О. М. Бевз, В. М. Папінов, Ю. А. Скидан

ПРОЕКТУВАННЯ ПРОГРАМНИХ ЗАСОБІВ СИСТЕМ УПРАВЛІННЯ

Частина 1

Основи об`єктно-орієнтованого проектування

Міністерство освіти і науки України
Вінницький національний технічний університет

О. М. Бевз, В. М. Папінов, Ю. А. Скидан

ПРОЕКТУВАННЯ ПРОГРАМНИХ ЗАСОБІВ СИСТЕМ УПРАВЛІННЯ

Частина 1

Основи об`єктно-орієнтованого проектування

Навчальний посібник

Вінниця
ВНТУ
2010

УДК 681.3.06(075.8)
ББК 34.630.2
Б36

Рекомендовано до друку Вченою радою Вінницького національного технічного університету Міністерства освіти і науки України (протокол № 5 від 29.12.2008 р.)

Рецензенти:
І. І. Хаймзон, доктор технічних наук, професор
А. М. Пєтух, доктор технічних наук, професор
В. П. Посвятенко, кандидат технічних наук

Бевз, О. М.

Б36 Проектування програмних засобів систем управління. Частина 1. Основи об`єктно-орієнтованого проектування : навчальний посібник / О. М. Бевз, В. М. Папінов, Ю. А. Скидан. – Вінниця : ВНТУ, 2010. – 125 с.

В навчальному посібнику розглядаються сучасні об`єктно-орієнтовані методи програмного забезпечення систем управління.

Кожний розділ містить контрольні запитання.

Рекомендується для студентів, магістрів які здобувають освіту за спеціальністю ”Системи управління та автоматики”, а в подальшому за спеціальністю ” Системна інженерія ”.

УДК 681.3.06(075.8)
ББК 34.630.2

ЗМІСТ

ПЕРЕДМОВА

Проектування програмного забезпечення (ПЗ) – процес визначення архітектури, компонентів, інтерфейсів, інших характеристик системи й кінцевого результату. Проектування ПЗ як інженерний процес входить зараз як складова дисципліна загального інженерного процесу розроблення програм – *програмної інженерії(software engineering)* .

Термін програмна інженерія вперше вжито в 1968 році на науковій конференції НАТО, чим було засвідчено, що створення програмного забезпечення досягло такого ступеня розвитку, коли можна застосовувати інженерні технології. З того часу програмне забезпечення проникло в усі сфери людського буття, а його розроблення стала справді масовою діяльністю.

Інструменти підтримки розроблення програмних систем зробили величезний стрибок у своєму розвитку, і тепер звичайною практикою стали: створення програмних систем як композиції готових компонент, побудова інтерфейсів користувача шляхом маніпуляцій з мишкою, візуалізація процесів та продуктів розробки.

Водночас, залишаються актуальними питання організації розроблення великих систем, шляхів та засобів досягнення їхньої високої якості й відповідності до потреб замовника. Саме володіння методами розв'язання таких проблем дає надію на вдалу реалізацію великих програмних систем та їхню конкурентоспроможність.

Отже, важливо об'єднати досвід попередніх розробок як певну суму професійних знань, які могла б опановувати переважна більшість виконавців розробок, і тоді передовий досвід став би надбанням широкого кола професіоналів.

Саме таких зусиль тепер докладають професійні об'єднання та провідні виробники програмних систем для визначення ядра професійних знань, що становлять предмет програмної інженерії як фаху. Це ядро вони назвали ***Software Engineering Body of Knowledges (SWEBOK).*** Першим результатом їхньої діяльності є визначення головних розділів ядра знань та відповідних тематичних рубрик.

Нині проблеми і питання програмної інженерії, зокрема проектування ПЗ, висвітлюються в низці статей, в окремих параграфах і главах закордонних монографій. Проте монографії, яка охоплювала б усі основні питання цієї проблематики, в українській і російськомовній літературі немає. Це й спонукало нас до спроби заповнити цю прогалину і написати навчальний посібник.

До посібника включено сучасні методи проектування ПЗ, що використовуються зараз в програмній інженерії і пройшли вже тривалу практичну апробацію під час розроблення великих програмних систем. Водночас посібник є спробою цілісного викладення основних положень

процесів проектування ПЗ програмної інженерії. Його сплановано так, що кожний розділ відповідає послідовності процесів проектування сучасних програмних систем, починаючи від поставлення завдання і закінчуючи моделями програмної системи, які задовольняють замовника і можуть бути реалізованими на вибраній платформі.

Автори посібника виклали зміст процесів проектування ПЗ відповідно до напрацювань зі створення SWEBOK. Добираючи матеріал, ми керувалися прагненням висвітлити концепції та методи, перевірені практикою, обминаючи націлені на майбутнє теоретичні дослідження, які ще не підтвердили своєї придатності для широкого впровадження.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

Цей навчальний посібник є загальним введенням у сучасні інженерні методи проектування програмного забезпечення різних систем управління. Оскільки немає простих розв’язань для складних питань, тому, не шукаючи панацеї, ми зробили спробу розглянути широке коло питань, які виникають і потребують відповідей на шляху від розуміння замовником його потреб до створення готових до реалізації моделей ПЗ, що задовольняють згадані вище потреби.

Посібник має 7 розділів. До кожного розділу додаються контрольні запитання і завдання. Відповідаючи на запропоновані запитання та розв’язуючи завдання, студент зможе ще раз зосередити свою увагу на центральних аспектах кожного розділу і проконтролювати своє розуміння викладених знань. Нижче подається короткий зміст розділів.

Розділ 1. Програмна інженерія як фах.

Подається визначення програмної інженерії, її специфіка як інженерної діяльності зі створення програмних систем. Обговорюються питання становлення фаху, зміст його головних напрямів дій, зв’язок з іншими комп’ютерними дисциплінами.

Розділ 2. Базові поняття програмної інженерії.

Наводяться головні елементи еталонної моделі програмної інженерії та заснованих на них моделей життєвого циклу розроблення програмних систем.

Глава 3. Інженерія вимог.

Обґрунтовується інженерія вимог як визначальна стадія життєвого циклу, мета якої — сформулювати договір між замовником і розробником мовою, ступінь формалізації котрої не заважає досягненню домовленостей. Аналізуються й зіставляються кілька конструктивних підходів, перевірених широким використанням, які претендують на роль стандартів у майбутньому, але майже невідомі вітчизняним розробникам через брак доступних публікацій. Розглянуто об’єктно-орієнтовані методи аналізу

вимог та аналізу проблемних областей.

Розділ 4. Проектування програмних систем.

Розглядаються проблеми трансформації вимог замовника до розроблення в проектні рішення з визначення структури та особливостей функціонування майбутньої системи.

Розділ 5. Метод UML як потенційний стандарт засобів моделювання в програмній інженерії.

Розглядаються концепції та елементи сучасного методу, який набув широкого визнання та застосування. Мова методу призначена для специфікації, візуалізації, конструювання та документування артефактів програмних систем.

Розділ 6. Трансформація проекту в програму.

Подається загальна схема деталізації проекту під час створення вихідної програми.

Розділ 7. Повторне використання у програмній інженерії.

Висвітлено підходи до використання готових напрацювань (так зване повторне використання) як до такого, що є ключовим чинником підвищення якості програмного продукту та економії потрібних для цього ресурсів, а саме: часу та вартості розробки. Сформульовано основні поняття і означення; запропоновано класифікацію категорій повторно використовуваних компонент; подано підходи до створення та інформаційного забезпечення їх використання.

Додаток А. Глосарій (словник найбільш вживаних термінів).

Навчальний посібник призначений для студентів напряму “Автоматика та управління”, які навчаються за спеціальністю “Системна інженерія” і вивчають комп’ютерні дисципліни, пов’язані з розробленням програмних засобів та систем.

1 ПРОГРАМНА ІНЖЕНЕРІЯ ЯК ФАХ

Виробництво й використання комп’ютерних програм є тепер масовою діяльністю: як засвідчує статистика, розробленням програм зайнято майже сім мільйонів людей, а тих, хто активно використовує програмні системи у своїй діяльності за фахом, нараховують десятки мільйонів. Програмні системи набули статусу соціально значущого фактора, який впливає на безпеку та добробут суспільства.

За таких обставин світове суспільство прийшло до висновку, що технологія виробництва програм потребує свого оформлення як самостійний інженерний фах, який має забезпечити у світі відповідний кадровий потенціал для обсягу програмних розробок, що постійно зростає. За чотири десятиріччя досвіду з програмування створено передумови для такого оформлення, і тепер ми можемо спостерігати інтенсивний процес

визначення нового фаху, котрий названо програмною інженерією. Цей процес протікає у двох руслах – як широка дискусія у відповідних спеціальних журналах (наприклад, у джерелах [1,2]) та як цілеспрямована організаційна діяльність міжнародних професійних комп'ютерних об'єднань на чолі з відомими ACM та IEEE Computer Society, для концентрації зусиль яких створено спеціальний комітет. У його публікаціях [3] дефініція програмної інженерії визначається так: програмна інженерія – це система методів, засобів та дисципліни планування, розробки, експлуатації й супроводження програмного забезпечення, здатна до масового відтворення.

Як бачимо, до ключових процесів у сфері діяльності даного фаху віднесено планування та супроводження. При цьому перший визначається як аналіз цілей і завдань розробки, можливості її реалізації та потрібних *для* цього ресурсів, а другий (супроводження) трактується не стільки як усунення знайдених хиб, скільки як визначення й реалізація необхідних змін, зумовлених еволюцією потреб та умов діяльності користувачів. Адже один з авторитетів програмної інженерії М. Джексон [4] стверджує, що золотим правилом професії є таке: всяка закінчена програмна система одразу потребує змін.

Чому мова йде саме про інженерну діяльність? Згадаймо, як вона визначається у відомому загальному тлумачному словнику [5]:

– інженерія — це застосування наукових результатів, яке дозволяє мати користь від властивостей матеріалів та джерел енергії;

– діяльність із створення машин для надання корисних послуг. На відміну від науки, метою якої є здобуття знань, для інженерії знання є лише засобом отримання користі. Як казав ще один з визнаних авторитетів Ф. Брукс [6], вчений будує, щоб навчитися, інженер навчається, щоб будувати.

Розроблення програмних систем визначається, як бачимо, як інженерна діяльність. Однак слід навести її досить значні відмінності від традиційної інженерії:

– традиційні гілки інженерії мають високий ступінь спеціалізації (авіоніка, машинобудування, хімія тощо). У програмній інженерії спеціалізація є помітною тільки у досить вузьких застосуваннях, як-от транслятори, операційні системи і деякі інші;

– об'єкти традиційної інженерії добре визначені, і маніпуляції з ними відбуваються у вузькому контексті, коли проблеми ухвалення рішень стосуються окремих деталей, а не загальних питань. Принципи побудови автомобіля або літака апріорно визначені, їхні типові складові — також, напрацьовано колекцію типових проектних рішень та деталей, що відповідають типовим потребам замовників: гусениці для бездоріжжя, двері позаду для транспортування поранених, шипи на шинах для ожеледиці тощо; нова розробка потребує змін лише окремих з них. У

програмній інженерії подібної типізації немає;

– окремі проблеми традиційної інженерії та відповідні їм готові рішення добре класифіковані й каталогізовані. У програмній інженерії нова розробка виглядає як нова проблема, в якій досить важко розгледіти аналогії із системами, які було побудовано раніше, бо класифікації та каталогізації розв'язаних проблем практично немає.

Перелічені відмінності потребують значних зусиль для нівелювання їх, і саме тепер світова комп'ютерна спільнота визнала доцільність та вчасність таких зусиль. Це визнання матеріалізували міжнародні професійні об'єднання, створивши спеціальний комітет, метою якого є перетворення програмної інженерії на спеціальність, що визнається офіційно і має зафіксовані ознаки для розпізнавання її як такої.

Практика спеціалізації професійної діяльності, що склалася в цивілізованому світі, дозволяє вважати професію "зрілою", якщо для неї є:

– система початкового навчання за фахом;

– механізми розвитку вмінь та навичок персоналу, необхідні для практичної діяльності;

– сертифікація персоналу, організована в рамках професії;

– ліцензування фахівців, організоване під керівництвом органів влади (зокрема, для систем з підвищеним ризиком, як-от для АЕС та їм подібних);

– системи професійного вдосконалення кваліфікації персоналу та відстеження сучасного рівня знань і технологій за фахом, щоб уможливити для фахівців виживання за умов інтенсивного розвитку фаху;

– етичний кодекс фахівців;

– професійні об'єднання.

Для програмної інженерії є лише потужні професійні об'єднання, такі, як широковідомі серед професіоналів американське об'єднання комп'ютерних спеціалістів ACM (Association for computer machinery) і комп'ютерна спілка при Інституті інженерів з електроніки та електрики IEEE Computer Society, об'єднаними зусиллями яких у 1999 році було ухвалено етичний кодекс професіонала з програмної інженерії [7].

Для наявності усіх інших з перелічених ознак потрібно, щоб було зафіксовано суму знань, які становлять фахову кваліфікацію персоналу. Згадані вище професійні об'єднання та ряд потужних виробників програмного забезпечення стали як ініціаторами, так і організаторами діяльності з визначення суми професійних знань з програмної інженерії, для чого й було створено спеціальний комітет, який об'єднав широке коло провідних спеціалістів.

У 1999 році комітет напрацював перелік головних розділів, які, на думку членів комітету, є сумою необхідних знань з даної професії, для кожного з розділів подано перелік тем, що входять до його складу [3].

Таким чином, визначено два вищі рівні дерева знань.

Назвемо ці розділи:

– *аналіз вимог (requirement analysis)* до програмної системи, яку мають створити;

– детальний проект згаданої системи;

– кодування;

– тестування системи;

– процес супроводження програмного продукту;

– керування конфігурацією;

– забезпечення якості розробки;

– забезпечення відповідності розробки до вимог її замовників – *валідація (validation)* та забезпечення відповідності кодів до проекту – *верифікація (verification)*;

– процес удосконалення отриманого програмного продукту. Напрацьовано також перелік суміжних дисциплін, знання з яких необхідні для фахівців з програмної інженерії. До таких дисциплін, зокрема, належать:

– комп'ютерна наука;

– управління проектом;

– електрична інженерія;

– математика;

– телекомунікації та мережі;

– менеджмент;

– когнітивні науки.

Отже, зміст нової інженерної дисципліни – програмна інженерія можна вважати, певною мірою, визначеним, хоча деякі корективи цілком імовірні.

Контрольні запитання і завдання

1. У чому суть інженерної і наукової діяльності?

2. У чому специфіка програмної інженерії як інженерної діяльності?

3. Назвіть ознаки професії. Які з них властиві програмній інженерії?

4. Які з названих ознак вимагають наявності зафіксованої суми фахових знань, що становлять основи кваліфікації персоналу?

2 БАЗОВІ ПОНЯТТЯ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Еталонну модель програмної інженерії можна визначити як взаємодію трьох факторів:

– процесів;

– продуктів;

– ресурсів.

Кожна програмна система протягом свого існування проходить з певною послідовністю фази або стадії від задуму до його втілення в програми, експлуатацію та вилучення. Така послідовність фаз називається *життєвим циклом розробки (Software life cycle processes)*. На кожній фазі відбувається певна сукупність процесів, кожен з яких породжує певний продукт, використовуючи певні ресурси.

Усі продукти всіх процесів програмної інженерії являють собою певні описи — тексти вимог до розробки, погодження домовленостей, документацію, тексти програм, інструкції з експлуатації тощо.

Головними ресурсами програмної інженерії, які визначають ефективність її розробок, є час і вартість цих розробок.

Різновиди діяльності, котрі становлять процеси життєвого циклу програмної системи, зафіксовано в міжнародному стандарті ISO/IEC 12207 : 1995—0801 : Informational Technology – Software life cycle processes [1].

Згідно з наведеним стандартом, усі процеси поділено на три групи:

– головні процеси;

– допоміжні процеси;

– організаційні процеси.

До головних процесів віднесено такі:

– процес придбання, який ініціює життєвий цикл системи та визначає організацію-покупця автоматизованої системи, програмної системи або сервісу;

– процес розроблення, який означає дії організації — розробника програмного продукту;

– процес постачання, який означає дії під час передавання розробленого продукту покупцеві;

– процес експлуатації, який означає дії з обслуговування системи під час її використання — консультації користувачів, вивчення їхніх побажань тощо;

– процес супроводження, який означає дії з керування модифікаціями, підтримки актуального стану та функціональної придатності, інсталяцію та вилучення версій програмних систем у користувача.

У свою чергу, до процесу розроблення входять такі процеси:

– інженерія вимог до системи;

– проектування;

– кодування й тестування.

До допоміжних процесів віднесено ті, що так чи інакше забезпечують якість продукту. Терміном *якість продукту* позначено сукупність властивостей, які зумовлюють можливість задоволення потреб

замовника, котрий сформулював їх у формі вимог на розробку.

До організаційних процесів віднесено менеджмент розробки, створення структури організації, навчання персоналу, визначення відповідальності кожного з учасників процесів життєвого циклу розробки.

Стандарт, розглянутий нами вище, є головним чинником визначення змісту діяльності у сфері програмної інженерії, і всі знання, яких потребують професіонали з програмної інженерії, формулюються стосовно процесів, визначених стандартом ISO/IEC 12207:1995 – 0801: Informational Technology – Software life cycle processes.

Зупинимося докладніше на процесах розробки програмного забезпечення, які в сукупності мають забезпечити шлях від усвідомлення потреб замовника до передавання йому готового продукту. На цьому шляху виділяють низку характерних робіт.

Визначення вимог. Збір та аналіз вимог замовника виконавцем і подання їх у нотації, яка є зрозумілою як для замовника, так і для виконавця.

Проектування. Перетворення вимог до розробки в послідовність проектних рішень щодо способів реалізації вимог: формування загальної архітектури програмної системи та принципів її прив'язки до конкретного середовища функціонування; визначення детального складу модулів кожної з архітектурних компонент.

Реалізація. Перетворення проектних рішень на програмну систему, яка реалізує такі рішення.

Тестування. Перевірка кожного з модулів та способів їхньої інтеграції; тестування програмного продукту в цілому (так звана верифікація); тестування відповідності функцій працюючої програмної системи *вимогам (requirements)*, поставленим до неї замовником (так звана валідація).

Експлуатація та супроводження готової програмної системи.

За десятиріччя досвіду з побудови програмних систем напрацьовано низку типових схем послідовності наведених вище робіт. Такі схеми назвали моделями життєвого циклу. Історично першою застосовувалася так звана *водоспадна* або *каскадна модель,* за якою вважалося, що кожна з робіт виконується один раз і в тому порядку, в якому їх перелічено вище. Інакше кажучи, робилося припущення, що кожну з робіт буде виконано настільки ретельно, що після її завершення й переходу до наступної роботи повернення до попередньої не потрібно. На рис. 2.1 показано послідовність робіт за *водоспадною (каскадною) моделлю.* Повернення до початкової стадії робіт передбачається, як бачимо, лише як результат супроводження.

Цінність такої моделі полягає в тому, що вперше було зафіксовано послідовність процесів розроблення та стадії готовності програмного продукту, а недоліком є те, що в її концепцію покладено модель фабрики, коли продукт проходить стадії від задуму до виробництва, після чого

передається замовникові як готовий виріб, зміну якого непередбачено, хоча й можлива заміна на інший подібний продукт у разі рекламації.

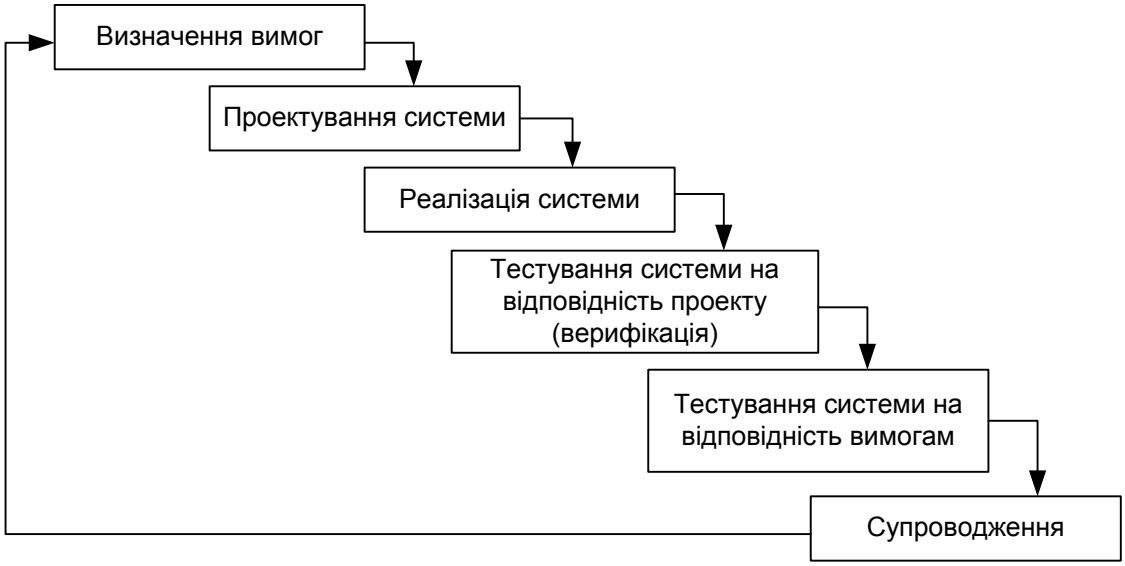

Рисунок 2.1 – Водоспадна модель життєвого циклу програмних систем

Для програмного продукту така модель не підходить з кількох причин. По-перше, висловлення вимог замовником – це суб'єктивний, неформалізований процес, який, як засвідчує багаторічний досвід, може багаторазово уточнюватися протягом розроблення і навіть після її завершення та випробовування, якщо з'ясується, що замовник "хотів зовсім інше". По-друге, змінюються обставини та умови використання системи, тому загальновизнаним законом програмної інженерії є закон *еволюції,* котрий можна сформулювати так: кожна діюча програмна система з часом потребує змін або перестає використовуватися.

Зважаючи на необхідність еволюції, водоспадну модель можна розглядати як модель життєвого циклу лише для першої версії розробки. Враховуючи, що на кожній стадії робіт може виникнути потреба змін, і цю потребу має бути задоволено таким чином, щоб: документація, яка є продуктом кожної стадії (опис вимог, опис проекту тощо), відповідала дійсному стану розробки після внесення змін, було створено так звану спіральну модель розвитку робіт, відміною якої є можливість багаторазового повернення до стадії формулювання вимог до розробки з будь-якої стадії робіт, якщо виявиться необхідність внесення змін.

На рис. 2.2 подано зображення спіральної моделі розробки, в якій кожний виток спіралі відповідає одній з версій розробки. На кожній стадії розроблення аналізується потреба змін, а внесення змін на будь-якій стадії обов'язково починається з внесення змін до попередньо зафіксованих вимог.

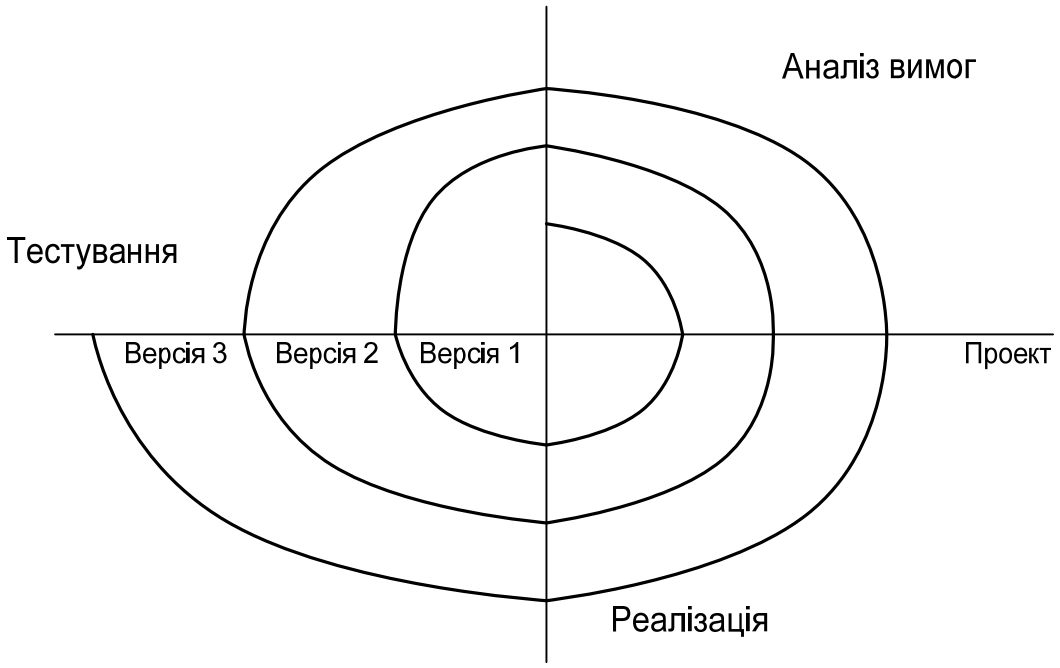

Рисунок 2.2 – Спіральна модель життєвого циклу розробки

Контрольні запитання і завдання

1. Які три чинники визначають еталонну модель програмної інженерії?
2. Який вигляд мають продукти програмної інженерії?
3. Назвіть головні ресурси програмної інженерії, що визначають ефективність розробок програмного забезпечення.
4. Сформулюйте визначення життєвого циклу розробки програмного забезпечення.
5. Назвіть три основні групи процесів життєвого циклу.
6. Перелічіть процеси кожної з груп.
7. Який міжнародний стандарт визначає перелік і зміст процесів життєвого циклу програмного продукту?
8. Чи всі процеси, зазначені в стандарті, має бути виконано в кожній розробці програмного забезпечення або чи надає стандарт такі можливості, які можуть бути актуальними для конкретного випадку?
9. Назвіть етапи процесу розроблення програмного забезпечення.
10. Основні моделі життєвого циклу розробки програмного забезпечення і їх відмінності.

3 ІНЖЕНЕРІЯ ВИМОГ

3.1 Інженерія вимог як процес

Кожна система — це певний перетворювач, чию поведінку і властивості якої ми будуємо в процесі створення програмної системи, і цю поведінку та властивості ми обираємо так, щоб вирішити нашу проблему.

Програмна система — це певна машина, котра вводиться у світ для того, щоб впливати на нього. Частини світу, які впливають на машину або піддаються її впливу, становлять так званий *домен предметної галузі (data domain)* або *домен застосування(application domain)*. Опис цього впливу дає відповідь на запитання: "Що робить система?" і визначає вимоги до системи у формі угод між замовником і виконавцем. Як вона це робить, визначає опис машини.

Будемо називати *вимогами* до програмної системи властивості, які мають бути в системи, якщо вона адекватно виконує свої функції. Прикладами таких функцій можуть бути: автоматизація притаманних персоналу обов'язків, забезпечення керівництва інформацією, потрібною для ухвалення рішень, керування фізичними пристроями виробничого процесу *тощо*. Тобто, програмна система може моделювати досить складну діяльність людей та організацій, їхньої взаємодії з фізичними пристроями тощо. Відповідно до цього, вимоги до програмної системи мають відображати всі труднощі такої діяльності.

