики якості, що покращуються при цьому, переносимість, зменшення використання ресурсів у процесі проектування.

Запропонуємо такі метрики для вимірювання переносимості коду.

1. Частина програмних конструкцій (класів, моделей, сервісів) у наявних бібліотеках коду, які можна використовувати без змін:

$$C_{port-code1} = \frac{N_{portable}}{N_{all}},$$

де $N_{portable}$ – кількість програмних конструкцій, які переносяться без змін;

N_{all} – загальна кількість конструкцій.

Зауважимо, що у випадку використання онтологічних моделей $C_{port-code1} = 1$.

2. Кількість додаткових програмних конструкцій (класів, бібліотек), які необхідно долучити до коду продукту при використанні конкретного класу (моделі) – $C_{port-add}$.

3.4. Ефективність використання онтологічних моделей на етапі тестування

На етапі тестування програмного продукту використання онтологічних моделей призводить до зменшення обсягів тестування. При цьому тестуванню підлягають розроблені або модифіковані моделі, а не продукт загалом.

При традиційному підході до тестування програмних продуктів їх тестують, виходячи з варіантів використання (use-cases). Для кожного такого варіанта визначають набір тестових сценаріїв. Якщо програмний продукт змінено, то важко визначити залежності між змінами у коді та сценаріями використання системи. Тому перетестовують усі сценарії використання або їх значну частину.

У випадку побудови програмного продукту на базі онтологічних моделей моделі, що додані до продукту без змін, у них можна не тестувати.

Ще одним джерелом спрощення процесу тестування є те, що моделі описують спосіб вирішення певної задачі і, як такі, часто відповідають певному сценарію викори-

стання системи. При цьому об'єктом тестування стають моделі, зменшуються кількість сценаріїв тестування та витрати часу на створення сценаріїв тестування.

Виміряти ступінь зменшення трудомісткості проведення тестування можна на прикладі програмного продукту, перенесеного на нову онтологічну платформу. При цьому доступні тестові сценарії як для традиційної архітектури, так і для нової. Метрикою, що вимірює ступінь зменшення трудомісткості проведення тестування, є відношення кількості сценаріїв у новій платформі N_{mod} до кількості сценаріїв у традиційній платформі N_{old} з урахуванням середньої складності (наприклад, кількості кроків) кожного сценарію $(C_{av-test-\mathrm{mod}}, C_{av-test-old})$.

$$C_{test} = \frac{N_{\mathrm{mod}} \times C_{av-test-\mathrm{mod}}}{N_{old} \times C_{av-test-old}}.$$

Важливим додатковим фактором покращання якості програмного продукту, побудованого з використанням онтологій, є те, що моделі, які повторно використовуються у ньому, попередньо вже пройшли тестування та практично використовуються в інших програмних продуктах, що дозволяє говорити про повторне використання тестування, проведеного для інших програмних продуктів, а також припустити зменшення кількості помилок та збільшення ступеня зрілості продукту.

Головні характеристики якості програмного продукту, покращені за умови використання онтологічних моделей на етапі тестування, – придатність до тестування, стабільність, зрілість.

3.5. Повторне використання концептів, знань, структур даних та коду на етапах підтримки й експлуатації програмного продукту

Якість програмного продукту після його завершення та введення в експлуатацію містить багато аспектів, пов'язаних з такими характеристиками якості програмного забезпечення, як зручність в експлуатації, придатність до аналізу, зручність впровадження змін, зручність адаптації, можливість заміни. Ці характеристики загалом відповідають таким групам характеристик якості, як зручність у використанні (usability) та супроводі (maintainability), переносимість (portability) коду.

У сучасних умовах придатність програмного продукту до швидкої адаптації при змінах у бізнесовій ситуації та проблемній області є важливою вимогою. У традиційному підході до побудови та розробки програмного продукту будь-які зміни у вимогах вимагають розробки нової версії програмного продукту, що передбачає виконання етапів аналізу, проектування архітектури, кодування, тестування та впровадження в експлуатацію. Розробка нової версії продукту займає тривалий час і вимагає значних коштів.

