ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

УДК 629.735.33(043.2)

Hryshchenko A.B.
National Aviation University, Kyiv

MATURITY MODEL CREATION METHODOLOGY

Since the appearance of first maturity model more than twenty years ago almost two hundreds more came up and still we do not have a clue how good and consistent maturity model should look like. Their application went out of IT boundaries and now they are widely used in different business process now. We can justify this with the central paradigm of these models – processes and their improvement. However, numerous shortcomings have been disclosed referring to both maturity models as design products and the process of maturity model design. Whereas research has already substantiated the design process, there is no holistic understanding of the principles of form and function – that is, the design principles – maturity models should meet.

Maturity models usually include a sequence of levels (or stages) that together form an anticipated, desired, or logical path from an initial state to maturity.

Usually maturity models are created to be independent from application domain peculiarities, so they may refer to processes as well as other entities (e.g. people, specific application domain objects). Aside of that, we can consider them from resources-view side distinguishing assets (i.e., process in- and outputs) and capabilities (i.e., repeatable patterns of action in the use of assets).

We can define three 3 main purposes maturity model can surve: descriptive, prescriptive and comparative.

Descriptive purpose mainly lays in giving a solid explanation on what is maturity in the given case, how we can evaluate current object. Main point here is that maturity model is diagnosis tool.

Prescriptive purpose outlines what we should do, providing specific and detailed courses of action on how to identify desirable maturity levels and provides guidelines on improvement measures.

Comparative purpose is used for internal or external benchmarking. Given sufficient historical data from a large number of assessment participants, the maturity levels of similar business units and organizations can be compared.

While creating a maturity model of our own we can use many different technologies modifying them up to our needs. Technologies used in IT development will perfectly fit, however we should remember to keep the process iterative, because evaluating and defining new stage is very important for thorough and fully described maturity model. In addition to this we should keep in mind that we should create detailed documentation. It is required for a better understanding and for comparative purpose directly. New maturity model should provide an innovative view on the problem and its solution.

Scientific Supervisor M.O Sidorov, Doctor of Technical Studies, professor

УДК 629.735.33(043.2)

Chepyuk N.V
National aviation university, Kyiv

USAGE OF ONTOLOGIES IN THE SOFTWARE ENGINEERING

Since the appearance of software engineering as the discipline and to these days the problems of modularization, reuse, integration and distribution of software components are among the central issues of the discipline. The more these tasks are automating and extending, the more important the definition and usage of ontologies as conceptual basis of these components becomes.

As ontologies are helping to formally represent knowledge in the form of sets of concepts within the application domain and the relationships between pair of concepts, they can be used to describe the application domain of the software engineering as the discipline itself as well as to describe the application domain of a particular problem software engineering strives to solve.

The usage of modeling principles is the basis of the software design process. Model Driven Development (MDD) is the relatively new software discipline which helps to design and develop software on the basis of modeling. The main concept of MDD is to increase the productivity of software developer's work by increasing the level of abstraction when developing software. To obtain such results MDD uses models, which are defined with the help of modeling languages (UML for instance). The models of the modeling languages are called metamodels.

There exists the synergy between software modeling languages and ontologies. In particular, it's based on the similarities between the standard concepts of UML and those of ontologies (for example, class, relations, inheritance). It was proposed to use UML for modeling ontologies due to the wide acceptance of UML by software engineers. Moreover, the benefit can also be obtained from usage of ontology reasoning services (for example, check for consistency) to reason over UML models, since they nowadays lack the support for formal validation and may contain hidden inconsistencies and redundancies. The practical reasoning contribution is the reasoned which allows to reason over UML class diagram. Ontologies of the application domain can be useful in software design: MDD principles allow using ontologies for extending UML activity diagrams. There is also the work in the area of extending the OWL for the means of using UML composite structures.

Object Management Group (OMG) initiated a standardization process to issue the request for proposal for Ontology Definition Metamodel (ODM). The main goal of this process was to define the metamodel for the two main ontology languages - RDF and OWL, the corresponding ontology of UML profile (to use standard UML tools for modeling ontologies) and the transformations between the ODM and other ontologies and modeling languages. This work resulted in the OMG ODM specification, published in 2009.

Scientific Supervisor M.O Sidorov, Doctor of Technical Studies, professor

УДК 004.412:004.415.53 (043.2)

Драпушко Ю.В.
Національний авіаційний університет, Київ

МЕТРИКИ ДЕФЕКТІВ ЯКОСТІ ПРОГРАМНОГО КОДУ

У технологіях виробництва програмних продуктів (ПП) тестуванню відводиться роль основного засобу забезпечення та контролю якості продукту. Це проявляється в тому, що процеси тестування все глибше інтегруються в проектні методи, а управління тестуванням стає найважливішою складовою управління проектами. Найкраще це демонструє V-модель життєвого циклу ПП.