В сучасних інформаційних технологіях фаза життєвого циклу, на якій фіксуються вимоги на розробку програмного забезпечення, є визначальною для його якості, термінів та вартості робіт. Саме на цій фазі мають бути зафіксовані реальні потреби користувачів, що стосуються функціональних, операційних та сервісних можливостей, які береться реалізувати розробник. Таким чином, на цій фазі домовляються замовник і виконавець, що визначає подальші дії виконавця.

Ціна помилок та нечітких неоднозначних формулювань на цій фазі дуже висока, бо час і засоби витрачаються на непотрібну замовникові програму, оскільки він мав на увазі зовсім інше, але не зумів сформулювати свої потреби. Внесення необхідних коректив при цьому може вимагати значних переробок, а інколи й повного перепроектування і, відповідно, перепрограмування. Тим часом статистика показує, що відсоток помилок у постановці завдань перевищує відсоток помилок кодування, і це є наслідком суб'єктивного характеру процесу формулювання вимог та майже повної відсутності засобів його формалізації. Так, згідно зі статистикою, в Америці витрачається щорічно до 82 млрд. доларів на проекти, визнані після реалізації такими, що не відповідають вимогам замовника, інакше кажучи, непотрібними.

Діючими особами процесу формулювання вимог є:

– носії інтересів замовників (досить часто замовника репрезентують кілька професійних груп, потреби яких можуть мати не лише відмінності, а й навіть суперечності);

– оператори, котрі здійснюють обслуговування під час функціонування системи;

– розробники системи.

До процесу формулювання вимог входять кілька підпроцесів.

Складання вимог. Джерелами відомостей про вимоги можуть бути:

– мета та завдання системи, які формулює замовник. Слід зазначити, що при безумовній пріоритетності думки замовника треба мати на увазі небезпеку неоднозначного розуміння цих формулювань замовником і розробником, а також властивість людини замовчувати багато визначальних подробиць, що є не ознакою злої волі, а лише наслідком або забудькуватості, або впевненості, що це "загальновідомо". Тому формулювання замовника підлягають глибокому осмисленню з боку виконавця;

– діюча система або колектив, який виконує її функції. Досить часто система, котру замовляють, має замінити собою попередню систему, що вже не задовольняє замовника, або певні функції діючого персоналу. Вивчення й фіксація наявних функціональних можливостей створює базу, розширення якої дозволить сформулювати вимоги до потрібної системи, в яких урахується набутий досвід замовника. Для цього джерела також є певна небезпека перенесення вад організації попередньої системи в нову. Наприклад, розподіл обов'язків серед персоналу певного відділу склався історично впродовж послідовного розширення кадрів і є недоцільним відносно функцій відділу. Тому, вивчаючи діючу систему, треба вміло відокремити потреби проблеми, яку розв'язує система, від закладених у стару систему невдалих організаційних рішень. З огляду на це, процес урахування діючої системи під час складання вимог до нової доцільно провести за три кроки:

а) за першим кроком вивчаємо фізичну структуру діючої системи (незалежно від того, автоматизована вона чи "людська");

б) за другим кроком робимо логічне узагальнення виявлених на першому етапі функцій та особливостей, виділяючи ті, що відображають

поставлену перед розробкою проблему, на відміну від закладених у стару систему окремих шляхів розв'язання проблеми (інколи не зовсім вдалих);

в) за третім кроком визначаємо логічне розширення функцій, виявлених на другому кроці, яке відповідає потребам нової системи як розвитку існуючої в заданому напрямі; визначені функції формулюємо як вимоги до нової системи;

– загальні знання щодо проблемної галузі замовника. Він формулює

і розуміє свої проблеми в термінах понять певної проблемної галузі і, як згадувалося вище, замовчує подробиці, котрі належать саме до загальновідомих знань (але, на жаль, тільки серед професіоналів відповідної галузі);

– відомчі стандарти замовника, котрі стосуються організаційних вимог, середовища функціонування майбутньої системи, її виконавських та ресурсних можливостей.

Методами складання вимог найчастіше є:

– інтерв'ю з носіями інтересів замовника та операторами;

– спостереження за роботою діючої системи з метою відокремлення її органічних властивостей від тих, що зумовлені структурою кадрів;

– фіксація сценаріїв усіх можливих випадків використання системи, виконуваних при цьому системою функцій, ролей осіб, котрі запускають ці сценарії або взаємодіють з системою під час її функціонування.

Продуктом процесу складання вимог є неформалізований опис цих вимог. Такий опис є фактичним контрактом на розробку між замовником і виконавцем. Обидві зазначені сторони мають розуміти його зміст, оскільки це розуміння гарантує, що система, в розробку якої буде вкладено працю виконавця, задовольнить замовника. Тому нотацію згаданого вище опису має бути орієнтовано на людину. Водночас цей опис є входом для наступного процесу інженерії вимог – аналізу вимог. Виконавцем цього процесу є розробник, а завданням є подальше уточнення й формалізація вимог та їхнє документування в нотації, що є однозначно зрозумілою колективу розробників для подальшого проектування, реалізації, тестування, документування програмного продукту та інших необхідних процесів життєвого циклу розробки (див. п. 2).

Аналіз вимог. Першим кроком аналізу має бути класифікація вимог. Множину зібраних вимог можна розподілити між двома головними категоріями:

– ті, що відображають можливості, які повинна забезпечити система, назвали *функціональними вимогами (functional requirement);*

– ті, що відображають обмеження, пов'язані з функціонуванням системи, назвали *нефункціональними вимогами (notfunctional requirement).*

Перша з наведених категорій дає відповідь на запитання: "Що робить система?", а друга визначає характеристики її роботи, наприклад, що ймовірність збою системи протягом години не може перевищувати однієї мільйонної.

Нефункціональні вимоги можуть виступати як окремий чисельний показник, наприклад, час чекання відповіді абонента не може перевищувати півсекунди. Інколи вони можуть мати комплексний характер і потребувати для свого втілення сукупності деталізованих властивостей, наприклад, "підвищити кількість обслуговуваних клієнтів удвічі".

Є кілька класів нефункціональних вимог, суттєвих для більшості програмних систем, які виражають обмеження, актуальні для багатьох проблемних галузей. Серед них назвемо такі:

– вимоги конфіденціальності;

– відмовостійкість;

– число клієнтів, котрі одночасно мають доступ до системи;

– вимоги безпеки;

– час чекання відповіді на звернення до системи;

– виконавські якості системи (обмеження щодо ресурсів пам'яті, швидкість реакції на звернення до системи тощо).

Для більшості названих класів може бути зафіксовано спектр характерних понять, які позначаються терміном **дескриптор** і які застосовуються для розкриття їхньої суті. Склад дескрипторів закріплено у відповідних міжнародних та відомчих стандартах, що дозволяє уникнути неоднозначності тлумачення зібраних вимог.

Функціональні вимоги пов'язані із семантичними особливостями проблемної галузі, в межах якої планується розробка. Проблема термінологічних розбіжностей для них є досить впливовим фактором ускладнення. На жаль, робляться спроби розв'язання її шляхом стандартизації термінології лише для кількох проблемних галузей, для яких властивий інтенсивний розвиток комп'ютерних систем, наприклад, для авіоніки та медицини. Але можна засвідчити наявність стійкої тенденції до створення стандартизованого понятійного базису більшості проблемних галузей, які набувають певного досвіду комп'ютеризації.

Наступним кроком аналізу вимог є встановлення їхньої пріоритетності, бо, як було вказано вище, вимоги, висунуті різними носіями інтересів у системі, можуть конфліктувати між собою. Крім того, кожна з вимог потребує для свого втілення певних ресурсів, надання яких може залежати також від визначеного для неї пріоритету.

Визначення впливу вимог на потреби в ресурсах є також кроком процесу аналізу вимог.

Ще одним із важливих завдань аналізу є передбачення можливих змін у зібраних вимогах і забезпечення можливостей внесення передбачуваних змін без суттєвого перегляду всієї системи. Такі можливості мають забезпечити живучість системи та її здатність до адаптації.

Нарешті, в процесі аналізу вимог має бути перевірено правдивість та відповідність їх інтересам замовника, висловленим усіма носіями інтересів цього замовника.

Продуктом процесу аналізу є побудована модель проблеми, орієнтована на її розуміння, якого має досягнути виконавець до початку проектування системи.

3.2 Концептуальне моделювання проблеми

3.2.1 Мета

Процес побудови моделі проблеми, орієнтованої на її розуміння людиною, назвемо *концептуальним моделюванням (conceptual modeling).* Кожна проблемна галузь – назвемо її доменом – має властиву їй систему понять, знайому тільки відповідним професіоналам, свою систему "замовчування" того, що має вважатися "загальновідомим" у рамках свого домену, свої характерні властивості (атрибути), відношення та правила поведінки. Однак роль концептуальної моделі полягає в тому, щоб бути посередником між професіоналами, котрі належать до різних доменів, наприклад, як програмісти та фахівці з бухгалтерії. Ступінь формалізації цієї моделі має бути достатнім, щоб забезпечувати точність та однозначність трактування носіями інтересів у розробці, і водночас не надмірним, щоб бути доступним для розуміння не лише математикам за фахом, і спроможним відобразити деталі фахових проблем багатьох верств спеціалістів.

Враховуючи складність та розмаїття завдань, які вирішуються за допомогою програмних систем, винайти єдину модель для всіх поки що не вдалося. Але є кілька пропозицій нотацій для моделей, що досить адекватно відображають окремі аспекти проблем, а в своїй комбінації охоплюють проблеми з достатньою повнотою. Розглянемо ці пропозиції.

3.2.2 Онтологія домену

Найпершою з моделей домену може вважатися його понятійна база, тобто система понять, за допомогою якої формулюються всі аспекти проблеми. Понятійна база визначає не лише термінологію, якою мають користуватися носії інтересів, котрі беруть участь у процесі аналізу вимог, а також і суттєві відносини між поняттями та їхньою інтерпретацією.

Сукупність термінології, понять, характерних для них відносин і парадигми їхньої інтерпретації в межах домену прийнято називати онтологією домену.

Зміст поняття може бути охарактеризовано сукупністю спільних істотних ознак тих явищ та предметів, які позначаються назвою поняття. Сукупність явищ, охоплених поняттям, називається його обсягом. У ментальній діяльності людини застосовуються відношення, які дозволяють будувати похідні поняття і встановлювати між ними зв’язки. Серед таких відношень назвемо найпоширеніші:

– узагальнення – це звуження істотних ознак поняття, при цьому розширюється коло охоплених поняттям об’єктів, тобто його обсяг. Приклади: свійська тварина є узагальненням понять свиня, корова та

багатьох інших;

– конкретизація – це додавання істотних ознак, завдяки чому зміст поняття розширюється, а обсяг поняття звужується. Наприклад, студент консерваторії є конкретизацією загального поняття студент;

– *агрегація (aggregation)* – це об'єднання низки понять у нове поняття, істотні ознаки нового поняття при цьому можуть бути або сумою ознак компонентів або суттєво новими. Прикладом першого типу є пральна машина з центрифугою, другого – автомобіль як композиція колес, двигуна, корпусу, керма тощо. Відношення агрегації часто має назву відношення частка – ціле;

– *асоціація (association)* – це найбільш загальне відношення, що утверджує наявність зв'язку між поняттями, не уточнюючи залежності між їхнім змістом та обсягами.

Для окремих доменів можуть використовуватися специфічні для них відношення. Наприклад, каталізатор – у хімії, нащадок-предок – у генеалогії, рід-вид – у біології тощо. Фіксація в онтології відношень між поняттями структурує понятійний базис домену проблемної галузі.

Онтологія дозволяє втримувати користувача в максимально можливому просторі визначених наперед можливостей, зміст яких зафіксовано і він зрозумілий як розробникові, так і замовникові. Очевидно, що перебування в такому просторі гарантує обидві згадані вище сторони договору (який матеріалізовано у формі вимог) від взаємних непорозумінь. Очевидно також, що для відносно нових і достатньо складних галузей застосування комп'ютерних систем повний спектр можливостей передбачити важко, тому важливу роль відіграє аспект внесення змін у вимоги.

Для подання онтологій використовуються спеціальні нотації. Наприклад, діаграми сутність–зв'язок мають форму графів, у вузлах яких лежать поняття, а дуги відповідають відношенням між ними. Слід зазначити, що у зв'язку з інформатизацією життя суспільства поширюється тенденція створення і стандартизації онтологій для окремих проблемних галузей, що суттєво спрощує формулювання вимог у таких галузях.

У розробку галузевих онтологій вкладається чимало інвестицій, навіть побудовано бібліотеки фахових онтологій [3], та, на жаль, не в Україні, де така робота ще й не починалася. Треба сказати, що побудова й супроводження національних фахових онтологій та гармонізація їх із закордонними зразками принесе велику віддачу від вкладених інвестицій, дозволить занотувати актуальний стан справ у відповідній галузі, створити середовище взаєморозуміння й обміну думками вітчизняних професіоналів як між собою, так і з міжнародною спільнотою відповідних спеціалістів.

У значущих для суспільства професіях така робота лежить у руслі помітної тенденції фіксації необхідної суми професійних знань, що

дозволяє проводити акредитацію відповідних навчальних програм навчальних закладів та сертифікацію спеціалістів як у масштабі окремої держави, так і в міжнародному масштабі. Отже, усе зазначене засвідчує, що розгортання робіт із створення національних онтологій уже на часі і є кроком до інтеграції наших фахівців у міжнародний контекст.

3.2.3 Моделі динамічних явищ домену

Розглянуті в п. 3.2.2 онтології відображають статичні властивості явищ домену, які не враховують їхніх змін за часом. Але більшість проблем, котрі вирішують за допомогою комп'ютерів, мають справу з динамічними явищами, для яких відстеження їхньої поведінки з перебігом часу є суттєвим аспектом вимог до системи.

Домени з такими властивостями назвали динамічними доменами.

Для динамічних доменів суттєвими поняттями є:

– *стан (state)* (домену, системи, об'єкта тощо) – фіксація певних властивостей на певний момент або інтервал часу;

– *інтервал стабільності (stability interval)* – інтервал часу, протягом якого не змінюється стан;

– *подія* – явище, що провокує зміну певного стану як перехід до іншого стану (ми розглядаємо лише дискретні процеси в доменах).

Поведінка домену в часі розглядається як прогрес від стану до стану. Серед динамічних доменів визначаються такі різновиди:

– *інертні домени (inertial domain),* зміна станів яких ніколи не відбувається з ініціативи об'єктів домену, а реалізується під керуванням зовнішніх агентів. Наприклад, оброблення текстів реалізується як результат виконання послідовних команд оператора;

– *реактивні домени (reagent domain),* зміна стану в яких є відповіддю на певну зовнішню подію. Наприклад, автомат, який видає цукерку у відповідь на монетку, кинуту в щілину монетоприймача;

– *активні домени (active domain),* які можуть переходити від стану до стану без зовнішніх стимулів, як це робиться з людиною або атмосферою.

Визначаючи вимоги до системи, важливо одразу вирішити чи планується вона як інертна, реактивна, чи як активна. Наприклад, для медичної системи принципово визначити чи планується вона як модель пацієнта, чи як реєстратор його стану.

Найпоширенішою моделлю поведінки динамічних явищ є так звана модель переходів у стани, яка базується на моделі скінченного автомата. Для кожного явища або об'єкта, котрий має динамічну поведінку, фіксуються стани, в яких він може перебувати. Дозволяється мати кінцеву множину станів, при чому перехід з одного стану до іншого відбувається дискретно і зумовлюється тим, що відбулася певна подія. Модель

поведінки визначає стани, властиві домену, події, які впливають на зміну стану в домені, та послідовність зміни станів залежно від послідовності подій, що відбуваються. Наприклад:

– якщо поїзд має стан "стоїть на станції", і диспетчер станції дав сигнал відправки, тобто відбулася подія "сигнал відправки", поїзд переходить у стан "початок руху від станції";

– якщо хворий має стан "приймання процедури електрофорезу", і відбулася подія "дзвінок годинника задзвенів", медик має перевести хворого у стан "процедуру прийняв".

Якщо змінюється стан якогось об’єкта, то це найчастіше супроводжується певною послідовністю дій, які треба при цьому виконати. Тому модель поведінки зазвичай має у своєму складі визначення сукупностей дій, котрі потрібно виконувати при переході до відповідних станів. При викладенні конкретних методів аналізу вимог буде наведено приклади відповідних нотацій для подання моделі переходів у стани.

3.2.4 Модель алгоритмів

Моделі алгоритмів, які застосовуються для формулювання вимог, мають визначити ті дії або процеси, котрі супроводжують переходи з одного стану в інший.

Зазвичай такі моделі визначають потоки даних, які передаються від стану до стану чи від об’єкта до об’єкта, та потоки управління між ними.

3.3 Об’єктно-орієнтована інженерія вимог

Кожна з наведених у п. 3.2 моделей так чи інакше має на меті перебороти складність вирішуваної проблеми шляхом її декомпозиції на окремі простіші та зрозуміліші складові. Досі ми не обговорювали, що таке складові. Сутностями, для яких визначаються стани, можуть бути фізичні об’єкти (людина, двигун), явища природи (повінь, атмосфера), політичні явища (вибори, мітинг), функціональні блоки системи керування (відділ кадрів, відділ збуту) тощо. Між тим, від того, яким чином ми будемо структурувати проблему, від нашого бачення її складових і взаємодії (інтерфейсів) цих складових залежить *зрозумілість (intelligibility)* та "прозорість" опису тих вимог, які будуть результатом процесу аналізу, його повнота й точність.

Типи складових компонент та правила їхньої композиції (інтерфейси) визначаються в програмній інженерії як *архітектура системи (system architecture)*. Модель декомпозиції проблеми, яка визначає архітектуру, називається парадигмою програмування. Відомими парадигмами, поширеними в програмній інженерії, є дві: модель функції-дані та об’єктна.

Модель функції-дані (data-function model) є історично першою. Згідно з нею, проблема дєкомпозується на послідовність функцій та даних, які обробляються за допомогою цих функцій. Тобто, елементами композиції є дані та функції над ними. Подання перших і других має бути узгоджено між ними. Якщо змінюються якісь дані, треба переглянути всі функції, які їх обробляють, і визначити, чи не потребують деякі з них змін також. Якщо змінюється функція, треба переглянути всі дані, *які вона* обробляє чи має своїм результатом, з метою пошуку тих, котрі залежать від внесених змін. Можна зробити висновок, що за такою парадигмою внесення локальних змін до постановки проблеми потребує ревізії всіх даних та всіх функцій, щоб бути певним, що вони не зазнали впливу внесених змін.

Парадигма об'єктно-орієнтованого підходу до розроблення програмних систем дозволяє уникнути зазначеного вище недоліку. Згідно з її концепцією загальне бачення проблем пропонується як таке, що узгоджується з викладеними нижче постулатами:

– світ складають об'єкти, які взаємодіють між собою;

– кожний об'єкт має певний набір властивостей або атрибутів (аналог суттєвих ознак поняття);

– атрибут визначається своїм іменем та значеннями, які він може приймати;

– об'єкти можуть бути у відношеннях одне з одним;

– значення атрибутів та відношення можуть із часом змінюватись;

– сукупність значень атрибутів конкретного об'єкта в певний момент часу визначає його стан;

– сукупність станів об'єктів визначає стан світу;

– світ та його об'єкти можуть перебувати в різних станах;

– у певні інтервали часу можуть виникати якісь події;

– події можуть спричиняти інші події або зміни станів;

– протягом часу кожний об'єкт може брати участь у певних процесах, які зводяться до виконання послідовності дій, різновидами котрих є переходи з одного свого стану в інший під впливом відповідних подій, збудження певних подій чи посилання певних повідомлень до інших об'єктів;

– дії, які можуть виконувати об'єкти, називають операціями об'єктів (як синоніми використовують також терміни методи об'єкта та функції об'єкта*);*

– можливі сукупності дій об'єкта називають його поведінкою;

– об'єкти взаємодіють шляхом обміну повідомленнями;

– об'єкти можуть складатися із частин.

Об'єкт – це певна абстракція даних та поведінки: множина екземплярів із спільним складом атрибутів та спільною поведінкою

становить клас об’єктів. Визначення класу проведено за так званим принципом приховування інформації, котрий можна сформулювати так: повідомляйте користувачу лише ті відомості, які йому потрібні для того, щоб скористатися вашими напрацюваннями, все інше приховуйте. Такий принцип має низку переваг, серед яких суттєві такі:

– користувача позбавлено необхідності знати зайве, тобто непотрібне;

– того, про що йому не повідомили, він не може пошкодити, тобто здійснено захист від його неправомірних дій як навмисних, так і випадкових;

– усе, про що не знає користувач, можна змінювати, і це не матиме на нього ніякого впливу.

Визначення об’єктів проведено згідно з наведеним принципом і складається воно з двох частин – видимої та невидимої. Перша з них містить усі відомості, які потрібні для того, щоб взаємодіяти з об’єктом, і називається інтерфейсом об’єкта, а друга містить подробиці його внутрішньої будови, і вони сховані або, як кажуть, інкапсульовані (тобто перебувають немовби в капсулі). Так, наприклад, якщо нашим об’єктом є прилад, котрий реєструє показники температури, то до видимої частини його визначення відносимо операцію показання актуального значення температури. Якщо ж до того бажано керувати гранично дозволеними діапазонами температур, як у тепличному господарстві, то ці діапазони мають бути визначені як видимі, тобто віднесено до інтерфейсу об’єкта. Іншим прикладом об’єкта може бути банк, зовнішня поведінка котрого виглядає як можливість виконувати "видимі" операції відкриття й закриття рахунка клієнта, поповнення чи зменшення рахунка клієнта, тоді як усі внутрішні дії служб банку з обслуговування рахунка, що забезпечують виконання "видимих" операцій, а також усі реєстри та інші внутрішні дані банку щодо клієнтів і наявності готівки в касирів інкапсульовано, тобто клієнтові про них знати непотрібно і неможливо.

Ще одним важливим засобом визначення об’єктів є так зване успадкування. Кажуть, що один клас об’єктів успадковує інший, якщо він повністю містить усі атрибути й поведінку успадкованого класу, але має ще додаткові атрибути та (або) поведінку. Клас, який успадковують, називається суперкласом, а клас, що успадковує, називається *підкласом.* Спадковість явним способом фіксує спільні та розбіжні риси об’єктів і дозволяє явно виділити складові компоненти проблеми, які можна використати в кількох випадках шляхом побудови для них декількох класів-спадкоємців. Класи можуть утворювати ієрархії спадкоємців довільної глибини, в котрих кожний відповідає певному рівню абстракції і є узагальненням класу-спадкоємця і конкретизацією класу, який успадковує сам. Наприклад, клас числа може мати спадкоємців – підкласи цілі числа, комплексні числа, дійсні числа. Усі наведені підкласи

успадковують операції суперкласу (відомі арифметичні операції складання та віднімання), але кожний з підкласів визначає свої особливості виконання згаданих операцій.

Відзначимо одну з найважливіших властивостей операцій об'єкта, яку названо поліморфізмом операцій. Сутність її полягає в тому, що об'єкт, котрий відправляє повідомлення, не може знати класу того об'єкта, який отримає відіслане повідомлення. Інтерпретація ж отриманого повідомлення визначається класом того, хто його одержить. Наприклад, повідомлення об'єктові A виконати операцію додавання до B буде трактуватись як додавання матриць, якщо класи об'єктів A та B визначено як матриці або як додавання цілих, якщо A та B належать до класу цілих.

Завдяки багатим можливостям абстракції даних і поведінки, інкапсуляції та можливостям успадкування об'єктно-орієнтований підхід до розроблення програм набув великої популярності і фактично витіснив усі інші. У п. 2 ми вказували, що модель діяльності з програмної інженерії визначається за допомогою трьох факторів – процесів, продуктів та ресурсів; послідовність процесів визначає стадії життєвого циклу розроблення програмних систем, і продуктами кожного з процесів є описи, які відповідають моделі певної стадії осмислення проблеми. За об'єктно-орієнтованим підходом усі зазначені вище моделі мають визначатися як *взаємодія певних об'єктів (interaction object)*, при цьому в моделі вимог фігурують об'єкти, взаємодія котрих визначає проблему, яку маємо розв'язувати за допомогою програмної системи, а в подальших моделях (у моделі проекту, моделях реалізації та тестування) йдеться про об'єкти, взаємодія яких визначає розв'язання тієї проблеми (моделі проекту та реалізації) або перевірку достовірності того розв'язання (модель тестування).

Відповідно до зазначеного вище, у світі запропоновано чимало методів об'єктно-орієнтованого аналізу вимог, об'єктно-орієнтованого проектування програм, об'єктно-орієнтованого програмування (згадаймо широко поширений C++). Але найбільшу цінність серед них мають ті, які узгоджені між собою.

Якщо вдається встановити відповідність між об'єктами моделей продуктів різних стадій (процесів) життєвого циклу розробки, кажуть, що моделі дозволяють трасування вимог. Трасування вимог – це можливість простежити послідовну трансформацію об'єктів вимог, зрозумілих замовникові, у відповідні компоненти продуктів послідовних стадій розробки, закінчуючи діючою програмною системою. Можливість трасування полегшує контроль за здійсненими трансформаціями та за внесенням змін протягом усього циклу розробки синхронно в усі напрацьовані продукти різних стадій розробки до її завершення, що відповідає спіральній моделі життєвого циклу (див. п. 2).

За парадигмою об'єктно-орієнтованого підходу концептуальне

моделювання проблеми (дивись п. 3.2) відбувається в термінах взаємодії об’єктів:

– онтологія домену визначає склад об’єктів домену, їхніх атрибутів та взаємовідношень, а також послуг (операцій);

– модель поведінки визначає можливі стани об’єктів, інциденти, які ініціюють переходи з одного стану в інший, повідомлення, які об’єкти надсилають одне одному;

– модель процесів визначає дії, які виконують об’єкти.

Всі успішні пропозиції об’єктних методів, а їх напрацьовано вже чимало, мають у своєму складі зазначені вище моделі, хоча вони й відрізняються своїми нотаціями та деякими іншими деталями. Одразу зазначимо, що жодна з них ще не набула загального визнання, при тому що багато з них мають досить широке коло прихильників і позитивний досвід застосування. Нижче ми зупинимося більш детально на трьох методах.

Метод С. Шлеєр та С. Меллора (див. п. 3.2.2) ми обрали переважно тому, що читач може з вітчизняної публікації [2] отримати досить повне уявлення про його можливості, які є, певною мірою, типовими.

Метод сценаріїв використання майбутньої системи, або сценарний підхід (див. п. 3.2.3), є, на погляд авторів, єдиним методом, який вказує шлях до виявлення об’єктів, суттєвих для домену проблемної галузі. Справді, всі методи декларують – як перший крок – виявлення об’єктів і попереджають, що вдалий склад об’єктів зумовить зрозумілість і точність вимог, але тільки сценарний підхід дає рекомендації щодо того, з чого починати пошук об’єктів.