Визначимо метрику складності модифікації програмного продукту традиційним способом:

$$C_{\mathrm{m}-old} = T_{ov} \times \sum_{i=1}^{n} K_{ov}^{i} + T_{ch} \times \sum_{j=1}^{m} K_{ch}^{j},$$

де n – загальна кількість програмних елементів (класів, функцій), m – кількість програмних елементів, які підлягають модифікації, K_{ov} – метрика складності i-того компонента, T_{ov} – середній час, що витрачається на аналіз, тестування, компоновку з розрахунку на один програмний компонент, K_{ch} – метрика складності компонента, що змінюється, T_{ch} – середній час, що витрачається додатково на аналіз, перекодування та перетестування з розрахунку на один програмний компонент, що змінюється.

Водночас зміни у вимогах та предметній області, які спонукають до розробки нової версії продукту, мають різний характер, що робить доцільним їх окремий аналіз. Розглянемо такі варіанти змін.

Заміна методу вирішення задачі або формування нового методу комбінацією існуючих. Подання предметної області залишається без змін.

Розробка нового методу вирішення задачі або уточнення існуючого.

Уточнення подання предметної області шляхом уведення нових сутностей, відношень та обмежень. При цьому визначені попередньо елементи предметної області не змінюються.

Зміна розуміння предметної області, яка відображена у модифікації та заміні попередньо визначеного формального опису цієї області.

У випадку заміни методу вирішення задачі або формування нового методу як комбінації з існуючих, модифікація коду є тривіальною і проводиться експертом предметної області в середовищі проектування. Додатковий час при цьому витрачається на тестування та підготовку даних для нього. Об'єктом зміни виступає модель. Оцінити складність проведення зміни можна з використанням такої метрики:

$$C_{md} = K_m \times (T_m + T_t),$$

де K_m – метрика складності (кількість компонент) моделі, T_m – час, що витрачається в середньому на модифікацію, та T_t – тестування в розрахунку на один компонент.

Якщо потрібно розробити нову модель або модифікувати існуючу, то, порівняно з попереднім випадком, збільшуються витрати часу на створення і тестування моделі. При цьому також об'єктом модифікації й тестування виступає модель, і складність цього процесу, порівняно з модифікацією традиційного коду, є незначною.

У випадку введення нових сутностей, відношень або обмежень в онтологію, при збереженні існуючих її частин без змін, проектувальник онтології проводить валідацію запропонованих змін. Він аналізує зміни на предмет відсутності конфліктів з наявними елементами онтології та обгрунтовує доцільність модифікації онтології. Модифікація онтології проводиться у редакторі онтологій. Складність проведення модифікації залежить від кількості нових елементів, доданих до онтології.

$$C_{md-ont-add} = K_{\mathrm{mod}} \times (T_{an} + T_{\mathrm{mod}}),$$

де $C_{mod-ont-add}$ – метрика складності, K_{mod} – кількість доданих до онтології компонент, T_{an} – час на аналіз та валідацію, T_{mod} – час на проведення модифікації у редакторі онтологій з розрахунку на один компонент.

Причиною зміни існуючих елементів онтології є переосмислення змісту предметної області, а також виявлення змістовних та логічних помилок в онтології. Зміна існуючих елементів онтології вимагає змінити та перетестувати усі моделі, які працюють зі зміненими елементами. Така задача є доволі трудомісткою, адже кількість моделей у репозиторії моделей є значною. Архітектура та функціональні можливості системи моделювання роблять тривіальною вирішення задачі знаходження усіх моделей, на які впливають зміни в онтології. Для кожної такої моделі необхідно провести аналіз впливу змін, визначити обсяг необхідних модифікацій, провести зміну моделі та перетестувати змінену модель.

$$C_{md-on-rev} = (K_{on} \times T_{on}) + K_{md-on} \times (T_{an} + T_{md-\mathrm{mod}} + T_{test}),$$

де $C_{mod-on-rev}$ – метрика складності, K_{on} – кількість змінених елементів онтології, T_{on} – час, що витрачається на модифікацію одного елемента онтології у редакторі онтологій, K_{md-on}

– кількість моделей, які необхідно змінити, T_{an} – час на аналіз та проектування зміни в моделі, T_{md-mod} – час на проведення зміни, T_{test} – час на тестування.