Метрика якості коду – кількісний показник, що дозволяє оцінити програмний код з різних сторін. Для непрямої оцінки рівня якості процедурно-орієнтованих програмних засобів використовують метрики дефектів якості. Дана група метрик аналізує наявні помилки, їх щільність та кількість у програмному коді.

1. Метрика дефектів якості програмних засобів (метод Альбрехта)

DQ = Кількість помилок/Кількість рядків коду, де DQ - щільність помилок.

Цю метрику якості зручніше розрахувати на основі функціональних покажчиків (FP - function points) Алана Альбрехта.

Перевага методу - легкість обчислення.

Недолік методу - результати ґрунтуються на суб'єктивних даних.

Показник щільності дефектів обчислюють як відношення загальної кількості знайдених дефектів до кількості тестових процедур, виконаних для даної функціональності або сценарію використання системи.

2. Універсальна метрика якості коду

CRD = Кількість дефектів/хвилина, де CRD - code review defects.

Дана метрика вимірюється в ході процесу перегляду коду (code review). Для того, щоб переконатись, що код короткий, простий та відповідає діловій меті «переглядач» запускає секундомір і починає дивитись код. Кожен раз, коли він зустрічає в програмі помилку, додає до свого лічильника +1. Таким чином, розділивши показник лічильника на час, отримаємо метрику якості коду.

3.Alternative defect density - альтернативна щільність помилок, розраховується як кількість дефектів на 1000 рядків доданого коду.

4. Defect density - щільність помилок для коду автора. Якісний аналіз даної метрики допоможе визначити проблемні участки коду, ефективність роботи конкретного програміста, попередити програміста про потенційну небезпеку змін, розподілити ресурси тестування, оптимально обрати дату релізу. Отримати дану метрику можна, наприклад, за допомогою MSR Tools.

Таким чином можна хробити висновок, що описані вище метрики допомагають краще розібратись з кодом, оцінити наявний стан проекту, побачити проблемні місця, визначитись з пріоритетами задач для рефакторингу та контролювати якість програмного коду в процесі розробки.

Науковий керівник – О.П.Дишлевий старший , старший викладач

УДК 004.4 (043.2)

Книшик М.А.,студентка
Національний авіаційний університет, м. Київ

ЗАСІБ ВИМІРЮВАННЯ МЕТРИК ПОКРИТТЯ КОДУ

Метрики покриття коду – корисний інструмент для контролю якості програмного продукту. І хоча високі значення метрик не є гарантією якості, вони дають змогу знайти місця в коді, що, можливо, потребують додаткової уваги. А відстеження зміни значень протягом проекту допомагає підтримувати якість тестування та проекту вцілому на належному рівні. Проте існують деякі проблеми з їх вимірюванням. Є небагато вимірювачів, призначених спеціально для метрик покриття. Їх вимірювання не є точними та різняться в різних засобах. Дуже мало засобів мають зручний графічний інтерфейс, яким зручно користуватись. І зовсім немає засобів, які підтримували б ручне тестування.

Зважаючи на ці проблеми, було розроблено засіб для вимірювання покриття коду. Програма має простий та зрозумілий графічний інтерфейс, тому не потребує додаткового навчання. Вона здатна виміряти покриття рядків, гілок та шляхів. Результати можна представити графічно, що полегшує їх сприйняття та інтерпретацію. Також для оцінки покриття коду була введена власна метрика. Вона визначається зі значень попередніх трьох, взятих з певними ваговими коефіцієнтами. Метрика розраховується за формулою:

$$M = 0{,}2L + 0{,}3B + 0{,}5P ,$$

де *L*-покриття рядків,*B*-покриття гілок,*P*-покриття шляхів. Вона дає більш комплексне поняття покриття коду, включаючи в себе значення всіх основних метрик.

Ще одна перевага розробленого засобу в тому, що він дозволяє запускати не тільки модульні тести, але й інтерфейс програми, що тестується. Це дає можливість тестувати програми вручну, а вимірювач розрахує покриття для цих тестів.

Для вимірювання засіб використовує інструментування коду. Вихідний код програми, що вимірюється, копіюється в інший каталог. Далі вимірювач аналізує його, та додає свої оператори перед кожним рядком та в оператори умов, одночасно рахується їхня кількість. Створюється два статичних масиви з розмірністю, що відповідає кількості рядків та гілок. Після цього інструментований код запускається, а додані оператори, якщо вони виконуються, змінюють відповідні значення масивів. Незмінені значеня відповідають непокритим елементам. Виходячи з данних у масивах можна обчислити значення всіх чотирьох метрик програми.