Метод UML (Unified Modelling Language) є узагальненням цих двох методів та кількох інших і тепер претендує на те, щоб стати міжнародним стандартом методу аналізу вимог та проектування програмних систем.

3.4 Метод інженерії вимог С. Шлеєр та С. Меллора

Програмна система розглядається як сукупність визначеного ряду доменів проблемних галузей, кожний з яких **є** окремим світом, населеним своїми об’єктами, і котрий аналізується незалежно від інших. Продуктом аналізу домену є три складові, які будуються, відповідно, за три етапи, а саме:

– онтологія домену, яку автори даного методу називають інформаційною моделлю системи або інформаційною моделлю домену;

– модель станів об’єктів, визначених у складі інформаційної моделі (або онтології);

– модель процесів, які супроводжують переходи з одного стану в інший.

3.4.1 Інформаційна модель або онтологія домену. Пошук об’єктів

Як ми вже зазначили в п. 3.2.2, завданням першого етапу є виявлення суттєвих об’єктів і встановлення зв’язків (відношень) між ними. Для цього ми маємо, по-перше, знайти такі об’єкти, а по-друге, дати їм унікальні та значущі назви. Ці дії суто суб’єктивні, єдина рекомендація щодо цього, яку надали автори методу, полягає в переліку категорій, серед яких доцільно проводити пошук. Це такі категорії:

– реальні предмети світу, котрі фізично втілені, наприклад, Іван Іванович, стілець у кабінеті, Дніпро;

– абстракції фізичних предметів світу, наприклад, людина, тварина, літак, кабель, дім, сад;

– ролі предметів, тобто абстракції їхнього призначення або мети використання. Наприклад, для домену університет значущими є ролі – декан, ректор, студент, викладач; для домену хімічне виробництво – очисна споруда, каталізатор, відстійник; для домену адміністрація міста – платник податків, виборець, мер;

– інциденти, тобто абстрактні події, котрі впливають на зміну стану об’єкта, наприклад, паводок, вибори, залік, прогул;

– взаємодії, тобто об’єкти, які характеризують відношення об’єктів, наприклад, контракт, перехрестя доріг або вулиць, *угода;* специфікації, тобто подання правил, стандартів, критеріїв якості, обмежень користування.

Тож почнемо аналіз домену проблемної галузі послідовно для кожної з перелічених вище категорій явищ, виявляючи суттєві об’єкти та їхні класи (див. п. 3.3.1).

Для класів об’єктів вибираються значущі імена, унікальні в межах домену.

Атрибути об’єктів. Склавши список виявлених об’єктів, для кожного з них потрібно визначити його характерні ознаки або властивості, які в інформатиці називають атрибутами. Кожний атрибут є абстракцією однієї з характеристик об’єкта, котрі властиві всім представникам класу об’єктів. За атрибутом закріплюємо ім’я, унікальне в межах класу.

Зазначимо, що вдало вибрані імена (як кажуть, мнемонічні імена, тобто такі, котрі передають сутність об’єктів, які вони позначають) є важливим чинником для розуміння програм.

Для кожного з визначених атрибутів задаються його можливі значення (типи значень). Способи опису можливих значень можуть бути такі:

– числовий діапазон;

– перелік можливих значень;

– посилання на документ, у якому визначено можливі значення;

– правила генерації значень.

Ідентифікатори об'єктів. Для об'єкта визначається так званий ідентифікатор – це один або кілька атрибутів, значення чи сукупність значень яких точно вирізняють екземпляр об'єкта з-поміж інших у класі. Прикладом ідентифікатора можуть бути: назва або ім'я об'єкта, табельний номер працівника (бо серед них можливі носії однакових прізвищ та імен), номер паспорта, код платника податків, номер автомобіля. Сукупність атрибутів, які становлять ідентифікатор, може залежати від області визначення об'єкта. Так, якщо йдеться про котів однієї родини, то зазвичай кличка кота є унікальною в родині, якщо ж ідеться про котів одного двору, то, можливо, доведеться уточнити кличку кота його масть (Васько.рудий) або ім'ям його хазяїна (*Васько.Олена*).

Посилання на ідентифікатор подається як перелік через крапку атрибутів, котрі входять до складу ідентифікатора, як це видно з наведених вище прикладів. Так само посилання на атрибут при необхідності може уточнюватися класом, поданим через крапку, наприклад: викладач, стаж-роботи, літак.розмах-крил, собака.порода.

Подання інформаційної моделі в цьому методі базується на відомій реляційній моделі даних. Атрибути об'єктів подаються як атрибути відношень за такими правилами:

– кожний екземпляр об'єкта одночасно обов'язково має одне значення (тобто значення не може бути відсутнім або невизначеним);

– атрибут є одновимірним і не має внутрішньої структури або кількох значень одночасно;

– якщо до ідентифікатора входить кілька імен атрибутів, усі вказані імена атрибутів, окрім першого, належать до першого вказаного імені, яким є ім'я об'єкта.

Об'єкти нумеруються. Об'єкт зображається прямокутною рамкою, всередині якої подається номер та ім'я об'єкта, а також імена його атрибутів, наприклад, на рис. 3.1.

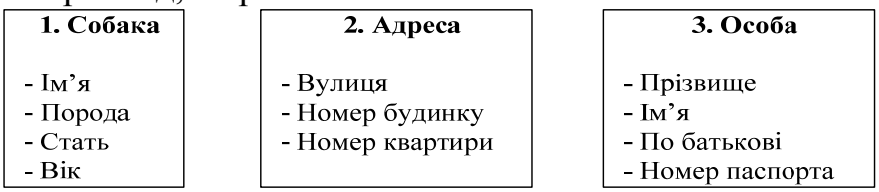

Рисунок 3.1 – Зображення об'єктів з атрибутами

Зв'язки об'єктів. Визначивши склад класів об'єктів домену та властиві їм атрибути, розглянемо зв'язки (відношення) між об'єктами домену.

Об'єкти одного класу можуть брати участь у бінарних, тобто попарних зв'язках з об'єктами іншого або того самого класу.

Розглянемо кілька прикладів зв’язку:

– власник авто має авто, а авто належить власнику;

– прибиральник прибирає кімнату, а кімната прибирається ним;

– проект ведеться керівником, керівник веде проект;

– керівник займає кімнату, кімната належить керівнику;

– літак займає доріжку аеродрому, доріжка зайнята літаком;

– проект має виконавців, виконавці зайняті в проекті;

– керівник керує виконавцем, виконавець підпорядкований керівнику;

– виконавець займає кімнату, кімната містить виконавця. Кожна фраза фіксує можливість екземпляра певного класу об’єктів бути у певному відношенні (або зв’язку) з екземпляром іншого класу (приклади 1–5) або того самого класу (приклад 6). Суттєвою рисою наведених зв’язків є число екземплярів об’єктів, які можуть одночасно брати в ньому участь.

Розрізняються три фундаментальних види зв’язку:

– один до одного (1 : 1), коли у зв’язку беруть участь по одному екземпляру з кожного боку (приклад 5). Одночасно на одній злітній смузі може бути тільки один літак, і один літак може займати тільки одну смугу. Приклади 3 та 4 можуть також бути прикладами зв’язку 1:1, якщо в організації керівник займає окремий кабінет і керує одноосібно тільки одним проектом;

– один до багатьох (1 : n), коли один екземпляр об’єкта певного класу може підтримувати відношення одночасно з декількома екземплярами об’єктів іншого або того самого класу. Приклад 1. Якщо виключається спільне користування авто, але можна мати кілька авто. Інший приклад – 6, коли керівник може мати декілька підлеглих, але в кожного з них один шеф. Якщо керівник може керувати кількома проектами, то приклад 4 також належить до такого виду зв’язку;

– багато до багатьох (m : n), коли у зв’язку можуть брати участь по декілька екземплярів об’єктів з кожного боку. Це ілюструють приклади 1, якщо тими самими кількома автомашинами дозволяється користуватись декільком особам, та 2, коли декілька прибиральників прибирають по черзі кілька кімнат.

Ми бачимо, що опис зв’язків відображає певні вимоги до статичних залежностей, які бувають для задач, котрі має розв’язувати програмна система. Метод С. Шлеєр та С. Меллора передбачає спеціальну графічну нотацію для фіксації зв’язків, що базується на діаграмах відомої моделі Чена сутність — зв’язок (entity — relations) і є інформаційною моделю (онтологією) проблемної галузі.

Подання інформаційної моделі проводиться в такий спосіб.

Зв’язки між об’єктами зображаються стрілками, що вказують

напрямок зв’язку. Біля рамки об’єкта, котрий бере участь у зв’язку, на лінії стрілки вказується роль, яку цей об’єкт підтримує в даному зв’язку. Зв’язок 1:1 позначається двоспрямованою стрілкою, що має по одному наконечнику стрілки з кожного боку. Зв’язок 1:п позначається стрілкою, що має два наконечники з боку того об’єкта, для якого у зв’язку можуть брати участь декілька екземплярів, і, нарешті, по два наконечники з кожного боку має стрілка, яка позначає зв’язок виду *n* : *m*.

Над стрілкою може вказуватися назва (ім’я) зв’язку. Зв’язки, наведені вище, є безумовними, бо обов’язково кожний екземпляр об’єкта заданого класу бере участь у вказаному зв’язку. Передбачено також можливість умовних зв’язків, коли окремі екземпляри об’єктів певного класу за певних умов можуть не брати участі у зв’язку, в цьому разі відповідний кінець стрілки позначається літерою *y*. Наприклад, деякі доріжки аеродрому в певний момент часу можуть бути вільні від літаків. На рис. 3.2 подано фрагменти інформаційної моделі, кожний з яких відповідає одному з прикладів зв’язків 1–8, наведених вище.

При цьому за назву зв’язку обрано літеру *R,* за якою стоїть номер прикладу.

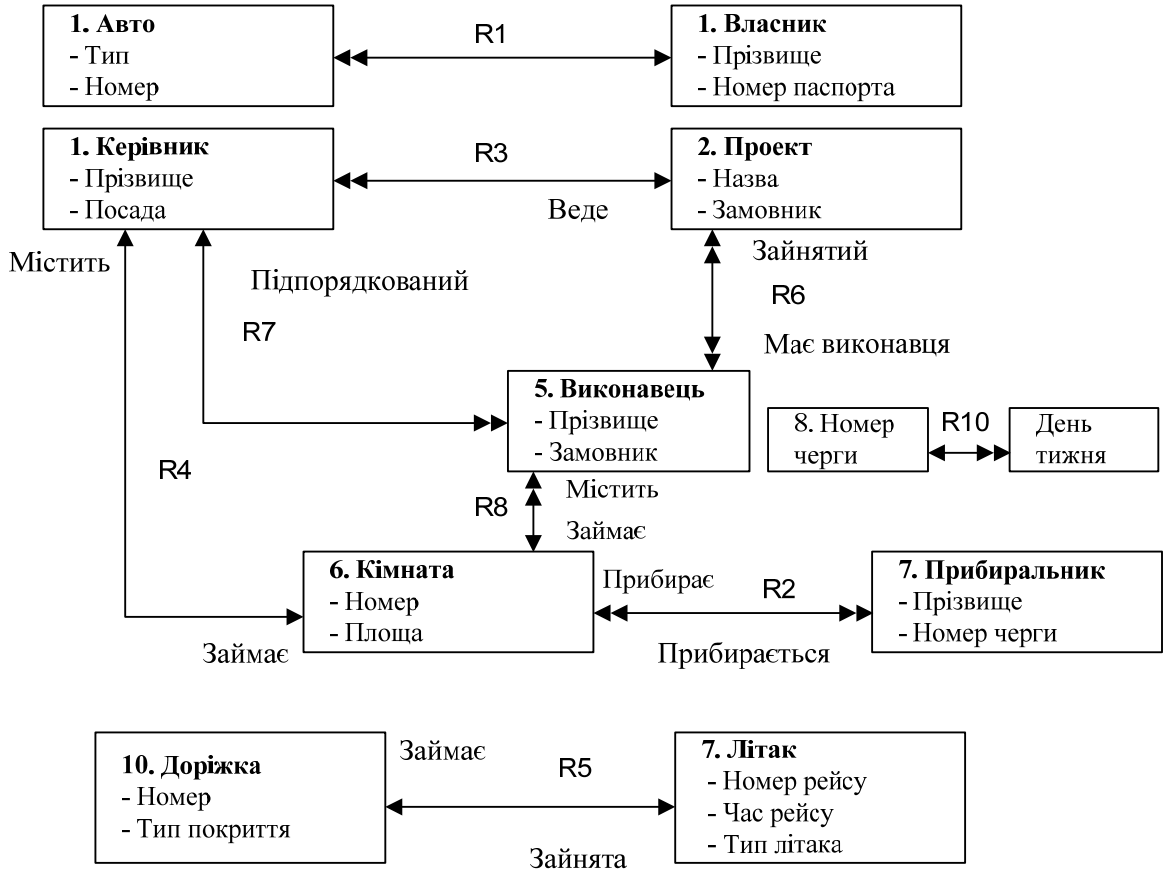

Рисунок 3.2 – Приклади фрагментів інформаційної моделі

Інформаційна модель проблеми на наступних фазах життєвого циклу розробки програмної системи відображається на структури баз

даних. Власне, таке відображення є продуктом проектних рішень з реалізації зв'язків, задекларованих як вимоги до розробки, що буде розглянуто у п. 4.2.1.

Є одне відношення, яке має особливу вагу для подання онтологій. Це відношення успадкування (див. п. 3.3.1), за допомогою якого виражаються спільності та розбіжності між визначеними класами об'єктів. Зазвичай, відношення успадкування подаються на окремих діаграмах – на так званих діаграмах класів. На рис. 3.3 наведено приклади таких діаграм.

При цьому діаграму інформаційної моделі супроводжують неформальним описом усіх об'єктів, їхніх атрибутів та зв'язків, в яких об'єкти беруть участь.

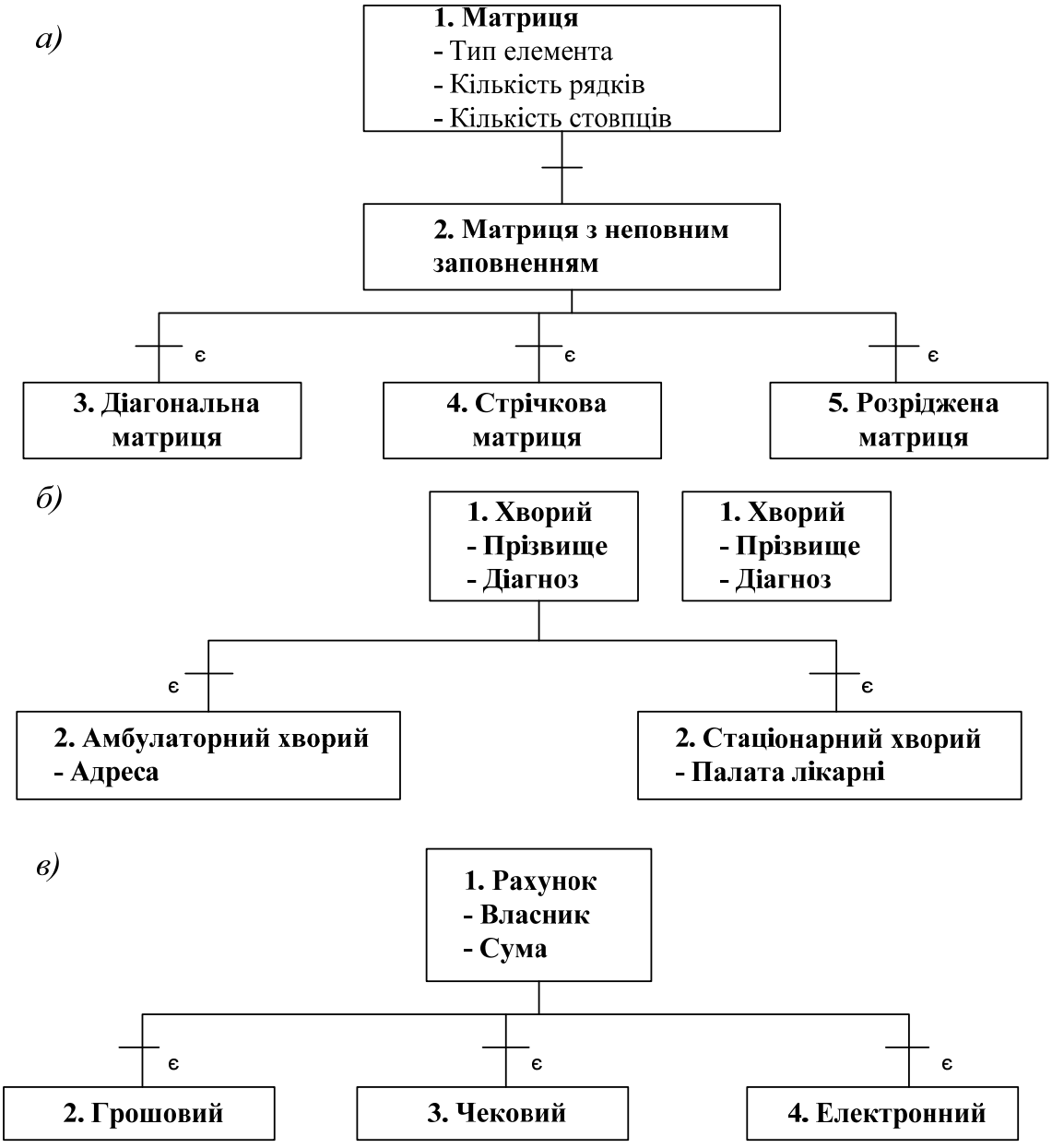

Рисунок 3.3 – Приклади діаграм класів

3.4.2 Модель станів

Модель, яку буде розглянуто нижче, має відображати динаміку змін, котрі відбуваються в стані кожного з визначених у п. 3.4.1 класів об’єктів, або, як кажуть, динаміку їхньої поведінки. Зазначимо, що всі екземпляри одного класу об’єкта, за визначенням поняття клас, мають однакову поведінку. Нагадаємо наведені у п. 3.2.3 базові поняття моделі динаміки поведінки об’єктів:

– стан об’єкта визначається поточними значеннями окремих його атрибутів, а стан домену – сукупністю станів його об’єктів;

– стан об’єкта змінюється внаслідок того, що відбулися певні події або з’явилися певні стимули;

– зміна стану супроводжується певними процесами, котрі визначено для кожного стану як такі, що мають відбутися після досягнення цього стану.

У методі, який ми розглядаємо, запропоновано одразу дві альтернативні нотації для фіксації динамічних аспектів вимог як поведінки визначених об’єктів. Одна з них – графічна – називається *діаграмою переходів у стани* (ДПС). Друга – таблична – називається *таблицею переходів у стани* (ТПС). Обидві нотації базуються на автоматі Мура, відомому з теорії автоматів. Згідно з цим методом, побудову моделі станів починаємо з того, що серед визначених інформаційною моделлю класів об’єктів виділяємо ті, котрі мають динамічну поведінку, тобто змінюють свій стан з плином часу, або, як кажуть, мають життєвий цикл від створення екземпляра об’єкта через зміни його станів і до зникнення об’єкта. Для відображення поведінки таких об’єктів у вимогах до розробки потрібно:

– визначити множину станів, в яких об’єкт може перебувати, при тому кожний стан є абстракцією стану, в котрому може перебувати кожен з екземплярів класу об’єктів;

– визначити множину інцидентів або подій, які спонукають екземпляри класу змінювати свій стан;

– визначити для кожного із зафіксованих станів правила переходу, котрі вказують, в який новий стан перейде екземпляр даного класу, якщо певна подія з визначеної для класу множини подій відбудеться тоді, коли він перебуває в даному стані;

– визначити для кожного з визначених станів дії або процеси, які потрібно виконати при набутті даного стану.

Графічна нотація для подання наведеної вище інформації передбачає таке:

– кожному стану, визначеному для класу об’єктів, присвоюють назву та порядковий номер;

– кожній визначеній події присвоюють унікальну мітку та назву;

– на діаграмі ДПС стан позначається рамкою, яка містить номер та назву стану;

– перехід від стану до стану зображається спрямованою дугою, позначеною міткою та назвою події, яка зумовила перехід;

– початковий стан позначається стрілкою, що веде до відповідної йому рамки, і є станом, в якому екземпляр об'єкта з'являється вперше (ініціалізується). Допускається кілька початкових станів на ДПС;

– заключний стан визначає кінець життєвого циклу екземпляра об'єкта, що може настати в одному із двох випадків – або екземпляр існує далі, але його поведінка втрачає динамічний характер, і тоді такий стан позбавляється спеціальної позначки або екземпляр зникає, і тоді заключний стан позначається пунктирною рамкою;

– під рамкою зазначаються дії, які має виконати екземпляр об'єкта, коли він набуває відповідного рамці стану.

Зазначимо, що при застосуванні ТПС дії, відповідні станам, позначаються окремою нотацією.

Зупинимося коротко на можливих діях при зміні станів.

Дії виконуються екземпляром, котрий змінює стан.

Подія, яка викликає зміни стану, є сигналом керування, що зазвичай передає якісь дані. Вони мають нести досить інформації, щоб визначити екземпляр класу, котрий змінює стан (або створити новий екземпляр) і забезпечити даними відповідні дії. Різновиди дій такі:

– оброблення інформації, яку несе подія;

– зміна певного атрибута об'єкта;

– обчислення;

– генерація події для деякого екземпляра деякого класу (можливо, для самого себе);

– генерація події, що має передаватися зовнішнім щодо даного домену об'єктам, як-от людина-оператор, інша система, фізичний прилад тощо;

– прийом повідомлення про події від зовнішніх об'єктів;

– взаємодія з двома специфічними об'єктами — таймером та системним годинником.

Таймер – це механізм вимірювання інтервалу часу, який вважається вбудованим у даний метод системним об'єктом, що не потребує визначення.

Атрибутами цього об'єкта є:

– унікальний ідентифікатор екземпляра таймера;

– залишок часу (інтервал часу, через який буде подано сигнал про настання певної події);

– мітка події, яка настане, коли залишок часу буде дорівнювати

нулю;

– ідентифікатор екземпляра об’єкта, для якого встановлюється таймер. Екземпляр таймера встановлюється для окремого екземпляра певного керованого об’єкта (наприклад, бак накопичувача, духова шафа печі, шахматист Іванчук) для повідомлення про настання події, даними якої є значення атрибутів таймера. Окремі події передбачено для скидання таймера на нуль та знищення таймера.

Приклад моделі станів з використанням таймера див. на рис. 3.4.

Альтернативною графічній нотації ДПС є таблична нотація – так звана таблиця переходів у стани (ТПС). Кожний з можливих для класу об’єктів станів подається рядком ТПС, а кожна з можливих подій – стовпцем. Клітинка ТПС визначає стан, в який переходить об’єкт, якщо відповідна стовпцю подія відбудеться, коли він перебував у стані, котрий відповідає стовпцю. При цьому можлива ситуація, коли певна комбінація подія – стан не веде до зміни стану екземпляра об’єкта або неможлива. Тоді у клітинці ТПС відповідно зазначається: "подія ігнорується" або "не може бути", або клітинка залишається пустою.

У табл. 3.1 наведено приклад ТПС, що відповідає ДПС, зображеній на рис. 3.4.

Таблиця 3.1– Приклад таблиці переходів

	П1	П2	П3	П4	П5	П6	П7	П8
С1	2							
С2		3						
С3			4					
С4				5				
С5					6			
С6						7	2	
С7								1

Якщо вибирати між ДПС і ТПС, то аргументом на користь першої нотації – ДПС – є її наочність та визначення дій, тоді як друга з них – ТПС – дозволяє зафіксувати всі можливі комбінації стан – подія і забезпечити повноту та несуперечність подання вимог.

Системний годинник – це також вбудований у метод об’єкт, атрибути якого – показники системного часу (години, хвилини, день, місяць, рік) – можна читати.

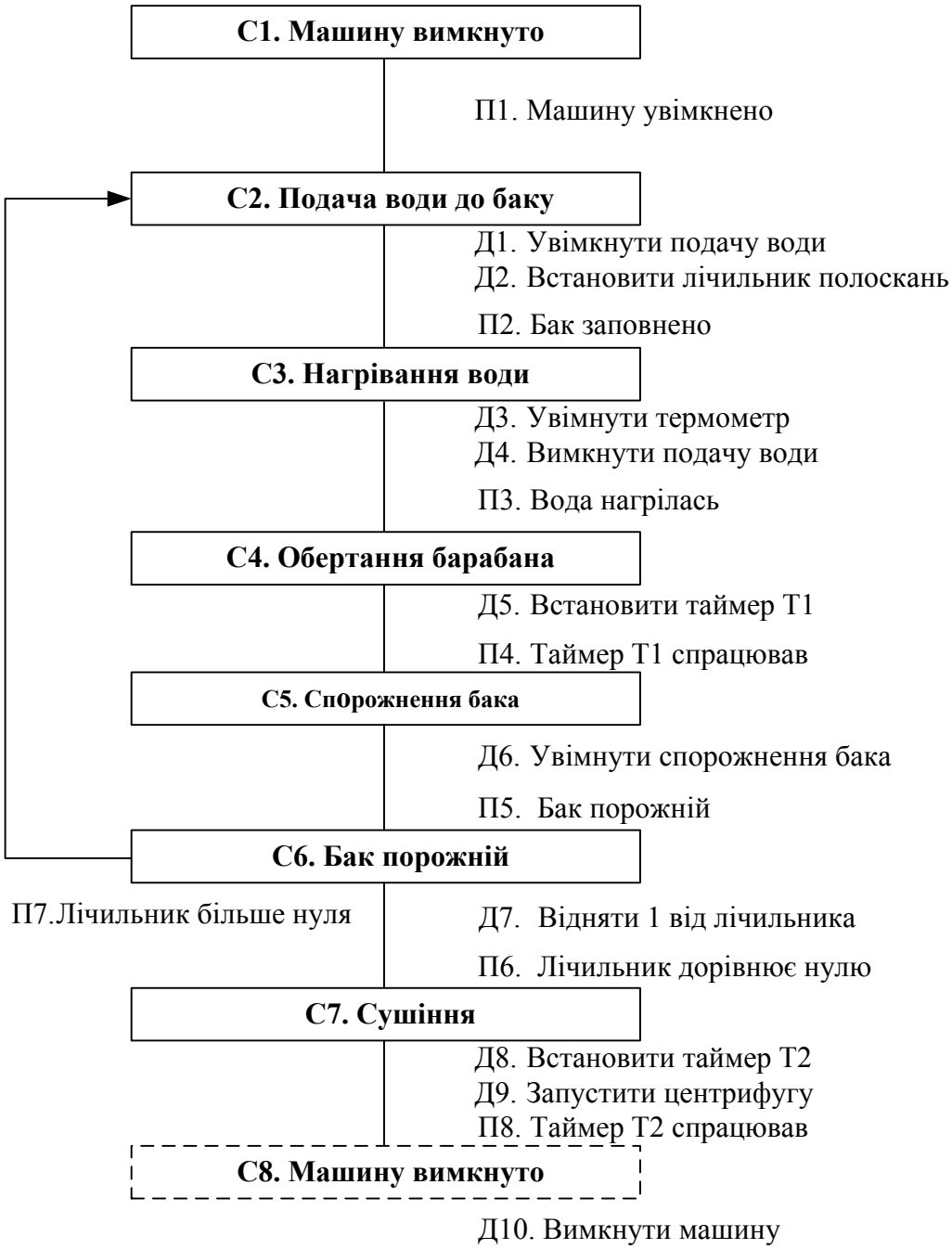

Рисунок 3.4 – ДПС автоматичної пральної машини

Все, що було сказано вище, стосується окремих об’єктів як базових складових – компонентів архітектури системи в цілому.