При цьому задачі проведення змін та тестування проводяться у середовищі проектування і не займають значного часу. Найбільшим є час виконання аналізу та проектування змін у моделях.

При порівнянні складності модифікації програмної системи для випадків 1–4 неможливо дати точні оцінки метрик складності, не враховуючи особливості конкретних проектів та колективів розробників. Водночас доцільно ранжувати їх на основі експертних оцінок. Порівнюючи складності, отримуємо шкалу метрик, в якій складність зростає:

$$(C_{md}, C_{md-ont-add}, C_{md-ont-rev}, C_{m-old}).$$

У випадку зміни подання онтології ще одним джерелом збільшення якості програмних продуктів на основі онтологічних моделей є проактивне виявлення та виправлення дефектів, які виникають через несвоєчасне врахування у програмному продукті змін у предметній області. При використанні онтологічних моделей і внесенні зміни в онтологію переробляють усі залежні від цієї зміни моделі. Таким чином усувають можливі дефекти, що виникають через неузгодженість предметної області та моделей ще до того, як вони проявляться як помилки у роботі програм.

При традиційному підході до створення програмних продуктів зміни у предметній області часто враховують тільки тоді, коли вже виявлені некоректні результати в роботі програмної системи. При цьому виправлення дефектів, як правило, обмежується конкретним програмним продуктом. Інші програмні продукти, що залежать від тих самих особливостей предметної області, оновлюються окремо, що потребує додаткових затрат ресурсів на аналіз та модифікацію. У випадку програмних систем, що використовують онтологічні моделі, усі залежні моделі оновлюють разом.

Таким чином, використання онтологічних моделей для побудови програмних систем приводить до покращання таких характеристик якості, як коректність, кількість дефектів, відповідність функціональним вимогам, зручність впровадження змін.

Онтологічні моделі мають невелику кількість елементів та просту структуру. Вони подаються з використанням концептів предметної області і орієнтовані на специфікацію методу вирішення конкретної задачі. При виникненні необхідності зміни моделі ця зміна також формулюється з використанням концептів предметної області. Створення та модифікація моделі не вимагають знання мов програмування й проведення кодування. Таким чином, до вирішення задач створення та підтримки моделей можна залучити фахівців з визначених предметних областей.

У традиційному підході до створення програмних систем взаємодія розробника та фахівця предметної області по виявленню та формалізації вимог до системи є окремим етапом розробки, на якому є ризик виникнення помилок через неповне або хибне взаєморозуміння розробника та фахівця предметної області. При використанні онтологічних моделей фахівець предметної області самостійно створює модель з наявних в онтології сутностей та залежностей. Таким чином, фактично усунуті можливі причини помилок, що виникають через неправильне формулювання вимог до системи.

4. Висновки

Підведемо підсумки аналізу впливу використання онтологічних моделей для побудови програмних продуктів на характеристики якості програмного продукту (табл. 1).

Таблиця 1. Джерела підвищення якості програмного продукту з використанням онтологічних моделей та метрики для оцінки

Назва етапу життєвого циклу	Характеристики якості, покращені за рахунок використання онтологічних моделей	Джерело покращання характеристик якості	Метрики для оцінки ступеня покращання характеристик якості
Аналіз вимог та розробка архітектури системи	Відповідність вимогам, точність, інтероперабельність, переносимість, використання ресурсів	Ступінь повторного використання концептуалізації	$C_{ent-re} = N_{en-new} / N_{en-old}$, $C_{en-\det} = N_{el-new} / N_{el-old}$
		Ступінь зміни інтероперабельності	$C_{\text{int}}(A,B) = \frac{N^A_{diff} + N^B_{diff}}{N^{AB}_{comm}}$
Дизайн та кодування	Переносимість коду, використання ресурсів	Відсоток програмних конструкцій, які можна використовувати без змін	$C_{port-code1} = \frac{N_{portable}}{N_{all}}$
		Кількість додаткових програмних конструкцій	$C_{port-add}$
Тестування	Придатність до тестування, стабільність, зрілість	Зменшення трудомісткості проведення тестування	$C_{test} = \frac{N_{\text{mod}} \times C_{av-test-\text{mod}}}{N_{old} \times C_{av-test-old}}$
Підтримка та експлуатація	Зручність у використанні та супроводі, переносимість	Складність модифікації програмного продукту	Традиційним способом: $C_{\text{m}-old} = T_{ov} \times \sum_{i=1}^{n} K^i_{ov} + T_{ch} \times \sum_{j=1}^{m} K^j_{ch}$
			Якщо змінюється метод вирішення задачі: $C_{md} = K_m \times (T_m + T_t)$
			При введенні нових елементів в онтологію: $C_{md-ont-add} = K_{\text{mod}} \times (T_{an} + T_{\text{mod}})$
			Якщо модифікується онтологія: $C_{md-on-rev} = (K_{on} \times T_{on}) + K_{md-on} \times (T_{an} + T_{md-\text{mod}} + T_{test})$