Результати вимірювань розробленого засобу були порівняні з результатами інших засобів. Відхилень значень від вимірювачів, що працюють за тим самим принципом немає або вони незначні.

Науковий керівник – О.П. Дишлевий,старший викладач

УДК 51.681.3 (043.2)

Максимець О.М.
Київський Національний Університет ім. Тараса Шевченка

ВЕРИФІКАЦІЯ ПРОГРАМ ЗА ДОПОМОГОЮ МАШИННОГО НАВЧАННЯ

Верифікація програм є одним з найважчих (якщо не найважчим) етапом розробки програмного забезпечення. Труднощі цього етапу полягають у тому, що від розробника, крім знань чисто програмістського характеру, потрібні знання і володіння методами сучасної алгебри, логіки, комбінаторики, теорії чисел та інших суміжних областей. Крім цих суб'єктивних чинників є й об'єктивні чинники, пов'язані з тим, що в даний час наявні методи верифікації не знаходяться на достатньому рівні розвитку, який дозволяв би верифікувати системи індустріальних розмірів. Загалом картина, яка спостерігається на даний момент така, що складність програмного забезпечення постійно зростає, а методи його аналізу істотно відстають.

Теорія машинного навчання є віткою штучного інтелекту, предметом вивченням якої є системи, які можуть навчатися на даних. З розвитком теорії машинного навчання застосування їй знаходиться в інформаційних системах, які містять накопичену історію даних. Одне з відомих застосувань машиного навчання є перевірка орфографії при написанні тексту. Даними для навчання такої системи служать книжки і орфографічно правильні тексти. Підхід до навчання базується на мовних правилах і абстрактному представленні речень.

Завдяки платформам по розміщенню відкритого програмного забезпечення і багатій історії проектів з відкритим кодом існує достатньо данних, на яких міг би навчатися, так званий, верифікатор програмного коду. Таким чином, аналізуючи базу потенційно вірного програмного коду, алгоритм міг би навчитися перевіряти правильність програм і вказувати на «сумнівні» частини програми.

Алгоритм машиного навчання для верифікатора, як і у випадку з лінгвістичною мовою, має мати своє абстрактне представлення програми або її частини, яке було б незалежним від іменування зміних і мови програмування. Але, як і у випадку з орфорграфом, верифікатору не обов'язково знати «значення» програми, тобто враховувати формальну постановку задачі.

Робота алгоритму роботи верифіктора буде зводитись до представлення коду в потрібній абстрактній структурі. Другий етап ─ це пошук відповідних частин програми в базі потенційно «правильних» програм і прийняття рішення, щодо сумнівності чи правильності розглядуваної програми.

Сигналом для розробки такого верифіктора служить і те, що для платформ розміщення відкритого ПЗ певним чином розв'язана проблема оцінки якості коду. Для кожного проекту зберігаються дані про кількість виправлень, скачувань і релізів проекту. Що є своєрідною мірою правильності наявного коду.

Науковий керівник – С.Л. Кривий, д. ф.-м. н., проф.

УДК 004.413 (043.2)

Нетреба О.М., Гриненко О.О.
Національний авіаційний університет, Київ

МОДЕЛЮВАННЯ ЕКОСИСТЕМ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Екосистеми програмного забезпечення стають все більш популярним засобом організації промисловості, який використовується провідними розробниками програмного забезпечення. Отже, екосистема програмного забезпечення - це сукупність суб'єктів та відносин, які функціонують як єдине ціле і взаємодіють із ринком програмного забезпечення та послуг. Відносини спираються на єдину технологічну платформу і працюють шляхом обміну інформацією, ресурсами і артефактами. В даний час не існує офіційного стандарту моделювання екосистем програмного забезпечення. Основні наслідки відсутності стандарту - виробники програмного забезпечення мають проблеми виділення конкретних екосистем програмного забезпечення та їх використання.

У інженерних дисциплінах, об'єкти пізнання досліджуються за допомогою моделей. Існує чотири типи засобів моделювання екосистем програмного забезпечення: i* моделі (SD & SR), нормативні i* моделі, PDC (Product deployment Context), SSN (Software Supply Network).

Модель i* відображає соціальні аспекти екосистеми та видає пріорітет соціальним суб'єктам, у яких є цілі, переконання, здібності і зобов'язання. Аналіз фокусовано на тому, як добре цілі різних акторів досягаються в контексті відносин між людиною і системними учасниками, а також як при зміні конфігурації цих відносин можна допомогти учасникам досягти своїх стратегічних цілей. i* стимулювало значний інтерес до соціально-мотивованого підходу до моделювання і проектування, і призвело до низки цих розширень та адаптацій.

PDC забезпечує швидкий огляд архітектури та залежностей програмного продукту в своєму працюючому навколишньому середовищі. Деталі, представлені PDC, показують ієрархію між різними продуктами і компонентами та забезпечують їх почерговий перегляд з різних мережевих локацій.