Важливим принципом об’єднання компонент у систему є наявність для компонент спільних подій, причому найчастіше одна з компонент породжує подію, а інші на неї реагують. На цьому принципі базується

спосіб об’єднання окремих об’єктів у систему.

Програмна система в цілому розглядається як взаємодія об’єктів, що моделюється як обмін між об’єктами системи та зовнішніми об’єктами-повідомленнями про настання певних подій та даних до них. При цьому зовнішні події, про які система не робила запиту, вважаються такими, що запускають систему. Зовнішні події, чекання на які передбачено в системі, подані як події, прохання повідомити про які надсилаються системою до зовнішніх об’єктів (як запит), котрі у відповідь, у свою чергу, надсилають повідомлення про настання подій до об’єктів системи.

Оскільки поведінка об’єкта подана відповідною ДПС, то поведінка системи в цілому подається як схема взаємодії окремих ДПС. Кожній з них присвоюється назва і вони зображаються на схемі овалом з цією назвою. Овали пов’язані між собою стрілками, що відповідають повідомленням про події, які пов’язують окремі ДПС. На стрілці вказується мітка події, а напрямок стрілки відповідає напрямку передавання повідомлення. Зовнішні об’єкти позначаються прямокутними рамками з їхніми назвами.

Приклад взаємодії моделей поведінки об’єктів наведено на рис. 3.5.

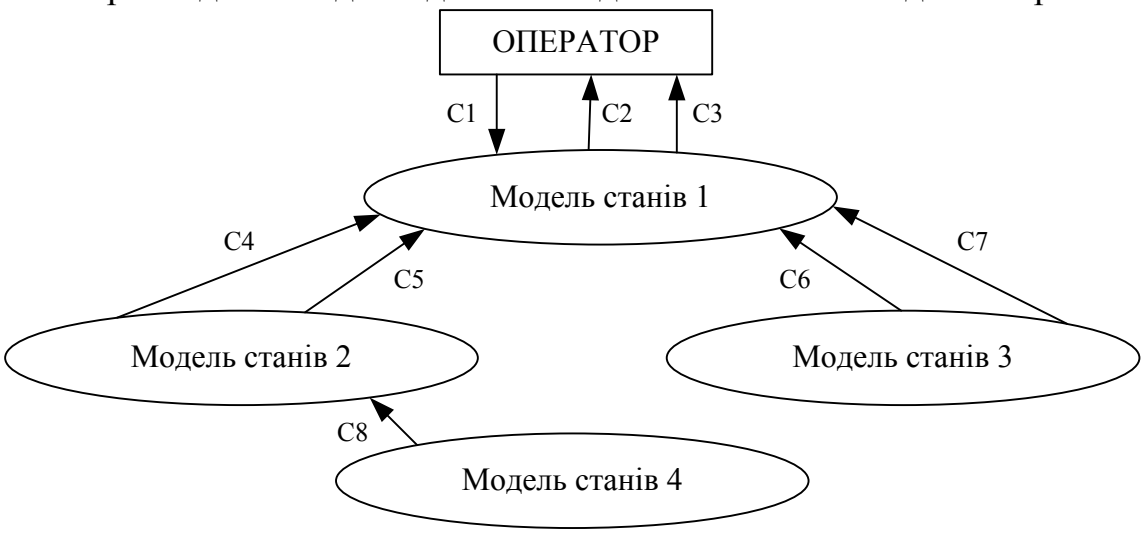

Рисунок 3.5 – Схема взаємодії моделей поведінки об’єктів

3.4.3 Модель процесів

Модель станів об’єктів, за допомогою якої висуваються вимоги до поведінки системи (див. п. 3.4.2), передбачає у своєму складі опис певних дій, котрі супроводжують зміни станів об’єктів. Дії є алгоритмами, що виконуються системою як реакції на події і визначають її функціональність. Розуміння вимог до системи передбачає і розуміння зазначених вище дій, інколи досить складних. Способом, що пропонується для подолання труднощів розуміння дій у даному методі, є декомпозиція їх на окремі складові, які отримали назву процесів. Послідовність

виконуваних процесів утворює потік керування; водночас процеси під час виконання обмінюються даними, що утворюють потоки даних; два зазначені типи потоків пропонується використовувати як моделі алгоритмів дій системи, для подання котрих у даному методі передбачено спеціальну нотацію, якій присвоєно назву діаграми потоків даних дій (ДПДД). Як джерела даних допускаються:

– атрибути об’єктів (що звичайно зберігаються в архівах – файлах або базах даних, які існують і після завершення роботи системи);

– системний годинник як показник системного часу;

– таймери (див. п. 3.4.2);

– дані подій;

– повідомлення зовнішніх об’єктів (людей-операторів, приладів тощо). Правила побудови ДПДД подано нижче:

– кожному з ДПС станів може відповідати тільки одна ДПДД;

– процес зображається на ДПДД як овал, всередині якого подано зміст або назву процесу;

– потоки даних зображено як стрілки, на яких вказуються ідентифікатори даних, що передаються від процесу до процесу; напрямок стрілки до овалу позначає дані, які є входами до процесу, напрямок від овалу – виходи;

– джерела даних зображено як прямокутні рамки чи рамки з відкритими сторонами;

– якщо джерелами даних є архівні об’єкти, відповідні потоки маркуються назвами атрибутів об’єктів, що передаються потоками, при цьому назва відповідного об’єкта може не вказуватися;

– потоки даних від таймера маркуються назвою таймера;

– потоки даних від системного годинника маркуються назвами показників часу (час, година, хвилина, день тощо);

– подія, повідомлення про яку отримує процес, зображається як стрілка, котра маркується назвами даних подій;

– якщо процес, який створив подію, та процес, який приймає повідомлення про подію, обидва належать до тієї самої ДПДД, відповідний потік пов’язує такі процеси;

– якщо подія, яку створив процес певної ДПДД, передається до процесу з іншої ДПДД, для першого із вказаних процесів вона позначається стрілкою, котра веде від процесу в "нікуди", а для другого – до процесу з "нізвідки", причому обидва рази стрілка маркується даними події, які передаються.

Процеси розрізняються за такими типами:

– так званий аксесор, що здійснює доступ до архівів;

– генератор подій;

– перетворювач даних (обчислення);

– перевірка умов.

Потоки керування на ДПДД позначаються пунктирними стрілками.

Якщо процес являє собою перевірку певної умови, при виконанні котрої здійснюється передавання керування до іншого процесу, то відповідний потік керування зображається перекресленою пунктирною стрілкою.

Приклад ДПДД наведено на рис. 3.6.

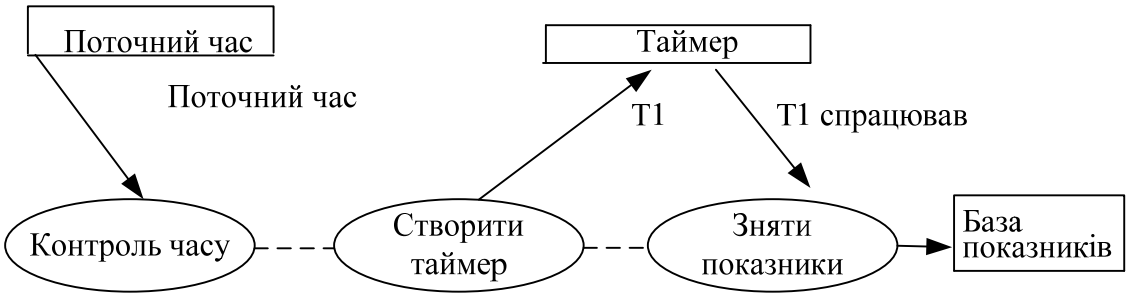

Рисунок 3.6 – Приклад ДПДД

До ДПДД додається неформальний опис функцій процесів, які входять до її складу. Нотація для опису подробиць дії процесів у даному методі не регламентується і залежить від смаків авторів.

Після побудови всіх ДПДД для всіх об’єктів системи доцільно побудувати загальну таблицю процесів для станів, до якої входять такі колонки:

– ідентифікатор процесу;

– тип процесу (див. вище);

– назва процесу;

– назва стану, в якому визначено процес;

– назва дії стану.

Створення такої таблиці має декілька цілей. По-перше, таблиця дає можливість перевірити несуперечність назв та ідентифікаторів процесів, по-друге, перевірити повноту визначених подій та процесів, по-третє, перевірити, чи всі визначені події генеруються певним процесом і чи всі згенеровані події обробляються певним процесом. Крім того, наявність такої таблиці дає можливість виявити процеси, спільні для кількох дій чи станів і уніфікувати їх.

Рекомендовано мати три зразки такої таблиці, кожен з яких має бути впорядковано за окремим критерієм (ключем): за ідентифікатором процесу, за моделлю станів, в яких процес використано, та за типом процесу.

3.4.4 Продукти інженерії вимог за методом С. Шлеєр та С. Меллора

За даним методом, результатом проведеного аналізу вимог до створюваної програмної системи є такі продукти:

1. Інформаційна модель системи (онтологія) у формі:
- діаграми сутність – зв'язок;
- опису об'єктів та їхніх атрибутів (подається неформально);
- опису зв'язків між об'єктами (подається неформально).

2. Модель поведінки об'єктів системи у формі:
- діаграми переходів у стани (ДПС) або таблиці переходів у стани (ТПС);
- опису дій ДПС (подається неформально);
- опису подій ДПС (подається неформально).

3. Модель процесів для станів об'єктів у формі:
- діаграми потоків даних дій (ДПДД);
- таблиці процесів станів;
- опису процесів (подається неформально).

Сукупність перелічених продуктів вважається достатньою для переходу до процесу проектування системи, мета і методи якого подаються в главі 4.

3.5 Метод інженерії вимог І. Джекобсона

3.5.1 Концепція моделі сценаріїв для складання вимог

Метод, про який буде йти мова далі, є, на наш погляд, єдиним методом, котрий вказує послідовний підхід до виявлення об'єктів, суттєвих для домену проблемної галузі. Справді, всі методи декларують – як перший крок – виявлення об'єктів і попереджають, що вдалий склад об'єктів зумовить зрозумілість і точність вимог, але тільки цей підхід дає рекомендації щодо того, із чого починати шлях до розуміння проблеми та пошуку суттєвих для неї об'єктів.

Автори позначили свій метод як базований на варіантах (на прикладах або сценаріях) використання системи, яку має бути побудовано. Домовимось, що в подальшому ми будемо застосовувати термін *сценарій (script)* для позначення прикладу чи варіанта використання системи. Складність проблеми зазвичай переборюється шляхом поділу її на окремі компоненти з меншою складністю. Велика система може бути надто складною, щоб її компонентами були одразу програмні модулі, тому розроблення системи за цим методом починається з осмислення її мети – тобто, для кого і для чого створюється система. Складність загальної мети

виражається через окремі складові мети. Складові мети можуть відповідати функціональним або нефункціональним вимогам, проектним рішенням та аргументам за чи проти інших складових мети. Вони є джерелом вимог до систем та засобом оцінювання їх, засобом виявлення суперечностей між вимогами і встановлення залежностей. Складові мети можуть бути кваліфіковані як жорсткі (обов'язкові) та м'які (бажані), як функціональні чи і нефункціональні, як точка зору користувача чи замовника або як точка зору середовища функціонування системи. Складові мети також можуть перебувати між собою у певних відношеннях, як-от: конфліктувати, кооперуватися, залежати чи не залежати одна від одної.

Наступним кроком є визначення носіїв інтересів, яким відповідає кожна зі складових мети, та можливих прикладів задоволення складових мети як сценаріїв роботи системи, котрі є уявленням користувачів про призначення й функції системи, що можна вважати першою ітерацією вимог до розробки.

Отже відбувається послідовна декомпозиція складності проблеми:

– складна проблема трансформується в сукупність складових мети;

– кожна із складових мети трансформується в сукупність можливих прикладів використання системи, тобто прикладів реалізації складових мети, що позначаються як сценарії;

– сценарії трансформуються в процесі аналізу їх у сукупність взаємодіючих об'єктів. Визначений таким чином ланцюг трансформацій (проблема – складові мети – сценарії – об'єкти) відображає ступені концептуалізації, тобто досягнення розуміння проблеми, послідовного зниження складності її частин та підвищення рівня формалізації їхніх моделей.

Зазначимо, що наведені вище трансформації зазвичай відображаються в термінах базових понять проблемної області, тож онтологія домену, якщо її вже створено, активно використовується для подання акторів та сценаріїв або твориться в процесі побудови такого подання. Тобто, онтологія домену є складовою моделі і вимог і за методом І. Джекобсона.

Вважається, що кожен сценарій запускає в роботу певний користувач, який є носієм певної мети. Абстракція особи користувача як певної ролі – ініціатора запуску певної роботи, поданої сценарієм, та обміну інформацією із системою – називається *актором (actor)*. Це абстрактне поняття, що узагальнює поняття діючої особи системи (як основної, для обслуговування якої систему замовлено, так і вторинної, для службового персоналу системи). Фіксація акторів є також певним кроком визначення складових мети системи (носіями яких є актори) та постачальників завдань, для вирішення яких створюється система.

Актор вважається зовнішнім фактором системи, дії котрого мають

недетермінований характер. Таким чином, підкреслюється різниця між користувачем як конкретною особою, й актором – роллю, яку будь-яка особа може відігравати в системі.

В ролі актора може виступати й інша програмна система, якщо вона ініціює виконання певних робіт даної системи, тобто актор є абстракцією зовнішнього об'єкта, екземпляр якого може бути людиною або зовнішньою системою.

Актор в моделі представлений класом, а користувач – екземпляром класу. Одна особа може бути екземпляром декількох акторів (наприклад, водій та касир), але ми розглядаємо у вимогах тільки ролі та сценарії, в яких вони беруть участь.

Особа-в-ролі або екземпляр актора – запускає ряд операцій у системі (транзакцію), які ми називаємо сценарієм. Коли користувач як екземпляр актора вводить стимул, стартує екземпляр цього сценарію, що приводить до виконання ряду дій відповідної транзакції, які закінчуються тоді, коли екземпляр сценарію знову чекає на вхідний стимул від екземпляра актора.

Екземпляр сценарію існує, поки він виконується. Його можна вважати екземпляром класу, описом якого є опис транзакції.

Для актора визначено такі правила:

– кожен сценарій може запускати тільки один актор;

– кожен актор може запускати кілька сценаріїв;

– взаємодія акторів в інтересах системи (тобто як складова її функціональності) дозволяється тільки через передбачені для цього сценарії;

– актор не визначає сценарію, він лише ініціює ланцюжок подій, який визначить сценарій;

– для кожного актора визначаються (неформально) його інтерфейси з тими сценаріями, які він запускає.

Сценарій – це повне протікання подій у системі й, очевидно, має стан та поведінку. Кожна взаємодія між актором та системою розглядається як новий сценарій. Сценарій може розглядатися як об'єкт. Якщо багато запусків сценарію системи мають подібну поведінку, можемо розглядати їх як клас сценаріїв. Виклик сценарію є породженням екземпляра класу. Отже, сценарії – це транзакції із внутрішнім станом. Для них складаються детальні описи – вони є критичними для ідентифікації дійсних об'єктів системи.

Модель системи керується сценаріями, внесення змін має здійснюватися шляхом заміни потрібних акторів та сценаріїв, які вони запускають. Така модель відображає побажання користувачів і легко змінюється за їхньою волею. Користувач добре розуміє її, і до початку проектування на ній можна відпрацювати його проблеми.

Зокрема, відпрацьовується інтерфейс сценарію – за допомогою прототипу можна моделювати виконання сценарію.

Введення акторів дозволяє відповісти на запитання: для чого робиться система? Хто її головний користувач? Актори визначають зовнішнє оточення системи, а її внутрішня суть визначається сценаріями.

Сценарій – це повний ланцюжок подій, ініційованих актором, та специфікація реакції на них. Сукупність сценаріїв визначає всі можливі шляхи використання системи. Кожного актора обслуговує своя сукупність сценаріїв.

Можна виділити базову мету подій, суттєву для сценарію та його розуміння, а також альтернативні — в тому числі і при помилках користувача тощо.

Розглядаючи окремі сценарії, ми тим самим розподіляємо функціональність системи на окремі складові, над якими можна працювати паралельно.

Для моделі сценаріїв пропонується спеціальна графічна нотація, основні правила якої наведено нижче:

– актор позначається іконкою людини, під якою вказується його назва;

– сценарій зображається овалом, всередині якого вказується його назва (що, зазвичай, відображає складові мети, які реалізуються сценарієм);

– іконка актора з’єднується стрілкою з кожним сценарієм, який запускає відповідний актор.

Приклад 1. На рис. 3.7 наведено приклад діаграми сценаріїв для клієнта банку.

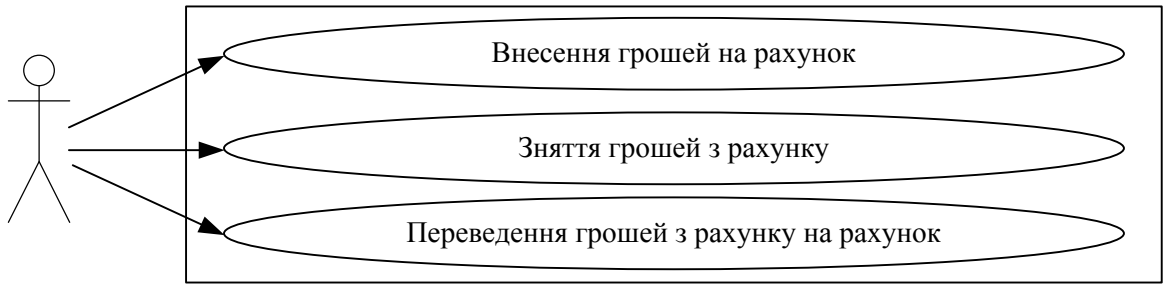

Рисунок 3.7 – Приклад діаграми сценаріїв для клієнта банку

Те, що клієнта визначено як актора, який запускає певний сценарій, означає, що для реалізації своєї складової мети він звертається не до касира чи клерка банку, а безпосередньо до термінала системи (бо в іншому разі сценарії розрахунків запускав би касир, а клієнт, який взаємодіяв би із системою опосередковано через касира, не розглядався б як актор системи). Зверніть увагу, що всі сценарії, включені до системи,

обведено рамкою, яка позначає межу системи, а актор перебуває поза рамкою, бо він розглядається як зовнішній фактор щодо системи.

Приклад 2. Нехай нам замовлено побудувати автоматизовану бібліотечну систему. Сформулюємо кілька складових мети її побудови:

– автоматична реєстрація читача;

– перевірка наявності при зверненні читача за літературою абонемента та боргів з отриманої раніше літератури, термін користування якої скінчився;

– фіксація книг, які замовляє читач;

– фіксація факту видачі замовлень, які виконано;

– фіксація повернення книжок та журналів;

– організація черги відкладених замовлень, які не можна виконати через зайнятість їх іншими читачами;

– повідомлення читачеві про можливість виконання відкладених раніше замовлень;

– виявлення боржників і надсилання їм попереджувальних повідомлень;

На рис. 3.8 показано діаграму сценаріїв, що відповідає одному з варіантів вимог до бібліотечної системи. Розглянемо цей варіант.

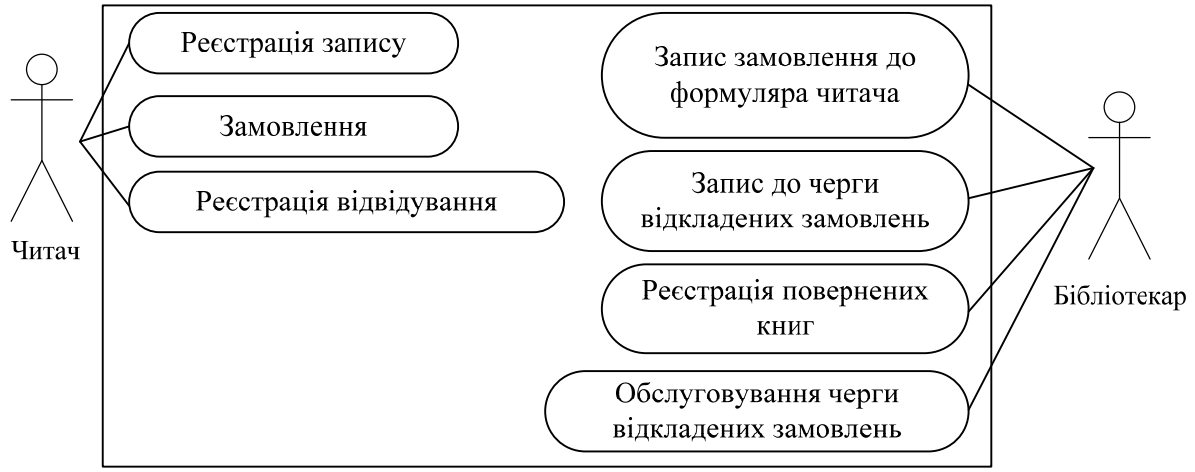

Рисунок 3.8 – Діаграма сценаріїв бібліотеки

Цілі 1–8 визначають ті бібліотечні процеси, які ми бажаємо автоматизувати. Вимогу до автоматизації певного процесу в системі, яку ми будуємо, буде подано одним або кількома сценаріями.

Визначимо, які актори будуть запускати сценарії.

Першим з акторів назвемо читача. Якщо ми домовилися, що саме читач запускає сценарії, котрі відповідають складовим мети 1, 2 та 3, то тим самим ми задекларували, що відвідування бібліотеки починається для нього з того, що він підходить до термінала й запускає відповідні сценарії, інтерфейс з якими передбачає повідомлення читачем системі своїх даних

під час реєстрації (складова мети 1), повідомлення абонементного номера, якщо він уже зареєстрований (складова мети 2), повідомлення свого замовлення (складова мети 3). Таке визначення треба розуміти як бажання замовника системи, щоб усе зазначене вище робилося без втручання бібліотекаря.

Другим актором обираємо бібліотекаря. Якщо ми віднесли до нього відповідальність за складові мети 4–8 (які він реалізує шляхом запуску відповідних сценаріїв), то тим самим ми висловили вимоги, щоб бібліотекар видавав замовлені примірники в руки читача, запускаючи при цьому роботу системи, результатом якої буде перенесення записів про виконані замовлення до формуляра читача, а про невиконані – до черги відкладених замовлень. При поверненні літератури запускаються сценарії відповідного занотовування у формулярі читача про звільнення поверненого примірника, який відтепер доступний, у тому числі й для відкладених замовлень.

У цьому процесі не показано діяльності з пошуку фізичних примірників замовленої літератури та переміщення їх від книгосховища до місця обслуговування читача і навпаки, бо замовник нашої системи не вимагає, щоб цю діяльність було автоматизовано, тому ми відносимо її до бібліотекаря і не відображаємо в сценаріях.

Приклад 3. Розглянемо систему, націлену на обслуговування роботи бібліотечної ради з формування фондів бібліотек. Передбачаються такі складові мети системи: збір пропозицій про передплату періодичних видань та придбання нових книг, погодження поданих заявок з обмеженнями фінансування, голосування про прийняття окремих пропозицій та ухвалення рішень за результатами голосування.

Першим актором нашої системи визначимо адміністратора бібліотечного фонду, а другим – члена бібліотечної ради.

Взаємодія між цими двома акторами визначається розподілом між ними відповідальності за формування фондів. Варіант такого розподілу, згідно з яким член бібліотечної ради подає пропозиції про замовлення книжок і передплатних видань і голосує за зведений список замовлень від усіх членів ради, фіксує наведена на рис. 3.9 діаграма сценаріїв. На рис. 3.9 зображено, що саме член бібліотечної ради запускає сценарії прийняття пропозицій та голосування. Це означає, що накопичення пропозицій виконується системою автоматично. Так само і голосування виконується системою як сценарій голосування, виконання якого полягає в поданні чергового замовлення членові ради, прийнятті його думки (за чи проти).

Рисунок 3.9 – Діаграма сценаріїв погодження замовлень

Адміністратор викликає сценарії:

– узагальнення поданих замовлень (вилучення однакових замовлень, поданих членами ради, впорядковування узагальненого списку поданих замовлень);

– обрахування сумарної вартості поданих замовлень і, якщо вона перевищує надані фінансові ресурси (бюджет), розсилання списку замовлень на голосування членам бібліотечної ради;

– оброблення результатів голосування і сортування замовлень за отриманими рейтингами;

– визначення частини списку замовлень, що вкладається в наданий бюджет;

– формування бланків замовлень до видавництв та розсилання їх. Нагадаємо, що кожен сценарій являє собою певну роботу (транзакцію), яку система виконує автоматично.

Відношення між сценаріями. Між сценаріями можна задати певні відношення, які зображаються на діаграмі сценаріїв пунктирними стрілками, позначеними назвами відповідних відношень.

Для сценаріїв визначено два типи відношень:

а) відношення "розширює" означає той факт, що функції одного сценарію є доповненням до функцій іншого. Зазвичай воно застосовується, коли маємо кілька варіантів того самого сценарію, тоді інваріантна його частина зображається як основний сценарій, а окремі варіанти змінної частини – як розширення. При цьому основний сценарій є стійкий щодо можливих випадків розширення його функцій не змінюється при такому розширенні і не залежить від нього.

Наведемо типові приклади (рис. 3.10), коли доцільне застосування відношення розширення:

1) для моделювання необов’язкових фрагментів сценаріїв (див. рис. 3.10, а);

2) для моделювання альтернативного перебігу подій у сценарії, що рідко відбувається (див. рис. 3.10, *б);*

3) для подання ситуації, коли кілька окремих сценаріїв організовуються як акції в спеціальний сценарій типу меню (див. рис. 3.10, *в).*

Можемо вважати, що під час виконання сценарію, який є розширенням, переривається виконання основного сценарію (того, який розширюється), причому другий не знає чи буде розширення і яке саме. Після завершення виконання сценарію розширення буде далі виконуватися основний сценарій;

б) відношення "використовує" означає, що певний сценарій може бути використано для розширення кількома іншими сценаріями (аналог процедури в мовах програмування).