Використання онтологічних моделей для побудови програмних систем дозволяє покращити якісні характеристики на всіх етапах життєвого циклу програмного продукту. Головним джерелом покращання якісних характеристик при цьому виступає можливість повторного використання концептуалізації та коду, поданого відповідними моделями. За рахунок використання єдиної концептуалізації при створенні різних програмних продуктів зростає ступінь їх інтероперабельності, переносимість коду.

СПИСОК ЛІТЕРАТУРИ

1. Model-driven systems development / L. Balmelli, D. Brown, M. Cantor [et al.] // IBM Systems Journal. – 2006. – Vol. 45. – P. 569 – 585.
2. Charvat J. Project Management Methodologies / J. Charvat. – Wiley, 2003. – 264 p.

3. Mellor S.J. Executable UML: A foundation for model-driven architectures / S.J. Mellor, M.J. Balcer. – Boston: Addison-Wesley, 2002. – 416 p.
4. Gruber T.R. A translation approach to portable ontology specifications / T.R. Gruber // Knowledge Acquisition. – 1993. – Vol. 5. – P. 199 – 220.
5. Норенков И.П. Интеллектуальные технологии на основе онтологий / И.П. Норенков // Информационные технологии. – 2010. – № 1. – С. 17 – 23.
6. Буров Є.В. Концептуальне моделювання інтелектуальних програмних систем / Буров Є.В. – Львів: вид-во Львівської політехніки, 2012. – С. 432.
7. Brachman R.J. An Overview of the KL-ONE Knowledge Representation System / R.J. Brachman, J.G. Schmolze // Cognitive Science. – 1985. – Vol. 9. – P. 171 – 216.
8. Daconta M.C. The Semantic Web: A Guide to the Future of XML, Web Services and Knowledge Management / Daconta M.C., Obrst L.J., Smith K.T. – Wiley, 2003. – 312 p.
9. Rumelhart D.E. Schemata: The building blocks of cognition / D.E. Rumelhart // Theoretical Issues in Reading Comprehension / R.J. Spiro, B.C. Bruce, W.F. Brewer (eds.). – Hillsdale, NJ: Erlbaum, 1980. – P. 33 – 58.
10. Erl T. SOA Principles of Service Design / Erl T. – Boston: Prentice Hall, 2007. – 608 p.
11. ISO/IEC 9126-1:2001. Software engineering – Product quality – Part 1: Quality model [Електронний ресурс]. – Режим доступу: http://www.iso.org/iso/catalogue_detail.htm? csnumber= 22749.
12. ISO 8402:1994. Quality management and quality assurance – Vocabulary. [Electronic resource]. – Режим доступу: http://www.iso.org/iso/catalogue_detail.htm?csnumber=20115.
13. ISO/IEC 25010:2011. Systems and software engineering. – Systems and software Quality Requirements and Evaluation (SQuaRE). – System and software quality models [Електронний ресурс]. – Режим доступу: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=35733.
14. ГОСТ 28195-89. Оценка качества программных средств. Общие положения [Електронний ресурс]. – Режим доступу: http://www.gametest.ru/doc/sw/28195_89.pdf.
15. Милютин А. Метрики кода программного обеспечения [Електронний ресурс] / А. Милютин. – Режим доступу: http://www.viva64.com/ru/a/0045.

Стаття надійшла до редакції 27.09.2012