SSN являє ряд пов'язаних між собою областей, таких як програмне забезпечення, апаратне забезпечення та обслуговування організацій, що співпрацюють для задоволення вимог ринку.

У доповіді було виконано моделювання екосистеми розробників програмних продуктів України.

Науковий керівник – М.О.Сидоров, д.т.н., проф.

УДК 004.413 (043.2)

Пашук А.В., Сидорова Н.М.
Національний авіаційний університет, Київ

ОНТОЛОГІЇ ГРУПОВОЇ ДИНАМІКИ

Процес розробки програмного забезпечення – це складна, багатоетапна діяльність, яку здійснюють розробники програмного забезпечення в складі команд. Тому процес розробки– це групова діяльність, а важливою її складовою є комунікації. Процес розробки передбачає залучення різних спеціалістів та їх тісну взаємодію, як між собою так і з замовником програмного забезпечення. Ефективність процесу розробки, його своєчасність та правильність залежить від коректно встановлених комунікацій.

Обмеженість комунікацій може привести до збільшення затрат часу на виконання операцій; відсутність чіткого розуміння цілей; погане надходження актуальної інформації; зростання витрат на реалізацію.

Завданнь які потрібно розв'язувати:

- Моделювання комунікацій в колективі, при якому забезпечується створення груп відповідно до поставлених задач та встановлення з'язків між ними;
- Оптимізація комунікацій між групами, забезпечення кращої продуктивності роботи;
- Вирішення проблемних ситуацій комунікацій у випадку реструктуризації груп чи зміни поставлених задач;
- Адаптація процесу розробки під нові вимоги.

Ці завдання можна розв'язати щляхом створення системи з представленням предметної області в формі онтологій.

Онтологія - це концептуальне представлення знань певної предметної області.

Цілі застосування онтологій в груповій динаміці є наступними: структурування інформаційної бази; формалізація опису процесів групової динаміки; автоматизація процесу аналізу; вирішення комунікаційних задач; проектування моделі комунікації; оптимізація комунікаційних процесів.

В доповіді наведено приклад побудови онтології групової динаміки при створенні програмних продуктів.

Науковий керівник – Є. М. Романов, к.т.н., доц.

УДК 629.735.33 (043.2)

Рибачук Я.А.
Національний авіаційний університет, Київ

МОДЕЛЬ ДОВГОІСНУЮЧОЇ ТРАНЗАКЦІЇ ПРИ ПРОЕКТУВАННІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розглянута модель довгоіснуючої транзакції (ДТ), яка базується на моделі Sagas – сукупності ДТ, що паралельно виконуються. Щоб полегшити проблеми керування ДТ було запропоновано в [1] поняття саги (англ. saga). ДТ це сага, якщо вона може бути записана у вигляді послідовності транзакцій, які можуть чергуватися з іншими ДТ.

Мета роботи – застосувати модель ДТ для підтримки процесів автоматизації глобальної розробки програмного забезпечення (ПЗ).

Процес створення ПЗ розглядається як виконання ДТ над інформаційною базою проектів (ІБП) від начала до завершення. При цьому ІБП розглядується як розподілена гіпертекстова база ресурсів проектів (програмна документація, тексти програм, графіки, то що). Система керування ДТ базується на сервіс орієнтованої архітектурі.

Головний сенс довгоіснуючої транзакції – виконувати все або нічого. Під час створення ПЗ, виконується безліч дій, які входять до ДТ, тобто або всі повинні бути виконані, зафіксовані, або повністю відмінена вся операція та відбутися повернення до попереднього стану ІБП.

Обробка даних, що поступають, приводить до великої кількості змін програмного забезпечення. Ці зміни потенційно можуть потерпіти невдачу, і чинники, які можуть впливати – досить велика кількість, тому система повинна в разі невдачі коректно повернути ІБП в стан на момент створення чергового контрольного збереження даних. Для цього створюється журнал транзакції.

Журнал транзакцій у поєднанні з сегментом відкату (область, в якій зберігається копія всіх змінних даних), що гарантує цілісність даних. В разі збою запускається процедура відновлення. При цьому аналізується наступне:

1. Якщо пошкоджений запис, то збій стався під час проставляння відмітки в журналі. Значить, нічого важливого не загубилося, ігноруємо цю помилку.
2. Якщо всі записи помічені як успішно виконані, то збій стався між транзакціями, тут також немає втрат.
3. Якщо в журналі є незавершена транзакція, то збій стався під час запису на диск. В цьому випадку ми відновлюємо стару версію даних з сегменту відкоту.

Науковий керівник – Оленін М.В., к.ф.-м.н., доц.