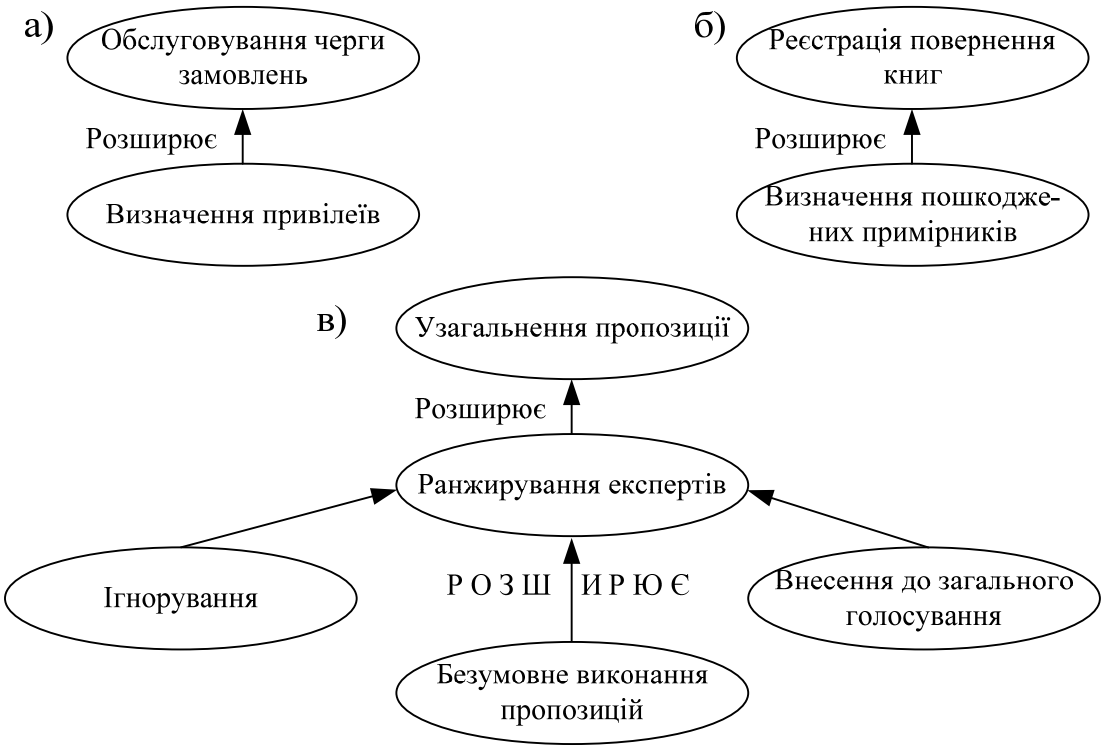

Рисунок 3.10 – Приклади розширення сценаріїв

Обидва введені відношення є інструментом визначення успадкування, якщо сценарії вважати об’єктами. Відмінність між ними полягає в тому, що при розширенні функція розглядається як доповнення до основної і може бути зрозумілою тільки в парі з нею, тоді як у відношенні "використовує" додаткова функція має самостійне визначення і її можна використати будь-де.

На рис. 3.11 показано, що сценарій "сортувати" пов’язаний відношенням "використовує" з кількома сценаріями.

Рисунок 3.11 – Приклад відношення "використовує"

Підсумовуючи сказане вище, можна зробити висновок, що продуктом першої стадії інженерії вимог — збір вимог — є модель вимог, яка складається з трьох частин:

– онтологія домену;

– модель сценаріїв, яка називається діаграмою сценаріїв;

– опис інтерфейсів сценаріїв.

Даний метод не регламентує жорстко нотацію для першої частини, тому можна скористатися нотацією моделі сутність – зв’язок, про яку йшла мова в п. 3.4.1.

Модель сценаріїв супроводжується неформальним описом кожного із сценаріїв, нотація якого не регламентується. Як один із варіантів, опис сценарію може бути подано як послідовність таких елементів:

– назва, яка позначає сценарій на діаграмах моделі вимог і яка є засобом посилання на сценарій;

– анотація (короткий зміст у неформальному поданні);

– актори, які можуть запускати сценарій;

– визначення всіх аспектів взаємодії системи з акторами: можливих дій актора та їхніх можливих наслідків; заборонених дій актора, якщо такі є, та їхніх можливих наслідків;

– передумови, які визначають початковий стан, тобто стан на момент запуску сценарію, необхідний для його успішного виконання, наприклад, наявність даних, яких він потребує;

– власне функції, котрі здійснюються під час виконання сценарію. Вони визначають порядок, зміст та альтернативи дій, які виконуються в сценарії, тобто його алгоритми;

– виняткові або нестандартні ситуації, що можуть з’явитися під час виконання сценарію і завадити його виконанню або потребувати спеціальних дій для усунення їх (наприклад, помилка в діях актора, яку здатна розпізнати система);

– дії, котрі є реакцією на передбачувані нестандартні ситуації;

– умови завершення сценарію;

– постумови, які визначають кінцевий стан сценарію під час його

завершення.

На подальших стадіях осмислення проблеми сценарій використання трансформується в сценарій поведінки системи — до наведених елементів додаються ті, що пов’язані з конструюванням рішення цільової проблеми та нефункціональними вимогами, наприклад:

– механізми запуску сценарію (наприклад, позиції меню);

– механізми введення даних;

– поведінка при виникненні надзвичайних ситуацій. Отже, новий опис успадковує попередній і деталізує його. Наступним кроком може бути сценарій тестування, який успадковує попередній, але розширює його визначенням очікуваних результатів та процедури тестування.

Опис інтерфейсів також подається неформально. Слід зауважити, що саме інтерфейси визначають, якими бачать систему її користувачі, тому корисно погодити з ними вже на стадії складання вимог усі подробиці взаємодії – як виглядає панель кожного з акторів, які вона містить елементи (меню, вікна вводу, кнопки-індикатори тощо).

Побудова прототипу системи, котра моделює реакцію цієї системи на дії актора, допоможе відпрацювати деталі й усунути взаємні непорозуміння між замовником і розробником.

3.5.2 Модель аналізу вимог. Визначення об’єктів

Модель вимог дає узагальнене уявлення про ті послуги (подані сукупністю сценаріїв), які майбутня система має надавати її клієнтам (акторам). Таке подання є предметом аналізу з метою подальшого структурування проблеми, котру розв’язуватиме згадана система. Оскільки обраною нами архітектурою є об’єктна архітектура, то результатом структурування має бути сукупність об’єктів, взаємодія яких визначає функціональність системи.

Означену сукупність знаходять шляхом послідовної декомпозиції кожного із сценаріїв на об’єкти, які відображають дії сценарію.

Декомпозиція сценарію керується намаганням провести такий вибір об’єктів, який дасть змогу забезпечити спроможність системи до адаптації в разі зміни умов або потреб використання системи. Нагадаємо, що аксіомою сучасного погляду на програмну інженерію є гасло: "Всяка працююча програмна система згодом потребує змін". Потребу в змінах готових систем усвідомлено професіоналами з програмної інженерії як об’єктивну реальність, а не лише як наслідок недоробок. Тож стратегія вибору базується на таких принципах:

– зміна вимог неминуча;

– об’єкт має модифікуватися тільки внаслідок зміни відповідних вимог до системи;

– об’єкт має бути стійким до модифікації і сприяти розумінню

системи;

– стійкість до модифікацій (або стабільність) системи розуміється як їхня локальність, яка полягає в тому, що зміна кожної з вимог має вести до відповідних змін якнайменшої кількості об'єктів (в ідеалі – одного об'єкта).

Керуючись переліченими принципами, в даному методі пропонується розрізняти типи об'єктів залежно від їхньої схильності до змін. Для її оцінки простір, в якому функціонує система, структуровано такими трьома вимірами:

– інформація, яку обробляє система (її внутрішній стан);

– поведінка системи;

– презентація системи (її інтерфейси з користувачами та іншими системами).

Вибір вимірів зумовлений експертними дослідженнями динаміки змін діючих систем щодо наведених вимірів. Результати таких досліджень засвідчують:

– впродовж життєвого циклу найбільших змін зазнають вимоги до презентації системи;

– поведінка системи суттєво більш консервативна, але зазнає змін досить часто;

– характер і структура інформації, яку обробляє система, є найстабільнішим виміром щодо попередніх двох.

Тож доцільно для кожного з вимірів функціональності системи мати свою сукупність об'єктів, яка його відображає; таким поділом ми локалізуємо в тексті подання вимог найстабільніші фрагменти, наймобільніші та проміжні. Згідно із сказаним вище, ми розглядаємо три типи об'єктів:

– об'єкти-сутності;

– об'єкти інтерфейсу;

– керуючі об'єкти.

Об'єкти-сутності (object substance) моделюють у системі довгоживучу інформацію, яка зберігається після виконання сценарію. Зазвичай їм відповідають реальні сутності, котрі відображаються в базах даних. Більшість об'єктів-сутностей може бути виявлено з аналізу онтології проблемної області, але до уваги беруться тільки ті з них, на які посилаються в сценаріях.

Об'єкти інтерфейсу (object interface) містять у собі функціональності, залежні безпосередньо від оточення системи й визначені в сценарії. За їхньою допомогою актори взаємодіють із сценаріями системи – їхнім завданням є трансляція інформації, яку вводить актор у події, на які реагує система, та зворотна трансляція подій, які виробляє система у повідомлення для актора. Такі об'єкти

визначаються шляхом аналізу описів інтерфейсів сценаріїв моделі вимог та аналізу дій акторів із запуску кожного з відповідних йому сценаріїв. Як перше наближення, один інтерфейсний об’єкт може бути визначено для однієї пари актор – сценарій. Можна побудувати ієрархію інтерфейсних об’єктів за відношенням "містить" – наприклад, панель містить кнопки, індикатори, меню тощо.

Інтерфейси можна розподілити на два види: "з людиною" та "з системою". При виявленні інтерфейсних об’єктів визначальним є передбачення змін – кожну точку можливих змін у сценарії доцільно визначати як інтерфейсний об’єкт.

Керуючі об’єкти – це об’єкти, які перетворюють інформацію, введену об’єктами інтерфейсу і подану об’єктами-сутностями, в інформацію, що виводиться інтерфейсними об’єктами (іншими словами, керуючі об’єкти відповідають функціям перероблення інформації). Часто вони є своєрідним "клеєм" для з’єднання об’єктів, формуючи ланцюжки подій і, в такий спосіб, взаємодію об’єктів.

Такий поділ слугує складовим мети з локалізації змін у системі. При перетворенні моделі вимог на модель аналізу кожний сценарій розбивається на об’єкти зазначених вище трьох типів. При цьому один і той самий об’єкт може бути присутнім в кількох сценаріях, і важливо розпізнати такі об’єкти, щоб уніфікувати їхні функції та визначити їх як єдиний об’єкт. Критерій розпізнавання є такий: якщо в різних сценаріях посилаються на один і той самий екземпляр об’єкта, то мова йде про той самий об’єкт.

Виділяючи об’єкти, формуємо базис архітектури системи як сукупність взаємодіючих об’єктів, для кожного з яких можна дослідити зв’язок з відповідним сценарієм моделі вимог. Отже, дотримується принцип трасування вимог від моделі вимог до моделі аналізу.

Нагадаємо, що об’єкт інкапсулює в собі певні атрибути, котрі визначають стан об’єкта та його поведінку. Цю поведінку визначають операції, які об’єкт може виконувати, та стани, в яких він може перебувати.

Модель аналізу має відповідну графічну нотацію: об’єкти-сутності зображаються символом  , об’єкти інтерфейсу зображаються символом  , керуючі об’єкти зображаються символом  .

Атрибути об’єктів подані прямокутниками, поєднаними прямою лінією з символом об’єкта, при цьому на лінії вказується назва атрибута, а в прямокутнику – його тип.

Між об’єктами визначаються асоціації, які зображаються одно- чи

двоспрямованими стрілками, на яких вказуються назви асоціацій.

Серед асоціацій застосовуються найпоширеніші:

– "взаємодіє";
– "складається з";
– "виконує роль";
– "успадковує";
– "розширює";
– "використовує".

Зазначимо, що асоціації між об’єктами суттєво відрізняються від асоціацій у моделях даних. Другі націлені переважно на здійснення навігації в базах даних, тоді як перші означають взаємодію об’єктів.

Як приклади, на рис. 3.12 подано фрагменти моделі аналізу, що відповідають діаграмі сценаріїв бібліотечної системи, яку зображено на рис. 3.8.

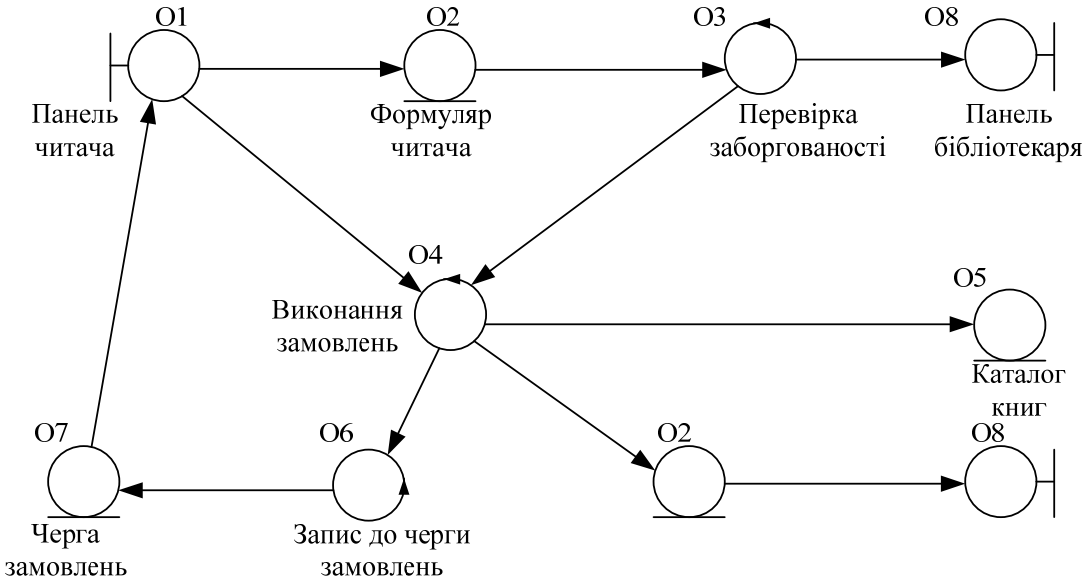

Рисунок 3.12 – Асоціації між об’єктами бібліотечної системи

3.5.3 Продукти інженерії вимог за методом І. Джекобсона

Продуктами стадії аналізу вимог за даним методом є:

– онтологія домену;
– модель сценаріїв;
– неформальний опис сценаріїв та акторів;
– опис інтерфейсів сценаріїв та акторів;
– діаграми взаємодії об’єктів сценаріїв.

Подальша деталізація проблеми за цим методом відбувається на наступних етапах життєвого циклу (див. розділ 4), при цьому зберігається трасування вимог, тобто відстеження відповідності об’єктів впродовж усіх етапів розробки.

Контрольні запитання і завдання

1. Як називається фаза життєвого циклу розробки програмного забезпечення, на якій формується контракт між замовником і виконавцем розробки?

2. Назвіть дійових осіб процесу формування вимог.

3. Назвіть джерела відомостей про вимоги.

4. Якою є послідовність кроків з урахування діючої системи в новій розробці?

5. Назвіть категорії класифікації вимог.

6. Складові мети і складові концептуального моделювання проблеми.

7. Що означає онтологія в концептуальному моделюванні проблеми?

8. Поясніть суть відношень, за допомогою яких будуються поняття: узагальнення /конкретизація, агрегація/, декомпозиція, абстракція, асоціація.

9. Які елементи моделювання динамічних властивостей доменів Ви можете назвати?

10. Назвіть елементи об’єктно-орієнтованого моделювання програмних систем.

11. У чому полягає принцип приховування інформації? Що дає цей принцип:

а) замовникові; б) виконавцеві?

12. Визначте операцію успадкування класів.

13. Назвіть продукти аналізу домену за методом Шлеєр і Меллора.

14. Якою є онтологія домену за методом Шлеєр і Меллора?

15. У чому суть і якою є нотація моделі станів за методом Шлеєр і Меллора?

16. У чому суть і якою є нотація моделі процесів за методом Шлеєр і Меллора?

17. У чому полягає концепція моделі сценаріїв для складання вимог? Поняття "актор".

18. Наведіть нотацію діаграми сценаріїв за методом Джекобсона.

19. Назвіть базові відношення сценаріїв методу Джекобсона і мету використання їх.

20. Які принципи і мета класифікації об’єктів за методом Джекобсона?

21. Якою є нотація взаємодії об’єктів у методі Джекобсона?

4 ПРОЕКТУВАННЯ ПРОГРАМНИХ СИСТЕМ

4.1 Проектування як процес

Проектування – це етап життєвого циклу розроблення програмних систем, наступний після інженерії вимог. Завданням цього етапу є перетворення побажань замовників системи, які ми подали як моделі вимог, у проектні рішення, що забезпечать здійснення згаданих побажань у формі відповідної системи програмування. Таким чином, під час проектування виконується трансформація простору вимог у простір проектних рішень. При цьому можна виділити процеси, котрі можна вважати відносно незалежними одне від одного і виконувати як послідовно, так і паралельно, окремими командами виконавців. Це такі процеси:

– концептуальне проектування полягає в уточненні розуміння й узгодження деталей вимог;

– архітектурне проектування полягає у визначенні головних структурних особливостей системи, яку будують;

– технічне проектування полягає у відображенні вимог середовища функціонування і розроблення системи та у визначенні всіх конструкцій як композицій компонент;

– детальне проектування полягає у визначенні подробиць функціонування та зв’язків для всіх компонент системи.

В основі проектування будь-якого продукту лежить парадигма подолання складності загального завдання шляхом декомпозиції цільового продукту на окремі його складові або компоненти. Це твердження діє і для програмних систем як продуктів програмної інженерії. У попередніх главах було зазначено, що для сучасного стану розвитку програмної інженерії домінуючою є об’єктно-орієнтована парадигма, за якою будь-яка система розглядається як сукупність взаємодіючих об’єктів, тож усі наведені вище підпроцеси проектування ми будемо розглядати щодо тих об’єктів, які було визначено на попередньому етапі життєвого циклу розробки – етапі інженерії вимог.

4.2 Концептуальне проектування

Уточнення вимог та адекватного розуміння їх замовником проводиться в кількох напрямах. Зупинимося на головних з них, котрі відповідають вимірам простору визначення завдань системи, наведеним у п. 3.5.2.

4.2.1 Уточнення даних

Уточнення інформації, яку система обробляє як дані, потребує відповідей на низку запитань, котрі мають бути отримані як результат даного підпроцесу.

По-перше, визначаються джерела надходження даних та те, яка сторона несе відповідальність за їхню достовірність – джерело чи система, що їх отримує. Якщо відповідальною визначено систему, необхідно доповнити проект системи відповідними блоками верифікації даних. По-друге, уточнюються атрибути даних. По-третє, визначаються способи матеріалізації зв'язків між об'єктами у формі відповідної організації даних. На цьому питанні зупинимося більш докладно.

Нагадаємо, що суттєві ознаки об'єкта подаються його атрибутами, причому один атрибут або певна сукупність атрибутів є визначальним для ідентифікації екземпляра об'єкта і називається ідентифікатором об'єкта. Об'єкти можуть перебувати у відношеннях або зв'язках між собою.

Коли в онтології домену визначено, що об'єкт А перебуває в певному відношенні чи зв'язку з об'єктом В, то мова йде про зв'язки між екземплярами об'єктів. Кількість екземплярів, які можуть брати участь у зв'язку з кожної сторони (А та В), визначає тип зв'язку (див. п. 3.4.1). Зазвичай зв'язок матеріалізується за допомогою додаткових атрибутів даних або навіть структур даних.

Будемо розрізняти статичні, сталі зв'язки, котрі не змінюються або змінюються досить рідко (наприклад, чоловік – дружина, батько – син, фірма – адреса) та динамічні зв'язки, які мають певні стани, що можуть змінюватися протягом робочого сеансу системи.

Статичні зв'язки реалізуються шляхом додавання спеціальних атрибутів для об'єктів, котрі беруть участь у зв'язку. Беручи до уваги, що переважною моделлю подання даних на сьогодні є реляційна модель, в якій не дозволяється мати множинні (повторювані) значення атрибутів, згадане додавання виконується за такими правилами:

а) при зв'язку 1 : 1 додатковий атрибут може визначатися для одного будь-якого з пов'язаних об'єктів і містити ідентифікатор екземпляра, який бере участь у зв'язку. Наприклад, якщо мова йде про паркування автомобілів у боксах гаража, то ідентифікатор бокса може вказуватися як атрибут автомобіля, при цьому ідентифікатор автомобіля не буде належати до атрибутів бокса; можливий і варіант навпаки, коли додатковий атрибут для подання зв'язку надається боксу;

б) при зв'язку 1 : n додатковий атрибут надається об'єкту, N екземплярів якого можуть брати участь у зв'язку. Наприклад, якщо працівник має кілька станків, то такий зв'язок є шляхом надання об'єкту "станок" додаткового атрибута "власник";

в) при зв'язку n : m утворюється додатковий об'єкт, так званий

асоціативний об’єкт, який фіксує два екземпляри (по одному для кожного з об’єктів), котрі беруть участь у зв’язку. Такий об’єкт, окрім своєї назви, має першим атрибутом ідентифікатор першого з пов’язаних екземплярів об’єктів, а другим атрибутом – ідентифікатор екземпляра другого.

Приклади зображення зв’язків *а, б, в* наведено на рис. 4.1.

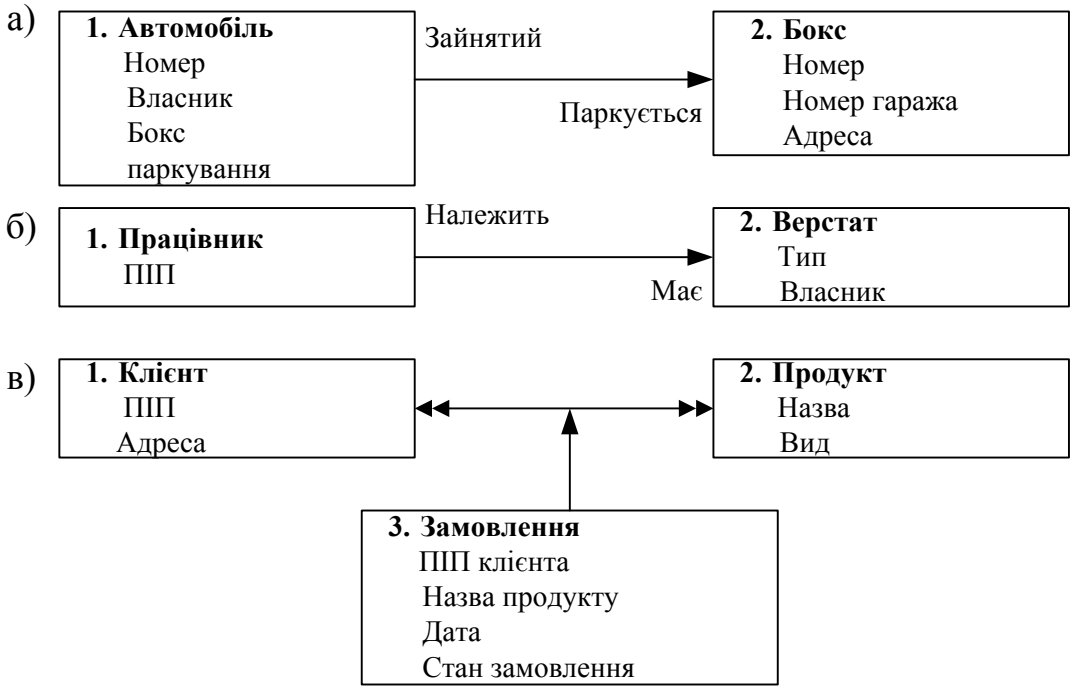

Рисунок 4.1 – Приклади зображення зв’язків об’єктів

Для певних завдань зв’язки між об’єктами можуть еволюціонувати із часом, і фаза еволюції може суттєво впливати на хід виконання відповідного завдання. Для таких видів зв’язку обов’язково будується асоціативний об’єкт, для якого визначається модель станів (див. п. 3.4.2). Для подання стану асоціативного об’єкта до складу його атрибутів додається атрибут, який фіксує його поточний стан. Нехай для прикладу на рис. 4.1, *в* замовлення може перебувати в станах: "отримано", "готується", "чекає розрахунку", "сплачено", "готове" тощо, тоді до атрибутів асоціативного об’єкта додається, крім зазначених на рис. 4.1, в, ще й атрибут "стан виконання замовлення", який у певні моменти часу може приймати одне з наведених вище значень. Зазначимо, що серед дій, котрі супроводжують переходи в стани для моделі станів зв’язків, мають бути операції створення нового екземпляра асоціативного об’єкта (коли нова пара екземплярів вступає у зв’язок) та його знищення (коли зв’язок переривається).

4.2.2 Уточнення інтерфейсів

Узгодження інтерфейсів з потенціальними користувачами системи на ранніх стадіях життєвого циклу розробки має на меті вирішення двох завдань. По-перше, допомогти користувачеві перевірити його розуміння системи і отримати його схвалення складових мети та функцій цієї системи. По-друге, дозволити розробникові впевнитися, що запропоновані ним правила взаємодії користувача та системи задовольняють обидві сторони з погляду взаєморозуміння, ефективності і швидкості сприйняття та реагування.

Організація інтерфейсів базується на певних ключових елементах, визначення яких має передувати проектуванню конкретних екранів та форматів обміну даними. Це такі елементи:

а) метафори, поширені в домені користувача. Під цим терміном розуміємо значущі терміни, образи та поняття, які є для нього знаними й зрозумілими або вивченими;

б) ментальна модель організації й подання даних, функцій та ролей;

в) правила навігації (перегляду) даних, функцій та ролей;

г) візуальні прийоми демонстрації перед користувачем елементів, визначених у п. 1—3;

д) методи взаємодії, які прогнозуються як відповідні уподобанням користувачів майбутньої системи.

У визначенні наведених елементів мають братися до уваги аспекти культури, до якої належать потенціальні користувачі системи, як-от норми, традиції, звичаї та міфи домену, й притаманні його дійовим особам критерії уподобань.

Перелічені елементи використовуються при проектуванні конкретних інтерфейсів.

Окремо наголосимо на необхідності використання в меню та в іконках інтерфейсів мовних форматів і правил їхніх трансформацій, обчислень валютних одиниць, метричних показників систем вимірювання (гривні, долари, рублі, метри, дюйми, бушелі тощо), які є звичними для кола користувачів системи, що будується. Вони створюють для користувачів певний психологічний комфорт.

Якщо необхідно зробити систему "полікультурною", тобто здатною до адаптації, необхідно текстуально виділити чутливі до культурного середовища елементи, які потребують заміни: іконки, звукові повідомлення, тексти. Слід пам’ятати, що довжина текстів суттєво залежить від лаконізму обраної мови спілкування (серед них англійська, мабуть, найбільш лаконічна). Тому довжина текстових повідомлень та вікон вводу текстів має бути параметром настроювання.

Зазначимо, що правила навігації мають враховувати традиції читання (зліва направо або навпаки).

Як бачимо, питань виникає багато, тому загальною рекомендацією є побудова всіх екранних та друкованих форм системи й "програвання" з користувачем різних їхніх варіантів, щоб обрати ті, котрі відповідатимуть його уподобанням. При цьому зазвичай доводиться робити вибір між різними несумісними характеристиками інтерфейсу, як, наприклад, зручність доступу та забезпечення конфіденціальності, швидкодія та складність оброблення, легкість сприйняття повідомлень (об'ємна графіка, звуковий супровід тощо) та вартість розробки.

Для побудови інтерфейсів є широкий вибір методів і засобів. Більшість з них базується на фіксації певних класів об'єктів інтерфейсу (вибір з меню, заповнення екранних форм, пряме маніпулювання – так званий стиль "зачепи та підтягни") та на засобах монтування їх у програмну систему як інтегрованих з нею блоків або автономних підсистем.

На закінчення нагадаємо, що ми вели мову про інтерфейси об'єктів, які було визначено під час аналізу вимог і зафіксовано у відповідних моделях. Інтерфейси об'єктів означають операції, які може виконувати об'єкт, та повідомлення, які він може надсилати або отримувати.

4.2.3 Уточнення функцій оброблення даних

Для зафіксованих у моделях вимог об'єктів уточнюються склад і зміст властивих їм операцій (методів) і уточнюються схеми взаємодії об'єктів.

Зміст операцій, які здатні виконувати об'єкти, може бути розкрито за допомогою діаграм потоків даних для кожної з операцій (див. п. 3.4.3).

Взаємодія об'єктів організовується шляхом обміну повідомленнями, у відповідь на які об'єкти виконують відповідні операції і змінюють свій стан (див. п. 3.4.2) або посилають повідомлення іншим об'єктам. Для уточнення поведінки об'єктів можна рекомендувати використання моделей у вигляді діаграм, котрі відображають аспекти взаємодії об'єктів. Такі діаграми входять до складу методу UML, про який ми згадували у п. 3.3 і який детальніше обговорюється в розділі 5. Зокрема, моделі поведінки об'єктів викладено в п. 5.4.

Вочевидь, усі уточнення, зроблені щодо даних, інтерфейсів та поведінки об'єктів сценарію можуть привести до необхідності перегляду моделей аналізу вимог або навіть і складу об'єктів. Важливо наголосити, що всі необхідні корекції слід починати з корекції продуктів етапу інженерії вимог – моделі вимог, моделі аналізу вимог та інших, причому витрати на пошук місць локалізації потрібних корекцій у згаданих моделях тим менші, чим повніше забезпечується трасування вимог.

4.2.4 Уточнення нефункціональних вимог

Вимоги, які називають нефункціональними, відображають здебільшого певні обмеження, накладені організацією або середовищем використання системи (див. п. 3.1). Різновидів нефункціональних вимог досить багато, але, зважаючи, що вони пов’язані з багатьма застосуваннями комп’ютерних систем і для них розроблено чимало готових рішень, є сенс вивчити можливість використання цих рішень у проекті, що розробляється. Можна стверджувати, що для різновидів нефункціональних вимог завдання їхньої реалізації становлять окрему спеціальну проблемну галузь, в моделюванні якої може бути застосовано ті самі методи, котрі було запропоновано в попередніх розділах цієї книги для моделювання доменів проблемних галузей прикладних застосувань. Серед таких назвемо вимоги секретності, вимоги безпеки, відмовостійкість, корпоративну роботу над спільними ресурсами тощо.

Будуючи моделі вимог для зазначених вище доменів, слід мати на увазі, що фактично вони використовуються у багатьох прикладних застосуваннях і можуть розглядатися як незалежні від прикладних застосувань автономні аспекти розгляду систем програмування. Для них напрацьовано чимало національних, корпоративних та відомчих стандартів, які, зокрема, фіксують відповідні онтології, можливі стимули та стани тощо. Тому, починаючи моделювання цих аспектів, треба дотримуватися відповідних стандартів. Це не лише дозволить зекономити зусилля з моделювання, а й створить передумови для використання готових програмних продуктів на подальших етапах життєвого циклу розробки. Детальніше про використання готових напрацювань у створенні програмних систем див. розділі 7.

Результатом уточнення згаданих типів нефункціональних вимог має бути розширення напрацьованих на етапі інженерії вимог моделей специфічними доповненнями як відповідних об’єктів, їхніх операцій чи зв’язків.

4.3 Архітектурне проектування

Архітектурне проектування полягає у визначенні головних структурних особливостей системи, яку будують, а саме: складу компонент, способів їхньої композиції, обмежень на їхні з’єднання.

Сучасні програмні системи – це досить складні композиції різних функцій, яким відповідають програмні модулі. Водночас є тисячі готових програмних продуктів, котрі можна включити в будь-яку програмну систему для виконання чітко визначених функцій, при цьому примітивні функції можуть складати композиції, які виконують певні узагальнені функції, ті, в свою чергу, можуть пов’язуватися в нові композиції тощо.

Для того, щоб сукупність готових до використання засобів можна було переглянути й зрозуміти, введено певну пошарову їхню структуризацію (рисунок 4.2).

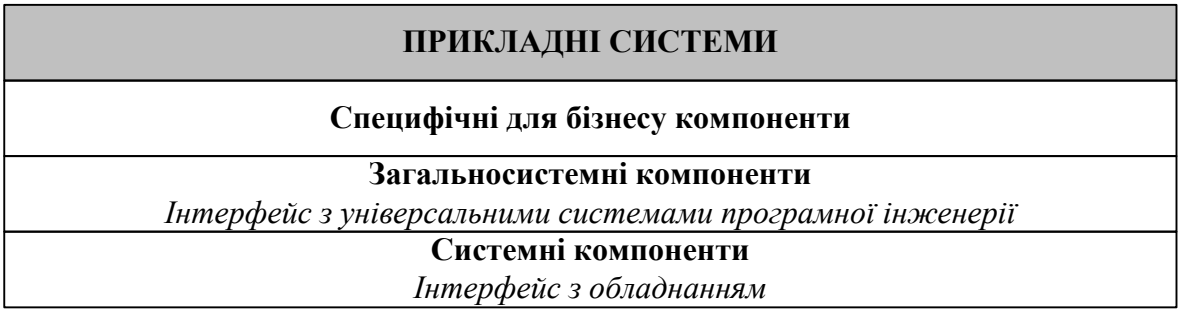

Рисунок 4.2 – Пошарова архітектура напрацювань у програмній інженерії

До першого, нижчого шару відносять системні компоненти, котрі здійснюють організацію взаємодії з так званими периферійними пристроями комп'ютерів (принтери, клавіатура, сканери, маніпулятори тощо). Вони здебільшого використовуються при побудові операційних систем і не потрапляють у поле зору розробників прикладних застосувань.

До другого шару відносять так звані загальносистемні компоненти або посередники, котрі забезпечують взаємодію прикладних застосувань з універсальними сервісними системами, з такими, як операційні системи, системи баз даних та знань, системи керування мережами тощо. Компоненти цього шару використовуються в багатьох прикладних застосуваннях як складові компонент прикладних програмних систем.

До третього шару відносять специфічні для певної проблемної галузі й залежні від неї компоненти, які може бути використано як складові для спектра програмних систем, призначених для розв'язання задач означеної галузі (так званої сім'ї програмних систем).

Нарешті, до четвертого шару відносять програмні системи, побудовані для вирішення конкретних задач конкретних груп споживачів інформації, заради яких, власне, і створено компоненти всіх інших шарів.

Компоненти кожного з поданих шарів використовуються, зазвичай, тільки в своєму шарі та в наступному (вищому шарі). Для кожного шару на сьогодні визначено відповідний набір професійних знань, умінь та навичок для створення й використання його компонент, що, певною мірою, визначає відповідне розшарування професіоналів у програмній інженерії.

Ведучи мову про архітектурне проектування програмних систем, ми будемо розглядати переважно бачення програмної системи як композиції компонентів третього шару, тоді як використання компонентів другого шару є предметом розгляду технічного й детального проектування (див. нижче).

Ми отримали продукт етапу інженерії вимог як сукупність об'єктів,

котрі належать до певного сценарію і взаємодія яких реалізує потрібні функції цього сценарію. Спробуємо з'ясувати, чи можна вважати об'єднання сукупностей об'єктів для всіх визначених сценаріїв системи складовими архітектури цільової системи? Інакше кажучи, чи можемо ми вважати, що такі об'єкти належать то третього шару компонент і їхня композиція є наочним представленням архітектури системи.

Відповідь на ці запитання негативна з таких міркувань: для складних систем кількість виділених об'єктів може налічувати сотні, і їхня композиція не буде мати виразного та зрозумілого подання, навіть з урахуванням тієї обставини, що об'єкти, багатьох сценаріїв можуть збігатися, тому буде потрібний додатковий аналіз для ототожнення їх.

Згадаємо основні принципи, напрацьовані як рекомендації для декомпозиції складної системи на компоненти або модулі:

– для компоненти має бути чітко визначена мета, щоб можна було перевірити, чи вона її виконує;

– для компоненти має бути чітко визначено всі її входи та виходи;

– компоненти мають утворювати ієрархію, кожний рівень якої відповідає рівню абстракції розгляду системи і дозволяє приховувати певні деталі, які буде відпрацьовано на наступних рівнях. Така покрокова деталізація прийняття рішень не стільки розподіляє вирішення складного завдання на кілька вирішень простіших завдань, скільки дозволяє відкласти детальні розв'язання проблем, щоб зосередитися на розв'язанні загальних рішень;

– робота над компонентами може вестися окремими членами команди із застосуванням кількох інструментальних засобів для кількох компонент, що суттєво впливає на ефективність роботи; але при цьому інтерфейси між компонентами мають бути прозорими й узгодженими, щоб інтеграція компонент в єдину структуру була можливою і базувалася на спільному розумінні проблеми. При цьому ключова якість об'єктного підходу – інкапсуляція внутрішніх дій і приховування всіх подробиць, які не стосуються правил використання компоненти – має діяти і для підсистеми як композиції об'єктів.

Враховуючи зазначене вище, можна прийти до висновку, що отримані нами сукупності об'єктів доцільно об'єднувати в підсистеми. При цьому необхідно керуватися такими міркуваннями:

а) кожна створювана підсистема має асоціюватися з певними елементами продукту інженерії вимог (як, наприклад, актор, сценарій, об'єкт тощо);

б) доцільно необов'язкові функції або часто змінювані функції виділяти як підсистеми, при цьому бажано кожну функцію, для якої прогнозуються зміни вимог, виділяти як окрему підсистему, пов'язану з одним актором (бо зміни найчастіше викликаються актором). Те саме можна рекомендувати і для функцій, використання яких у системі

необов’язкове (довільне) і його може не бути залежно від обставин використання системи;

в) інтерфейс підсистеми тим більше прозорий і зрозумілий, чим менше вона має взаємозв’язків з іншими підсистемами. Бажано, щоб кожна підсистема виконувала мінімум зрозуміло визначених послуг або функцій (в ідеалі – тільки одну) та мала фіксовану множину чітко визначених параметрів інтерфейсу.

Можна перелічити типи зв’язків, характерних для об’єктів:

– зв’язок за внесенням змін, коли зміна одного об’єкта потребує перегляду другого або обидва об’єкти залежать від зміни третього (наприклад, об’єкт-сутність та об’єкт управління залежать від об’єкта-інтерфейсу);

– зв’язок за управлінням, коли керований об’єкт не може виконувати свої функції без повідомлень керуючого або один об’єкт стимулює виконання операцій другого;

– зв’язок за даними, коли дані (атрибути) одного об’єкта використовуються другим. При цьому можуть передаватися тільки значення даних або, поряд зі значеннями даних, передається метаінформація щодо організації їх, необхідна для правильної інтерпретації даних.

Відокремивши змінювані й довільні підсистеми, проведемо аналіз зв’язків та залежностей, які є між об’єктами, що залишилися, з метою утворення підсистем з тісними внутрішніми зв’язками між об’єктами та прозорими зовнішніми інтерфейсами.

Способи об’єднання об’єктів у підсистему можна кваліфікувати так:

– зернисте поєднання – в підсистему об’єднуються об’єкти, які нічим не пов’язані між собою (відповідну підсистему створено для простого укрупнення компонент архітектури);

– логічне поєднання – в підсистему об’єднуються об’єкти, які є функціонально незалежними, але мають якусь спільну властивість, або для яких можна встановити певне логічне відношення (наприклад, ту саму функцію реалізовано для багатьох середовищ, як-от введення даних для дисків та портів мережі або різних типів даних, як, наприклад, цілого або комплексного);

– об’єднання за часом – у підсистему об’єднуються незалежні об’єкти, які активізуються в спільний проміжок часу;

– комунікативне об’єднання – в підсистему об’єднуються об’єкти, які мають спільне джерело даних;

– процедурне об’єднання – в підсистему об’єднуються об’єкти, які послідовно передають одне одному керування;

– функціональне об’єднання – коли кожний з об’єктів, що входить у підсистему, виконує частину робіт для здійснення загальної функції, яку

виконує підсистема, тобто всі об’єкти виконують спільне завдання.

Ми неодноразово підкреслювали, що розробляючи систему, слід постійно пам’ятати тезу: "Всяка зроблена система з часом потребує змін". Якщо проаналізувати наведені вище способи поєднання об’єктів у підсистеми з погляду стійкості до змін, коли кожна зміна вимоги потребує відповідної корекції мінімальної кількості архітектурних компонентів, то можна зробити висновок, що всі способи 1–5 не сприяють полегшенню модифікації вимог. Що ж до функціонального поєднання, то якщо ціль, яку реалізує таке поєднання, відповідає певним вимогам у моделі вимог, трасування вимог у моделі проекту можна вважати досягнутим.

Якщо в новостворюваній системі передбачається використання готових систем (так званих успадкованих систем), їх доцільно вважати підсистемами новостворюваної системи.

Використання готових компонент або спільних компонент для підсистем потребує спеціального розгляду, який буде подано у розділі 7.

Архітектурне проектування може потребувати перегляду моделі аналізу вимог. Наприклад, якщо поведінка певного об’єкта частково використовується в кількох підсистемах, таку частину доцільно виділити в окремий об’єкт або навіть у підсистему.

Виділення підсистем для дуже великих проектів є досить складною роботою і може вестися з урахуванням дещо інших критеріїв. Наприклад, якщо розроблення ведуть декілька груп різних рівнів компетентності або різних рівнів забезпеченості ресурсами, або роз’єднаних географічно, розподілення на підсистеми може вестися з пріоритетним урахуванням обставин, згаданих вище. Аналогічно при наявності розподіленого устаткування кожний логічний вузол може бути асоційовано з підсистемою.

Нотації для архітектурного проектування розглядаються в п. 5.10.

4.4 Технічне проектування

Технічне проектування полягає у відображенні вимог середовища функціонування і розроблення системи та визначенні всіх конструкцій як композицій компонентів. На цьому етапі відбувається прив’язка проекту до технічних особливостей платформи реалізації, СУБД, організації комунікацій, наявності фактора реального часу, виконавських вимог, таких, як швидкість реагування системи на зовнішні стимули тощо.

Об’єкти моделі аналізу вимог погоджуються з урахуванням перелічених вище особливостей, формалізуються всі стимули, які посилає чи отримує об’єкт, і всі операції, що є відповіддю на зазначені стимули.

Кожний з наведених вище аспектів прив’язки може потребувати побудови допоміжних інтерфейсних або керуючих об’єктів чи корекції існуючих. До того ж може виявитися можливість використання готових

підсистем, чий устрій дещо відрізняється від підсистем, які були досі визначені на основі аналізу вимог. Тоді вносяться відповідні корективи до моделі аналізу вимог та архітектури системи.

Наступним кроком проектування може бути врахування певних властивостей, які зазвичай належать до так званих показників якості. Зупинимося на деяких з них.

Надійність функціонування (operational reliability). Надійність функціонування системи можна значно підвищити, якщо передбачити і відпрацювати виняткові ситуації під час роботи системи. Тестування системи проводиться, щоб переконатися, що реалізація системи відповідає висунутим до неї вимогам. Але вимоги здебільшого обумовлюють, що має робити система, тоді як важливо також обумовити, чого вона не має робити. Одним із шляхів для цього є явна фіксація ситуацій, які унеможливлять правильну роботу системи (так звані виняткові ситуації).

Причинами виникнення виняткових ситуацій можуть бути: помилки користувача при зверненні до системи чи під час підготовки даних; непередбачені збіги обставин функціонування системи (невиявлені під час тестування помилки проектування); випадкові збої обладнання тощо. При цьому система може реагувати по-різному: відмовитися виконувати певну послугу, виконати її помилково або зруйнувати якісь дані. Вочевидь, для другої і третьої з перелічених реакцій неможливо передбачити наслідки, тоді як для першої можна докласти певних зусиль, щоб відновити роботоздатність системи, наприклад, виконати один з наведених нижче варіантів робіт:

– відновити попередній стан системи (що передував винятковій ситуації) і спробувати застосувати іншу стратегію виконання послуги;

– відновити попередній стан системи, внести необхідні корективи і повторити виконання послуги із старою стратегією;

– відновити попередній стан системи, сформувати повідомлення про помилку й зупинити систему в очікуванні реакції користувача.

Ми бачимо, що забезпечення надійності системи вимагає для кожної її послуги передбачення виняткових ситуацій, аналізу їхніх причин та наслідків, побудови механізму відтворення попереднього стану (для чого необхідна певна стратегія запам’ятовування поточного стану системи) та виправлення ситуації. Для забезпечення таких дій застосовуються типові засоби:

– подвійне обчислення й порівняння результатів або їхніх контрольних сум, у тому числі виконаних на різних процесорах;

– таймери, що визначають часові інтервали фіксації поточного стану;

– додаткові перевірки коректності даних, які передають актори чи зовнішні системи або окремі компоненти однієї системи.

Усі зазначені вище дії втілюються в додаткових компонентах

проекту і призводять до додаткових витрат на певні надмірні перестороги, які є ціною за надійність функціонування системи. Доцільність таких витрат визначається виключно специфікою систем. Якщо наслідки помилок незворотні, наприклад, як у систем підвищеного ризику (космічні та ядерні системи, моніторинг хворих, керування реальними об'єктами), доводиться йти на дублювання процесів і додаткові перевірки. Іноді навіть використовують паралельну роботу кількох команд або взаємну перевірку комп'ютерів, що працюють паралельно, коли один з них здійснює моніторинг іншого. Так, кажуть, що для керування системою ШАТЛ сім комп'ютерів дублюють один одного.

Переносимість системи (system portability). Під таким терміном розуміють можливість змінювати певні використовувані сервісні системи (операційні системи, системи комунікацій у мережах, СУБД тощо) шляхом локального настроювання відповідних модулів. Зазвичай йдеться про переносимість щодо конкретного типу сервісних систем, наприклад, переносимість щодо СУБД, переносимість щодо системи файлів тощо. Для реалізації таких властивостей визначаються об'єкти, які взаємодіють з типом сервісних систем, щодо якого декларується переносимість. Кожний з визначених у такий спосіб об'єктів замінюється на такий, що взаємодіє не безпосередньо із сервісною системою, а з якимсь абстрактним об'єктом-посередником, котрий здійснює трансформацію абстрактного інтерфейсу в інтерфейс конкретної сервісної системи. Об'єкт-посередник при цьому має властивість настроюватися на конкретну сервісну систему.

Так, на рис. 4.3 об'єкт-посередник має призначені для нього операції створення файлу, читання з файлу, записування у файл, модифікації записів файлу, знищення файлу.

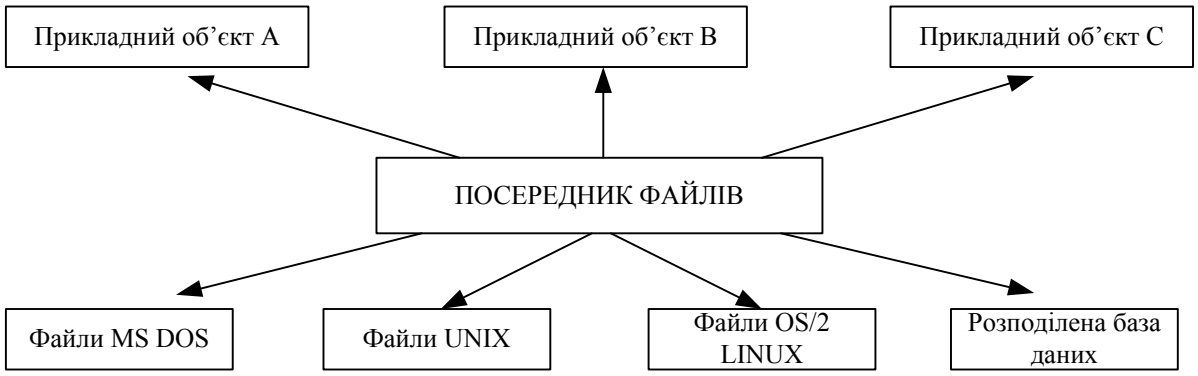

Рисунок 4.3 – Об'єкт-посередник

При цьому об'єкти прикладної системи, які звертаються до цього об'єкта за виконанням певних послуг для роботи з файлом, не опікуються подробицями організації файлу в конкретному середовищі його реалізації. Такі подробиці інкапсулює в собі об'єкт-посередник, який, залежно від

настроювання, звертається при цьому до системи файлів MS-DOS або UNIX. Таким чином, точку можливих варіацій відносно використання тієї чи іншої системи управління файлами чітко локалізовано, що дає нам право стверджувати про забезпечення стійкості системи, що будується, відносно зміни системи даного сервісу (управління файлами).

Подібні об’єкти-посередники доцільно проектувати для будь-якої передбачуваної можливості зміни вимог, яку не можна реалізувати простою заміною значень певних параметрів.

Нотації для подання продуктів проектування. Продукти проектування подаються переважно в нотаціях, які базуються на моделях аналізу вимог, більшість розглянутих нами діаграм (таких, як діаграми сутності-зв’язку, діаграми переходів у стани, діаграми потоків даних дій, діаграми класів тощо) активно використовуються як нотація для продуктів проектування. Але для них у зазначених діаграмах задіяно об’єкти проекту, що детальніше відображають не лише вимоги до розробки, а й рішення, які сприяють втіленню цих вимог. Звісно, нотації, запропоновані авторами різних методів об’єктно-орієнтованого аналізу та проектування, мають свою специфіку подання наведених вище діаграм.

Досить виразний набір діаграм для моделювання проекту має вже згадуваний метод UML подання у розділі 5.

На рис. 4.4 подано зображення класу об`єкта *дата* за методом С. Шлеєр та С. Меллора.

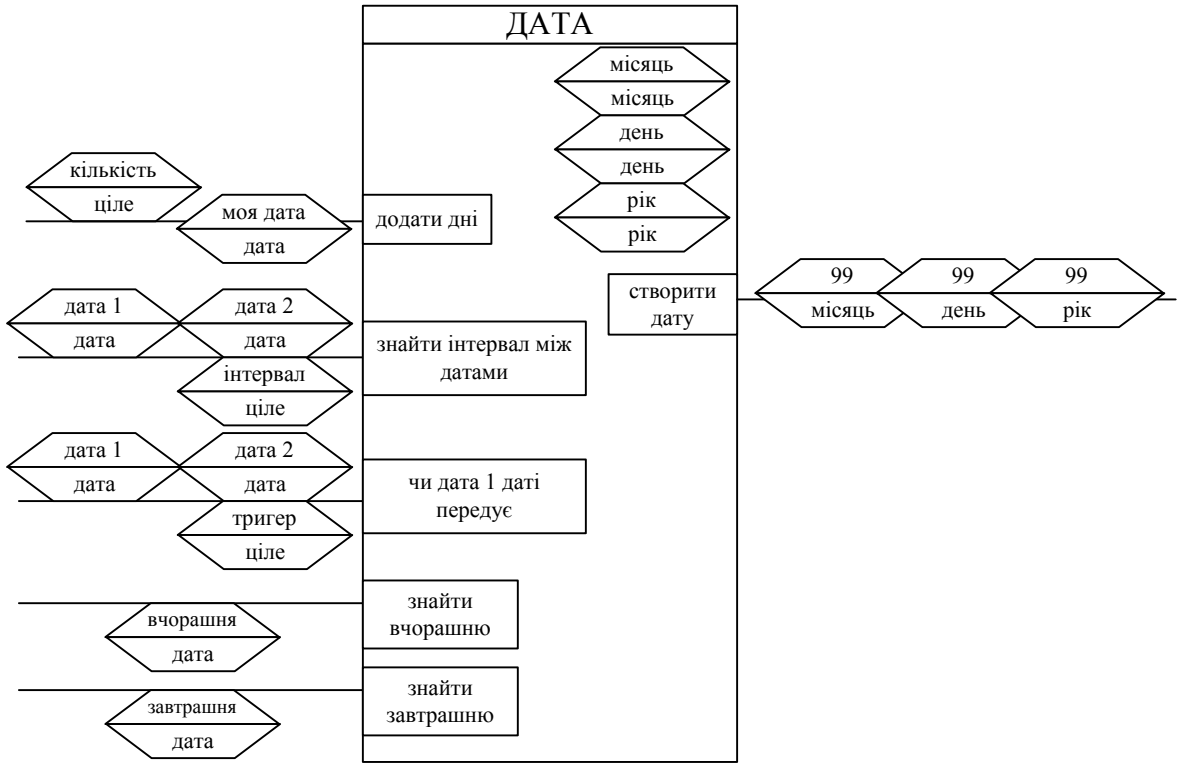

Рисунок 4.4 – Діаграма класу *дата*

Клас позначається прямокутником, під верхньою лінією якого пишеться його ім’я. Шестикутниками всередині прямокутника зображено атрибути класу, кожен шестикутник поділено горизонтальною лінією, над нею зазначається ім’я атрибута, а під нею – тип, до якого він належить. Так само позначені параметри зовнішніх операцій, які може виконувати клас. Самі операції позначаються прямокутниками всередині прямокутника класу. До кожної операції зовні веде лінія, над лінією розміщуються шестикутники вхідних параметрів відповідної операції, під лінією – вихідні, а на лінії – вхідні – вихідні.

Подана на рис. 4.4 діаграма класу визначає, що клас дата має атрибути день, місяць, рік і може виконувати такі операції: створити дату, додати до дати дні, знайти інтервал (у днях) між двома датами, знайти вчорашню або завтрашню дату, перевірити, котра з двох дат передує іншій. Для всіх наведених операцій на діаграмі визначено вхідні та вихідні параметри.

Зазначимо, що атрибут місяць визначає можливі значення від 1 до 12, атрибут день – від 1 до кінцевої дати кожного місяця, атрибут рік – чотири цифри року.

Нотація проектування за методом І. Джекобсона фактично увійшла як складова до методу UML, який буде подано в розділі 5.

Контрольні запитання і завдання

1. Визначте завдання етапу проектування програмного забезпечення.
2. Опишіть процеси етапу проектування.
3. Сформулюйте завдання концептуального проектування.
4. Які є засоби матеріалізації зв’язків у логічних структурах даних?
5. Перелічіть ключові чинники, котрі впливають на проектування інтерфейсів.
6. Назвіть нефункціональні вимоги, які треба врахувати на стадії проектування.
7. Які шари може бути виділено в сучасній архітектурі програмного забезпечення?
8. Якими аргументами треба керуватися при об’єднанні фрагментів програмного забезпечення в системи?
9. Які способи об’єднання об’єктів у системи Ви знаєте?
10. Опишіть процеси забезпечення надійності функціонування системи.
11. Які є способи забезпечення переносимості системи?
12. Які нотації використовують для подання продуктів проектування?

5 МЕТОД UML ЯК ПОТЕНЦІЙНИЙ СТАНДАРТ ЗАСОБІВ МОДЕЛЮВАННЯ В ПРОГРАМНІЙ ІНЖЕНЕРІЇ

5.1 Концепція методу

Метод UML [1] (Unified Modeling Language — уніфікована мова моделювання) є рідкісним прикладом плідної кооперації групи (G. Booch, I. Jacobson, J. Rumbaugh) провідних спеціалістів з програмної інженерії і авторів відповідних методів інженерії вимог, що набули значного визнання і широко застосовуються. Дослідивши всі переваги власних пропозицій і широкий спектр конкуруючих, вони інтегрували свої зусилля, створивши новий метод моделювання, якому дали цитовану вище назву UML. UML став базовим для багатьох провідних розробників програмного забезпечення, і тепер експерти прогнозують, що він набуде статусу міжнародного стандарту як метод моделювання продуктів усіх стадій життєвого циклу розробки програмних систем.

Автори визначають свій метод як мову для специфікації, візуалізації, конструювання й документування *артефактів (artifacts)* програмних систем, а також для моделювання бізнесу.

В основу методу покладено парадигму об’єктного підходу, за якою концептуальне моделювання проблеми (дивись п. 3.2) відбувається в термінах взаємодії об’єктів:

– онтологія домену визначає склад класів об’єктів домену, їхніх атрибутів та взаємовідношень, а також послуг (операцій), які можуть виконувати об’єкти класів;

– модель поведінки визначає можливі стани об’єктів, інциденти, що ініціюють переходи з одного стану в інший, повідомлення, які об’єкти надсилають одне одному;

– модель процесів визначає дії, які виконують об’єкти.

Як і в попередньо розглянутих методах, автори UML декларують, що, враховуючи складність проблеми концептуального моделювання, її не можна розв’язати єдиною нотацією. Концептуальна модель вимог пропонується як сукупність нотацій, переважно діаграм, котрі є візуалізацією подання основних елементів системи в моделі. Кожна з діаграм демонструє певну підмножину інформації, яка деталізує елементи, що являють собою певний аспект опису моделі та його семантику.

В комплексі сукупність включених до методу діаграм відображає найважливіші випадки функціонування системи. Перелічимо їх:

а) діаграми класів;

б) діаграми сценаріїв;

в) діаграми поведінки об’єктів, а саме:

1) діаграми послідовності;

2) діаграми співробітництва;

3) діаграми активності;

4) діаграми станів;

г) діаграми реалізації, а саме:

1) діаграми компонент;

2) діаграми розміщення.

Автори не закріплюють діаграми жорстко за окремими етапами життєвого циклу розробки, більшість діаграм може відображати кількома ступенями подробиць об'єкти кількох етапів: об'єкти аналізу вимог, проекту, реалізації тощо. Для цього передбачено можливість вказувати або замовчувати (залежно від стадії розроблення) окремі подробиці визначення.

Кожний вид діаграм відображає різні перспективи бачення й розуміння моделі. У моделі може бути по кілька діаграм кожного з описаних видів.

Оскільки одним з головних завдань методу є досягнення взаєморозуміння між учасниками розробки, спеціальну увагу приділено коментарям. Допускається два види коментарів. Перший з них – це традиційний неформальний текст, який може бути розміщено будь-де на діаграмі в рамці з відігнутим кутом, як на рис. 5.1.

Рисунок 5.1 – Коментар UML

Другий тип коментарю названо стереотипом. *Стереотип (stereotype)* – це засіб метакласифікації елементу в UML. Він є специфічним стандартизованим коментарем щодо категорії елементу, поданого на будь-якій діаграмі, своєрідним ярликом, який характеризує зміст елементу діаграми або його призначення.

Стереотипи зображаються на діаграмах своєю назвою, яка наводиться в подвійних кутових дужках, наприклад, «суперклас», «площа», «ініціація».

Певні стереотипи є фіксованими в UML і мають стандартні назви, як-от: «актор», «система», «підсистема», «подія», «виняткова ситуація», «інтерфейс», «метаклас», «послуга» (утиліта) та ін.

Окрім зафіксованих в UML, дозволяється визначати стереотипи за власною потребою розробника. Такі стереотипи позначають елементи моделей, типові для певного домену застосування, як, наприклад, «вимірювач», «контролер», «рахунок», «запас» тощо. Вони можуть вважатися інструментом розширення й адаптації UML до конкретних

доменів застосувань, призначених суттєво полегшити розуміння відповідних моделей.

Зупинимося на змісті окремих діаграм детальніше.

5.2 Діаграми класів

Така діаграма відображає онтологію домену і за змістом еквівалентна інформаційній моделі за методом С. Шлеєр та С. Меллора (див. п. 3.4.1): визначається склад класів об’єктів як базових абстракцій та їхні взаємовідносини. Причому нотація для опису класів забезпечує відокремлення опису функцій від опису даних, застосування принципів інкапсуляції і наслідування даних.

Діаграма має вигляд символів класів — так званих іконок та зв’язків між ними. Терміном іконка позначають стандартизоване, фіксованої форми, візуальне зображення (так би мовити, ієрогліф) певного поняття, яке легко розпізнається. Іконка класу має форму прямокутника, який може поділятися на дві або три частини. Верхня його частина обов’язкова, вона містить ім’я класу. Друга й третя частини прямокутника можуть наводитися або пропускатися і містять: друга – список атрибутів класу, третя – список операцій класу (див. рис. 5.2).

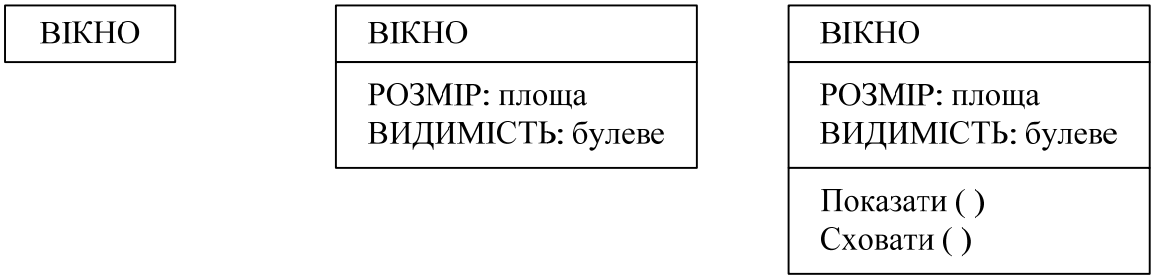

Рисунок 5.2 – Приклади подання класу

Інакше кажучи, діаграма класу може відображати лише імена класів або імена та відповідні атрибути класів, або імена, атрибути та операції (методи) класів.

Атрибути можуть бути такими, типи значень яких вважаються наперед визначеними в UML, як-от: розмір, площа, кут, видимість. Останній атрибут може мати такі значення:

– спільна (public) означає, що операцію класу можна викликати з будь-якої частини програми будь-яким об’єктом системи;

– захищена (protected) означає, що операцію можна викликати тільки об’єктом того класу, в якому її визначено, або його спадкоємцями;

– приватна (private) означає, що операцію можна викликати тільки об’єктом того класу, в якому її визначено.

Водночас користувач може визначати специфічні для нього

атрибути.

Операція – це сервіс, який може надавати екземпляр класу, якщо буде відповідний виклик. Операція має назву і список аргументів.

На діаграмі може бути показано не лише класи, а й окремі їхні екземпляри. Може бути побудовано діаграму екземплярів класів. З метою відрізнити класи від їхніх екземплярів назви других у зображенні іконки класу підкреслюються. Наприклад, на рис. 5.3, *а* зображено клас, а на рис. 5.3,*б* – його екземпляр.

а) *б*)

Рисунок 5.3 – Зображення класу (*а*) і екземпляру (*б*)

Класи можуть перебувати у певних відношеннях або зв’язках. Розглядаються бінарні асоціації, в яких об’єкт з кожної сторони відіграє свою роль (див. рис. 5.4).

Рисунок 5.4 – Ролі об’єктів в асоціаціях

Для окремих типових зв’язків семантика вважається встановленою. Це такі.

Асоціація – взаємна залежність між об’єктами різних класів, кожен з яких є рівноправним членом залежності. Для асоціації може позначатися кількість екземплярів об’єктів кожного класу, які беруть участь у зв’язку (0 – якщо жодного, 1 – якщо один, * – якщо багато). Можуть вказуватися мінімальна й максимальна кількість, наприклад, 0,1...* означає, що на відповідному кінці асоціації може не бути жодного екземпляра, бути один або багато. Приклад асоціації наведений на рисунку 5.5.

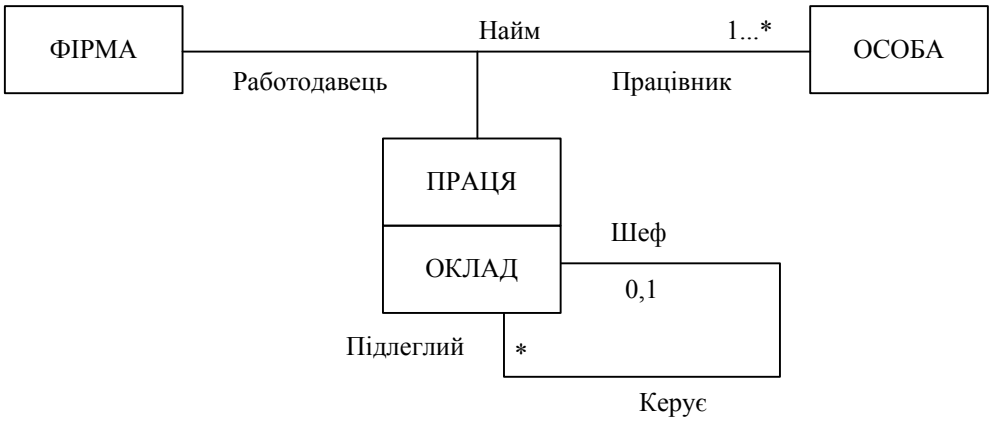

Рисунок 5.5 –Асоціація

Агрегація або відношення частина-до-цілого. Особливість цього відношення полягає в тому, що час існування об’єкта-частини збігається з часом існування об’єкта-цілого. Стрілка з ромбом на кінці, яка позначає відношення агрегації, спрямована від об’єкта-частини до об’єкта-цілого, як це подано на рис. 5.6.

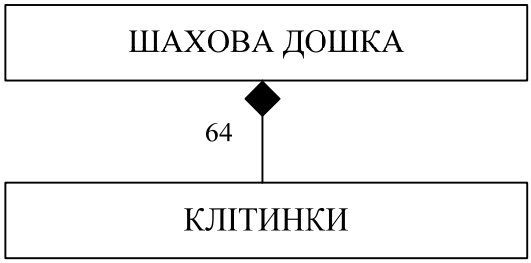

Рисунок 5.6 – Відношення агрегації

Наслідування підкласом властивостей суперкласу може мати позначку "один до багатьох". Різновидами наслідування можуть бути відношення узагальнення й спеціалізації. Приклади наведені на рис. 5.7.

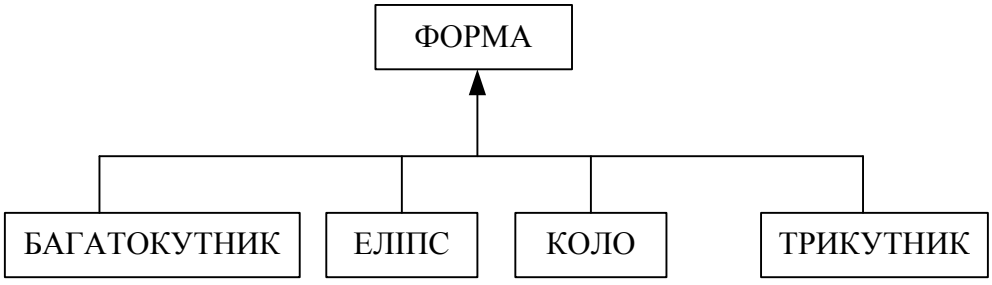

Рисунок 5.7 – Відношення узагальнення

Альтернативна асоціація. Деякий клас одночасно може перебувати у зв’язку тільки з одним елементом певної множини класів. Можливі альтернативи позначаються тим, що відповідні їм дуги перетинаються пунктирною лінією з позначкою {or} (або), як це показано на рис. 5.8.

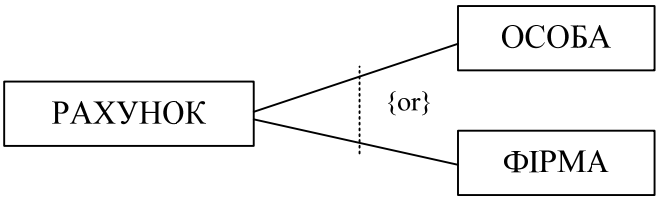

Рисунок 5.8 – Альтернативна асоціація

Залежність класів (class relation). Є багато видів залежностей між класами: деякий клас-клієнт може використовувати певний сервіс (операцію) іншого класу; класи можуть бути пов'язані відношенням трасування, коли один трансформується в другий унаслідок певного процесу життєвого циклу, наприклад, клас аналізу перетворюється в клас проекту, а потім у клас реалізації. Один клас може бути уточненням другого, як на рис. 5.9.

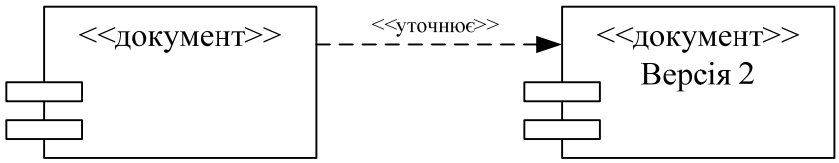

Рисунок 5.9 – Асоціація залежностей

Екземпляризація (specimenation). Це залежність між параметризованим абстрактним класом-шаблоном (template) і реальним класом, який ініційовано шляхом визначення параметрів шаблону. Прикладом параметризованих класів є контейнерні класи для мови програмування C++. На діаграмі класів параметризований клас позначається так: на рамці іконки класу зверху праворуч зображається штрихами маленький прямокутник, всередині якого подаються назви формальних параметрів шаблона, як на рис. 5.10, де *T* є параметр, що визначає тип елементу множини. На ім'я класу, який створюється внаслідок визначення параметрів шаблону, посилаються в кутових дужках після імені параметризованого класу-шаблону як префікса. Детальніше параметризовані абстрактні класи розглядаються в п. 12.3.3.

Діаграма класів може належати до екземплярів класу, суперкласів (абстрактних класів) або підкласів (конкретних класів). У кожному з конкретних прикладів на іконці класу перед його назвою зазначається його стереотип («підклас», «суперклас» тощо), при цьому за замовчуванням вважається клас.

Рисунок 5.10 – Параметризований клас

Для стереотипів, які позначають відношення, фіксованими є такі: «асоціація», «наслідування», «екземпляризація», «узагальнення», «розширення>> та інші. Крім того, користувач може вводити стереотипи, властиві специфіці його проблемної галузі, як-от: «успадковує», «контролює», «є наслідком» тощо.

5.3 Діаграми сценаріїв

Зміст і нотація цієї діаграми повністю збігаються з тими, що використовуються в методі І. Джекобсона (див. п. 3.5.1.).

5.4 Діаграми моделювання поведінки системи

Поведінка системи розглядається як обмін повідомленнями між об’єктами для досягнення певної мети. Різні сторони взаємодії об’єктів відображаються у різних діаграмах:

– діаграми послідовності демонструють впорядкованість взаємодії (тобто, обмін повідомленнями) в часі (уподовж лінії життя об’єктів);

– діаграми співробітництва демонструють ролі, які відіграють об’єкти під час взаємодії для досягнення певних цілей;

– діаграми активності демонструють потоки керування при взаємодії об’єктів;

– діаграми станів демонструють динаміку зміни станів об’єктів під впливом перебігу подій (аналогічно до діаграм переходів у стани, про які йшла мова у п. 3.4.3.).

5.5 Діаграми послідовності

Для наочного подання поведінки об’єктів у сценаріях застосовується спеціальна нотація, так звана діаграма взаємодії. Для її побудови кожному з об’єктів сценарію ставиться у відповідність його лінія життя, яка відображає перебіг подій між його створенням та руйнуванням. На діаграмі вона позначається вертикальною пунктирною лінією, на

верхівці якої в прямокутнику зображається назва об’єкта. Діаграма являє собою всі об’єкти, які беруть участь у взаємодії. Крайня права лінія зображає зовнішнє середовище, з яким взаємодіє сценарій, тобто його зовнішній інтерфейс (зокрема це може бути інтерфейс з актором). Допускається кілька зовнішніх інтерфейсів для одного сценарію.

Порядок сусідства об’єктів не має принципового значення і обирається довільно, з намаганням забезпечити наочність взаємодії.

Вертикальний вимір діаграми відповідає осі часу, плин часу вважається вподовж осі зверху донизу і має відносний характер: відстань на осі часу не має відповідності інтервалу реального часу, розташування вподовж осі часу показує лише послідовність подій.

Ліворуч від осі зовнішнього інтерфейсу може наводитися перелік дій, які відбуваються в сценарії за відповідний період часу.

Ініційовані екземпляри об’єктів, поведінка яких забезпечує зазначені дії, зображаються на відповідних лініях життя подовженими прямокутниками. Довжина прямокутника відповідає інтервалу активності екземпляра об’єкта (тобто часу виконання його операцій). Отже, діаграма послідовності відображає потоки керування.

Якщо екземпляр об’єкта існував до старту діаграми, перша стрілка, яка веде до об’єкта, проводиться нижче верхівки його лінії життя.

Якщо об’єкт створюється в певний момент часу, стрілка, яка відповідає повідомленню про його створення, спрямовується до верхівки його лінії життя.

Якщо екземпляр об’єкта руйнується, нижня межа його лінії життя позначається перехрестям як символом руйнування.

Лінії життя можуть розгалужуватися, що демонструє умовні варіації поведінки об’єкта, або зливатися.

Приклад діаграми послідовності наведено на рис. 5.11.

Як уже було сказано, взаємодія об’єктів контролюється подіями, які відбуваються в сценарії і які стимулюють об’єкти для посилання одне одному повідомлень. Повідомлення позначаються на діаграмі стрілками, над якими можуть вказуватися зміст повідомлення, параметри, які воно передає та порядковий номер повідомлення в сценарії. Зазначимо, що вказані на діаграмі повідомлення мають узгоджуватися з видимими операціями відповідних класів.

У наведеному прикладі екземпляри об’єктів 1, 2, 4 були до запуску сценарію і залишилися після його закінчення. Екземпляр об’єкта 3 було створено в сценарії і після виконання певних операцій зруйновано.

5.6 Діаграми співробітництва

Діаграми співробітництва являють собою сукупність об’єктів, поведінка яких значуща для досягнення складових мети системи, та

взаємовідношення тих ролей, які об’єкти відіграють у співробітництві. На даному вигляді діаграм моделюється статична взаємодія об’єктів, при цьому фактор часу не враховується і не відображається на діаграмі співробітництва.

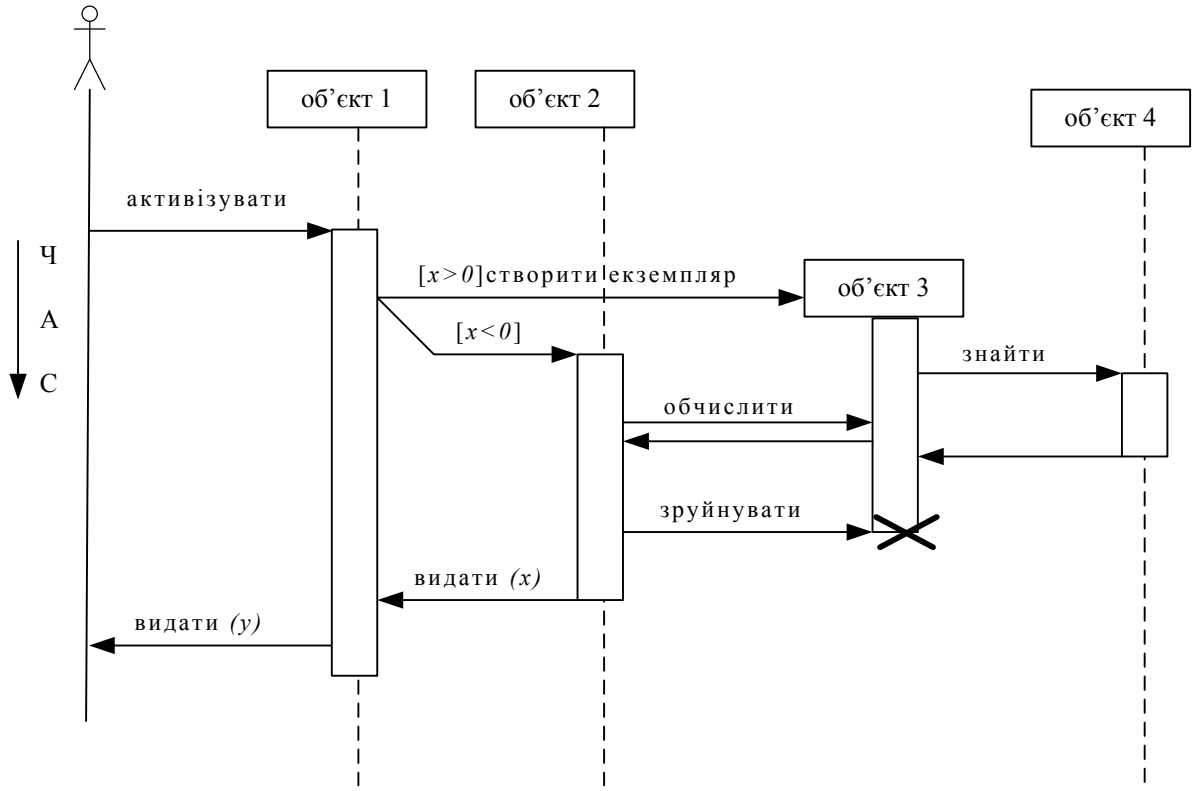

Рисунок 5.11 – Приклад діаграми послідовності

Діаграма співробітництва може бути параметризованою. Тоді вона являє собою абстрактну схему співробітництва – так званий патерн, для якого шляхом до визначення параметрів можна створити певну множину конкретних схем співробітництва. Докладніше патерни обговорюються в розділі 7 (див. п. 7.3.7.).

5.7 Діаграми діяльності

Модель діяльності в UML являє собою поведінку системи як певні роботи, котрі можуть виконувати як система, так і актор, причому послідовність робіт може залежати від прийняття певних рішень залежно від умов, що склалися. Окрема діяльність (робота) зображається на діаграмі прямокутником із закругленими кутами. Потоки керування між роботами показуються стрілками. Якщо мова йде про прийняття рішення, то з відповідного прямокутника виходять дві стрілки, на кожній може позначатися текст умови, якій вона відповідає. Діаграма діяльності нагадує відомі блок-схеми алгоритмів та програм, зокрема передбачено

відображення можливості виконувати паралельно кілька діяльностей і точки синхронізації завершення їх. Приклад. Нехай пошук замовленої книги в бібліотеці має супроводжуватися кількома супутніми процедурами, а саме збиранням даних про виконані та невиконані замовлення. Другі при цьому поділяються на невиконані через те, що в бібліотеці немає книг та через те, що вони в інших читачів. На рис. 5.12. подано відповідну діаграму діяльностей. Горизонтальні лінії позначають розпаралелення та синхронізацію окремих робіт.

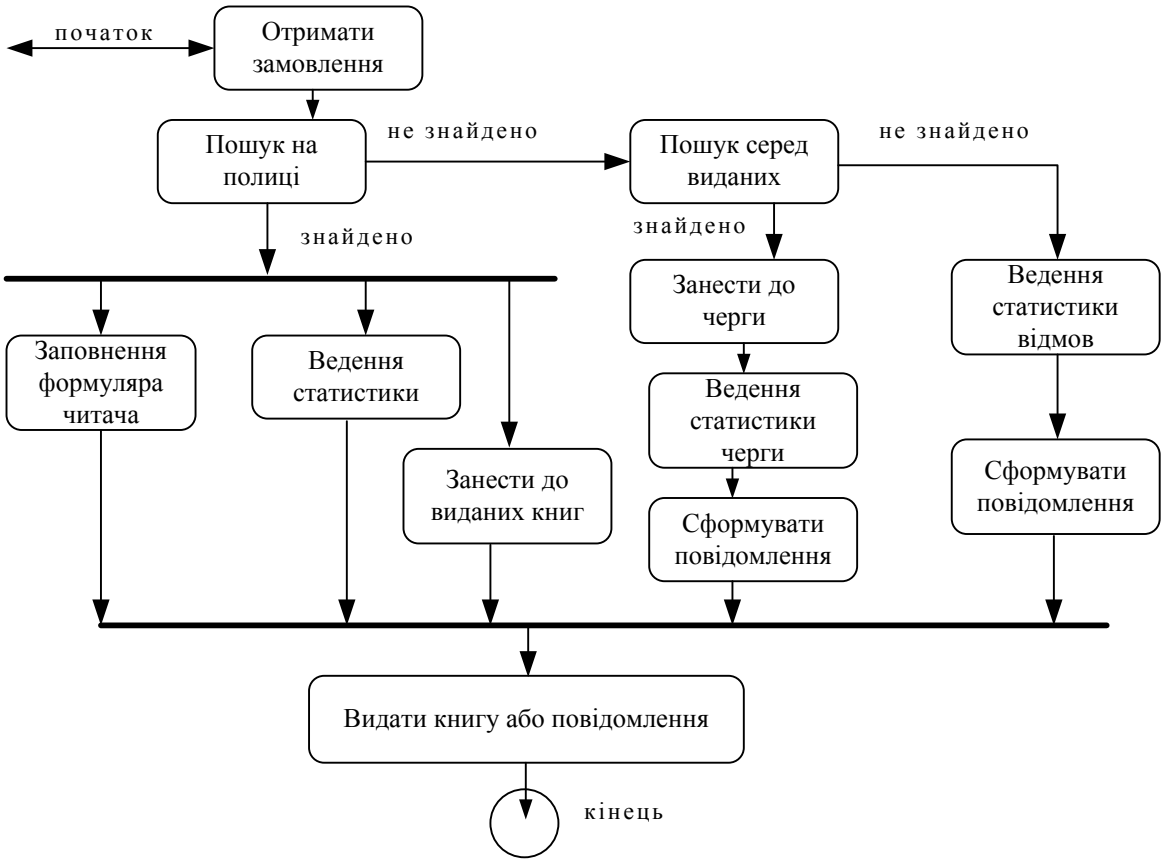

Рисунок 5.12 – Діаграма діяльностей сценарію замовлення книг у бібліотеці

Діаграма може відображати той факт, що певна діяльність виконується для кожного існуючого екземпляра об’єкта, наприклад, для кожного рядка замовлення (якщо, наприклад, замовляється кілька книжок одразу). Тоді стрілка, яка веде до такої діяльності, позначається зірочкою, як на рис. 5.13.

5.8 Діаграми станів

Як і діаграма переходів у стани (див. п. 3.4.2.), модель станів UML базується на використанні розширеної моделі скінченного автомата. Нею

визначаються:

- умови переходів (застереження – guards on transitions)
- переходи, зумовлені певними подіями;
- дії при переході;
- дії при вході в стан;
- діяльність, яка триває доти, доки стан є активним;
- дії при виході зі стану;
- вставка станів;
- паралельно діючі стани.

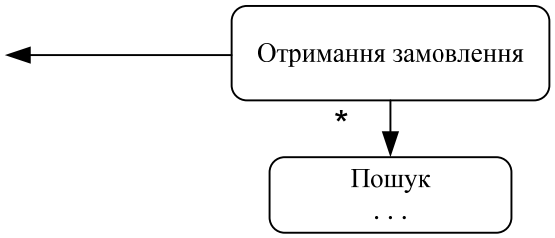

Рисунок 5.13 – Фрагмент діаграми діяльності для кожної замовленої книги

Можуть виділятися суперстани та підстани. Вони разом із вкладеними станами дозволяють конструювати ієрархію станів системи. У даному разі подія – це назва переходу. Події можуть мати аргументи, які визначають список даних, що передаються з переходом (такі, як помилка коду або моніторинг значення). Застереження визначає умови, які потрібно виконати для здійснення переходу. Нарешті список операцій визначає деякий список функцій разом з їхніми аргументами, котрі буде викликано як результат цього переходу.

5.9 Діаграми реалізації

5.9.1 Діаграми компонент

Призначенням діаграми компонент є відображення структур системи як композиції компонент і зв’язків між ними так, як їх уявляє собі програміст. Це граф, вузлами якого є компоненти, а дуги відображають відношення залежності. Серед видів компонент особливого обговорення заслуговує пакет (див. п. 5.10).

5.9.2 Діаграми розміщення

Призначенням даної діаграми є визначення складу фізичних ресурсів системи (що позначаються як вузли системи) та відношень між ними. Системи реального часу в багатьох випадках базуються на різних

платформах замовника. Інженер повинен розробити не лише програмну частину, а й визначити необхідні апаратні пристрої. Ці пристрої мають органічно взаємодіяти з програмними компонентами. Іконка програмної компоненти зображається як прямокутник з двома невеличкими прямокутниками, вузол обладнання – як прямокутник зі спеціальною рамкою. На рис. 5.14 показано розміщення компонент на вузлах системи.

Для діаграми зазвичай використовуються фіксовані стереотипи: <<процесор>>, <<пристрій>>, <<дисплей>>, <<пам'ять>>, <<диск>> тощо.

5.10 Пакети в UML

В UML передбачено загальний механізм організації деяких елементів (об'єктів, класів, підсистем тощо) в групи. Групування можливе, починаючи від системи в цілому і до підсистем різних рівнів деталізації, аж до класів. Результат групування названо *пакетом (package).*

Пакет визначає назву простору, який займають елементи, що входять до його складу і є засобом посилання на цей простір; це особливо важливо для великих систем, котрі налічують сотні, а інколи й тисячі елементів, і тому вимагають ієрархічного структурування.

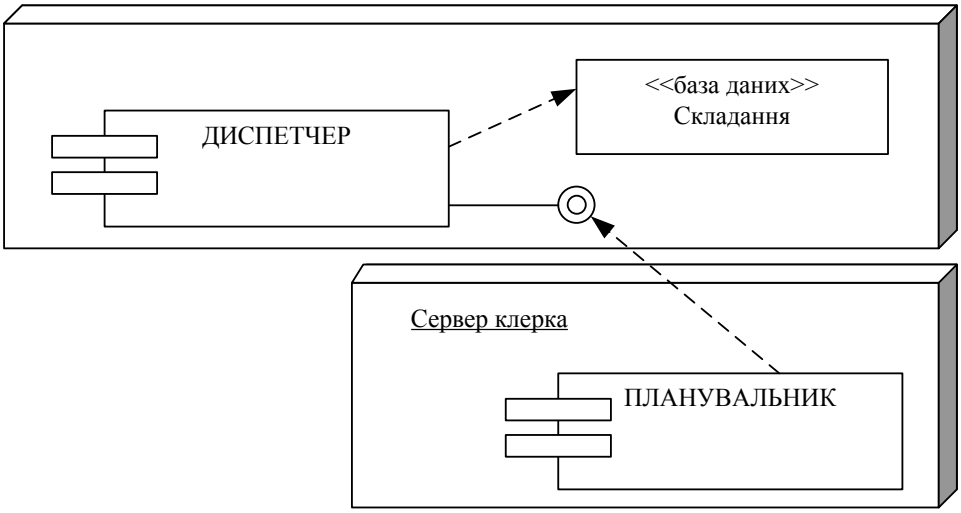

Рисунок 5.14 – Діаграма розміщення

Підсистема в UML розглядається як різновид пакета, який має самостійну функцію.

Групування в пакети може бути вкладеним, тобто складовими пакетів можуть бути класи, інші пакети та підсистеми.

Об'єднання елементів у пакети (так зване пакетування) може відбуватися з різних міркувань, наприклад, якщо вони використовуються сумісно або створені одним автором, або стосуються певного аспекту розгляду, як-от інтерфейс з користувачем, пристрої введення/ виведення і

под. На стадії реалізації до одного пакета може бути віднесено всі підсистеми, що в діаграмі розміщення (розглянуто вище) віднесено до одного вузла.

Призначення пакета – бути елементом конфігурації, тобто елементом, який можна включати як визначену складову композиції в побудові певної системи. На пакет можна посилатися у різних діаграмах, котрі можуть розробляти окремі команди спеціалістів. Терміном *конфігурація (configuration)* будемо позначати отримання програмної системи шляхом добору окремих екземплярів модулів з визначеного наперед складу їхніх варіантів. Так, наприклад, операційна система може мати у своєму складі конфігурацію модулів, що дозволяють взаємодію з різними пристроями, але лише окремі з них приєднані до даного комп'ютера, для якого створюється версія операційної системи як конкретна конфігурація з визначеної множини; система керування польотом літака має у своєму складі конфігурацію модулів, що забезпечують введення показників приладів конкретного борту літака.

Пакет часто може передбачати кілька версій конфігурації його складових.

Нотація для пакета в UML подає його зображення у формі прямокутника, що містить елементи, які він включає.

Пакет, який є підсистемою або системою, зображається прямокутником з закругленими кутами.

Над прямокутником ліворуч зверху розміщується менший за розміром прямокутник, в якому подається стереотип пакета та його назва. Якщо елементи, включені до пакета, не показують, його назва розміщується у великому прямокутнику. На рис. 5.15 показано пакет, названий А 1, до котрого включено пакет В 1, клас К 1 та підсистему С 1.

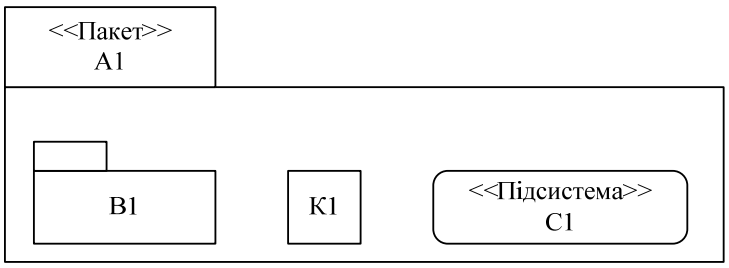

Рисунок 5.15 – Пакет UML

Серед фіксованих стереотипів для позначення різновидів пакета введено такі: <<система>>, <<прикладна система>>, <<підсистема>>, <<елемент конфігурації>>, <<складова системи>>, <<охоплююча система>>, <<фасад>> (facade — пакет, який є інтерфейсом іншого пакета), <<каркас>> (framework — типовий зразок взаємодії об'єктів, детальніше див. п. 7.3.8.), <<заглушка>> (stub — позначення пакета, який

буде описано пізніше) тощо.

Між пакетами може бути встановлено відношення з відповідними стереотипами. Наприклад, відношення А «імпортує» Б означає, що визначені в пакеті Б класи можна використовувати в класах пакета А.

Нагадаємо, що дозволяється виділяти власні категорії стереотипів, в тому числі і для різновидів пакетів, котрі відповідають власним смакам чи потребам. Наприклад, стереотип «успадковано» може додаватися до назви пакета, який є елементом застарілої версії системи, і без перероблення його включено до нової версії системи.

Пакет може належати до різних етапів життєвого циклу розроблення системи – аналізу вимог, проектування, реалізації або кодування, про що може бути зазначено у відповідному стереотипі.

Контрольні запитання і завдання

1. Які з елементів моделювання UML дозволяють визначити головні складові концептуального моделювання проблеми, а саме:
   а) онтологію домену;
   б) модель поведінки об’єктів;
   в) модель процесів?
2. Метод UML пропонує різні нотації (графічні діаграми) для різних аспектів опису проблеми. Чому не єдину?
3. Аналоги яких діаграм UML Вам зустрічалися:
   а) у методі С. Шлеєр та С. Меллора;
   б) у методі Джекобсона?
4. Чи дозволяє діаграма класів UML відобразити відношення:
   а) між класами об’єктів;
   б) між екземплярами класів?
5. Які значення може мати атрибут видимості класів та що вони означають?
6. Які відношення позначаються в діаграмі класів UML спеціальними графічними символами?
7. Які відношення пов’язують сценарії системи?
8. Які діаграми UML доцільно застосовувати для аналізу вимог? З якої діаграми доцільно починати?
9. Які діаграми відображають обмін повідомленнями як єдиний засіб взаємодії об’єктів?
10. Які діаграми зручно застосовувати:
    а) на стадії проектування;
    б) на стадії реалізації?

Чи можна застосувати ті самі діаграми для кількох стадій розроблення?

11. Яка роль стереотипів у нотаціях UML?
12. Які засоби групування елементів моделювання?

6 ТРАНСФОРМАЦІЯ ПРОЕКТУ В ПРОГРАМУ

Процес трансформації проекту в програму має кілька поширених назв: *кодування (coding), програмування (programming), реалізація (implementation)*. Наявність синонімів є наслідком не лише неузгодженості в термінології, а, мабуть, і тієї обставини, що кожний з термінів значною мірою відбиває певний аспект того процесу, про який ми ведемо мову в цьому розділі. Справді, потрібно виконати конструювання, визначивши складові системи як програмні модулі; потрібно запрограмувати ці модулі обраною мовою програмування, для чого слід не лише перевести нотації проекту в коди обраного засобу програмування, а й провести тестування та верифікацію отриманих кодів. А в цілому потрібно перетворити проект на працюючу програмну систему, документовану таким чином, щоб було зрозуміло, як її використовувати, інакше кажучи реалізувати програмну систему. Для певності далі будемо вживати термін реалізація.

Реалізація є першою із стадій життєвого циклу, внаслідок якої артефакти, створені людиною на попередніх стадіях, трансформуються в артефакти, які буде передано комп'ютеру, і тому мають бути для нього однозначно зрозумілими. Така трансформація потребує активного діалогу між особою, котра розуміє проект, і комп'ютером, який має трактувати відповідну до проекту реалізацію. Одна із сторін діалогу є людиною з непередбачуваною поведінкою, прихильністю до замовчування того, що вона має за "загальновідоме", до забудькуватості та нечіткого формулювання, інша є абсолютно педантичним комп'ютером-виконавцем.

Під час перетворення проекту на працюючу систему робиться цілий ряд уточнень та формалізацій, у процесі яких програміст трансформує свої знання в зрозумілі комп'ютеру артефакти і деталізує ті можливі варіанти, які передбачено проектом. При цьому програміст має подивитись з точки зору комп'ютера, щоб зрозуміти, що він уміє і знає з того, що не обумовлено в проекті. Вміння та знання комп'ютера визначаються сукупністю інструментальних засобів, які доступні програмістові. Нині це досить широкий спектр мов програмування, генераторів програм, бібліотек об'єктних модулів, класів, функцій тощо. Тож програміст має знати все або більшість з того, що знає комп'ютер (принаймні готові напрацювання, придатні для його проблемної галузі), щоб зробити правильний вибір потрібних інструментальних засобів та готових компонент, створити ті компоненти, яких не вистачає, а також інтегрувати все в працюючу систему. Прийняття рішення про використання тих чи інших готових компонент може привести до суттєвих змін в архітектурі проекту.

З іншого боку, обрання певного інструментального засобу визначає стилістичний напрям процесу реалізації. Можна назвати три поширені стилістичні напрями:

- лінгвістичний стиль, коли програма має вигляд текстів,

наближених до натуральної мови і тому звичних для більшості користувачів;

- математичний стиль, коли шляхом математичного чи логічного розмірковування можна досить точно висловити свої потреби, оскільки семантику математичних та логічних виразів точно встановлено. На жаль, цей стиль може бути застосовано лише для вузького кола добре сформульованих завдань із чітким і зрозумілим механізмом абстракції. Крім того, володіння таким стилем доступне лише відносно вузькому колу фахівців зі спеціальною освітою;

- візуальний стиль, коли для наочного уявлення взаємодії об'єктів системи активно використовуються зорові образи.

Слід зазначити, що тенденції розвитку інструментальних засобів розроблення програм є такими, щоб інтегрувати всі три стилі в одній розробці. Зокрема це стосується візуальних мов програмування (JAVA і под.).

Вибір і використання готових інструментів та компонент має свою специфіку в програмній інженерії, висвітленню якої присвячено розділ 7.

Контрольні запитання і завдання

1. У чому суть трансформації програмного проекту в програму.
2. Чи визначає сукупність наявних інструментальних засобів можливі напрями реалізації?

7 ПОВТОРНЕ ВИКОРИСТАННЯ В ПРОГРАМНІЙ ІНЖЕНЕРІЇ

7.1 Зміст проблеми

Однією з характерних рис інженерної діяльності є використання готових рішень або деталей. Однак усі, хто працює над створенням реальних систем, знають, що промислове використання готових рішень у програмній інженерії ще не стало повсякденною практикою. Якщо у світі працюють майже 8 мільйонів програмістів, то приблизно 80% з них працюють над створенням програм обліку й організаційного управління на кількох рівнях: окремого підрозділу фірми, окремого аспекту діяльності фірми, фірми в цілому, корпорації, галузі і, нарешті, держави. Це, переважно, задачі розрахунків, статистики, допомоги в прийнятті рішень при управлінні різними ресурсами – кадровими, фінансовими тощо.

За оцінками експертів, 75% цих робіт у світі дублюються – на тисячах підприємств створюються програми складського обліку, нарахування зарплати, розрахунку витрат на виробництво продукції, складання маршрутів деталей на виробничому конвеєрі тощо. Хоча

більшість із цих програм типові, але кожного разу знаходяться особливості, що не дозволяють застосувати розроблену раніше програму. Тому нині активно розвивається напрям, водночас науковий та інженерний, який названо повторним використанням або компонентним розробленням програм.

Компонентне розроблення (*component development)* – це метод побудови програмного забезпечення з конструкцій за каталогом – як композиції готових компонент. Йдеться не лише про порції програмного коду або програмні модулі.

Повторне використання, ревикористання (reuse) – це використання для нових розробок будь-яких порцій формалізованих знань, здобутих під час реалізації завершених розробок програмних систем.

Накопичений досвід розроблення систем програмного забезпечення може бути зафіксовано в різних формах, починаючи від конкретних параметризованих програмних модулів і кінчаючи програмними архітектурами та середовищами. Далі будемо називати повторно використовуваними компонентами (ПВК) елементи знань про минулий досвід розроблення систем програмування, якщо:

а) їх можуть використовувати не лише їхні розробники;

б) їх можна адаптувати для створення нових систем. Зауважимо, що при цьому потрібен каталог, з якого можна зрозуміти, які деталі є і як їх можна поєднувати в конструкцію. Тож повторне використання має включати систематичну цілеспрямовану діяльність зі створення каталогу [1]. Ми називаємо таке повторне використання *систематичним використанням (system reuse),* підкреслюючи цим, що йдеться не про компоненти, створені внаслідок розроблення програмної системи, після завершення якої "випадково" з'ясувалося, що ці компоненти можуть ще комусь стати в нагоді. Досвід так званої реінженерії готових програмних продуктів показав безперспективність пошуку таких знахідок. На відміну від нього, систематичне повторне використання є капіталомістким підходом, що передбачає наявність двох явно виділених процесів в життєвому циклі розробки програмних систем. Зупинимося на суті цих процесів.

Перший процес – створення ПВК. Він включає такі кроки:

а) вивчення спектра завдань, що вирішуються, виявлення серед них загальних підходів до вирішення;

б) побудову для них компонент, які реалізують знайдені підходи або окремі їхні елементи, котрі ми назвали повторно використовуваними компонентами;

в) побудову каталогу, націленого на пошук необхідних компонент. Для успішної реалізації цього процесу необхідно мати певний досвід у вирішенні не одного, а кількох подібних завдань, що дозволяє виявити як їхні спільні риси, так і розбіжності, щоб знайти загальне вирішення для

їхньої реалізації, а також способи настроювання на характерні для кожного завдання особливості.

Другий процес – конструювання цільових систем з готових компонент. Він передбачає такі кроки:

а) зрозуміти, що має робити нова цільова система, для чого вона створюється і які вимоги до неї ставляться;

б) знайти у каталозі серед готових компонент ті, які вважаються підходящими, і зрозуміти, що вони роблять;

в) зіставити мету нової розробки з можливостями знайдених ПВК і прийняти рішення про доцільність використання їх;

г) застосувати відібрані ПВК й інтегрувати їх до нової розробки, забезпечивши необхідні поєднання.

Створення ПВК потребує вкладення капіталу, а використання дозволяє отримати зиск за рахунок економії трудозатрат. Як і всі інвестиції, інвестиції в повторне використання потребують дослідження й оцінювання ефективності вкладень капіталу, прогнозування термінів та обсягів повернення цих вкладень, оцінювання ризиків та інших традиційних бізнес-процесів. Інакше кажучи, бізнес повторного використання, як будь-який бізнес, потребує спеціальних зусиль для менеджменту всіх його процесів. Критеріями успіху такого бізнесу є:

а) забезпечення повторного використання меншими трудозатратами, ніж розроблення програмних систем як разових продуктів;

б) забезпечення пошуку придатних для використання компонент меншими трудозатратами, ніж нове розроблення їхніх функцій для потреб системи, що проектується;

в) забезпечення настроювання компонент на нові умови меншими трудозатратами, ніж нова розробка.

Провідні світові виробники програмного забезпечення широко інвестують дослідницькі проекти з технологій повторного використання та накопичення ПВК.

Програми розвитку повторного використання досліджуються і на загальнонаціональному рівні (відомі загальнонаціональні програми Японії, Голландії, Великобританії, Італії; американські програми департаменту оборони, NASA, військово-повітряних сил США тощо), і на рівні провідних світових корпорацій (IBM, ERICSSON, MICROSOFT, HEWLETT-PACKARD та інші).

У подальших підрозділах цього розділу досліджуються механізми створення ПВК, класифікації їх, пошуку й використання.

7.2 Визначальні властивості ПВК та їх типові поєднання

В основі створення ПВК лежить ідея виявлення так званих родових знань про проблеми, які розв'язуються програмними системами. Цим

терміном позначаються знання, котрі сприяють вирішенню кожного з певного кола завдань (таке коло визначає певну проблемну галузь). Родові знання виявляються як результат абстракції артефактів програмного забезпечення. Під артефактами мають на увазі будь-які продукти діяльності фахівців з розробки програмного забезпечення (ПЗ).

Типи артефактів при цьому не обмежуються фрагментами коду. Вони можуть бути структурами проекту, структурами реалізації модулів, документації, трансформації і т.д.

За величезної різноманітності форм ПВК, які використовуються у комп'ютерній практиці, є безперечна спільність у техніці їхнього застосування. Всі вони конструюються в процесі абстрагування і застосовуються шляхом виконання процесів селекції, спеціалізації й інтеграції. Перелічені чотири процеси визначають, так би мовити, чотири виміри, в аспектах яких ПВК можуть абстрагуватися, класифікуватися, добиратися і зіставлятися. Розглянемо ці процеси докладніше.

Абстракція. Абстракція є визначальною властивістю ПВК, кожна абстракція – продукт аналізу ознак зібрання сутностей, для котрого виділяються суттєві спільні ознаки й відкидаються несуттєві з певного погляду. Внаслідок цього визначається абстрактна або родова сутність, що матеріалізується як ПВК.

Кожна абстракція може розглядатися як така, що має дві частини – видиму та приховану. Видима частина є специфікацією тих знань, які необхідні для використання ПВК. Прихована частина містить деталі реалізації ПВК, невидимі на рівні її специфікації.

Видиму частину, у свою чергу, доцільно розглядати як сукупність стабільної або фіксованої та змінної складових.

Змінна частина являє собою змінні властивості ПВК, її здатність адаптуватися до якогось спектра застосувань, фіксована частина – інваріанти. Тобто, специфікація абстракції із змінною частиною відображається в множину альтернативних реалізацій.

Розподіл абстракції на видиму фіксовану і змінну та приховану частини є не внутрішньою властивістю абстракції, це передовсім проектні рішення розробника абстракції, які він приймає, даючи відповіді на такі запитання:

- яка інформація потрібна користувачу? Потрібна користувачу інформація подається як специфікація (видима частина);
- які з властивостей абстракції користувачу доведеться змінювати? Такі властивості доцільно подати у змінній частині специфікації.

Як приклад, розглянемо стек – відому схему запам'ятовування елементів за принципом: "елемент, що надійшов до сховища даних останнім, видається першим". Абстрагуємося при *цьому* від структури та форми подання елементу стека. Опис принципу роботи стека подається в

специфікації, а максимальна глибина стека може бути або в прихованій частині (коли користувач нею не цікавиться), або в змінній частині (коли необхідно змінити її за бажанням користувача). Якщо подати глибину стека у фіксованій частині, користувач буде знати її значення, але не зможе його змінити.

Добір або селекція. Ключовою особливістю технологій використання ПВК є та обставина, що, приступаючи *до* нової програмної розробки, користувач не має відомостей про готові компоненти, тому у ПВК має бути чітка і виразна специфікація абстракції, котра *давала* б користувачеві можливість ефективно переглядати, розуміти, зіставляти і добирати підходящі для нього ПВК із числа наявних.

Подібно до бібліотечного пошуку, на допомогу користувачу може бути надано спеціальні каталоги й класифікатори зібрань компонент, призначених здійснювати навігацію пошуку і полегшувати розуміння сутності ПВК.

Спеціалізація. Як було зазначено вище, зазвичай ПВК є матеріалізацією певної абстракції або родового артефакту, який неможливо беспосередньо використати. Для його використання необхідно спеціалізувати або конкретизувати його, тобто провести операцію, обернену до абстракції, зміст котрої полягає в тому, щоб до родових ознак, властивих кожному з "представників роду", додати ознаки конкретного застосування – параметри, трансформації, обмеження тощо. Родовий артефакт є, по суті, абстракцією зі змінною частиною. Спеціалізація родового артефакту – це отримання реалізації абстракції на основі довизначення змінної частини ознаками конкретного застосування.

Інтеграція. Для ефективної інтеграції розрізнених артефактів, втілених у ПВК, до конкретної програмної системи потрібно, щоб користувач розумів інтерфейс артефакту, тобто особливості його взаємодії з іншими артефактами або з якимсь каркасом, який певною мірою можна вважати базою інтеграції. Можна сказати, що інтерфейс артефакту – це абстракція, в якій не враховуються внутрішні властивості артефакту, специфікація котрої визначає правила взаємодії артефактів між собою.

Для оцінювання ефективності абстракції вводиться інтуїтивна міра, яку названо *когнітивною відстанню (cognitive distance)*. Цим терміном позначається кількість інтелектуальних зусиль, які необхідно витратити розробникові програмної системи для того, щоб перевести її з однієї стадії життєвого циклу в іншу. Для створення програмної системи за допомогою ПВК розробник ставить за мету мінімізацію когнітивної відстані між початковою концепцією системи та її кінцевою реалізацією, готовою до виконання.

Засобами досягнення цієї мети є:

– використання фіксованих і змінних абстракцій, лаконічних і виразних водночас;

– максимізація прихованої частини абстракції;

– застосування автоматичного відображення специфікації абстракції в її реалізацію (наприклад, шляхом компіляції).

Якщо поглянути на розвиток у часі засобів створення програм з точки зору повторного використання, то історична картина виглядає як поширення застосувань комп'ютера на нові і нові домени, набуття практики вирішення окремих завдань в рамках цих доменів, узагальнення набутого досвіду як накопичення типових абстракцій та втілення їх у ПВК певної форми. Історично першим доменом застосування програмних засобів була галузь числових розрахунків, для якої було створено і перші ПВК, котрі мали вигляд спочатку типових програмних модулів (так званих підпрограм – аналог сучасних бібліотечних функцій), а згодом і мов програмування. Перше покоління мов програмування було втіленням типових абстракцій алгоритмічних обчислень (мови програмування Фортран, Алгол-60 та подібні їм).

Друге покоління з'явилось як відповідь на поширення комп'ютерних методів на домени оброблення великих обсягів економічної інформації, завдяки чому було виділено абстракції типових процесів оброблення даних, збирання програм з окремих модулів (Кобол, PL/1, Паскаль, Модула тощо).

Поява візуальних можливостей обміну інформації з комп'ютером покликала до життя наступні покоління ПВК, подані абстракціями комп'ютерної графіки, діалогової взаємодії, узагальненою концепцією інтерфейсу. Ці узагальнення втілено в мови програмування (C++, Java тощо) та багато інших форм подання узагальненого досвіду розв'язання певних проблем у вигляді ПВК. Нижче досліджуються різні форми втілення ПВК у сучасній комп'ютерній практиці.

7.3 Поширені в користуванні категорії ПВК

У сучасній технології програмної інженерії використовуються категорії ПВК, кожна з яких є типовим поєднанням наведених вище аспектів розгляду ПВК, а саме:

– типу абстракції, шляхом якої виділяються родові знання, готові до ревикористання в кожній задачі з певного спектра;

– механізмів спеціалізації або настроювання родових знань на конкретні вимоги стосовно створюваних або цільових програмних систем;

– механізмів інтеграції ПВК у цільові системи;

– механізмів вибору серед готових конкуруючих ПВК, тих, що відповідають вимогам цільової розробки і забезпечують мінімізацію когнітивної відстані для певних фаз життєвого циклу розробки. Експертне дослідження сучасного стану повторного використання дозволило виділити типові категорії ПВК, які значно поширені в сучасній практиці

програмних систем. Їхні особливості обговорюються в п. 7.3.1–7.3.8.

7.3.1 Мови програмування високого рівня

Конструкції мов програмування є абстракціями типів даних, функцій оброблення, способів конструювання алгоритмів та складання програм з окремих конструкцій, характерними для окремих проблемних галузей або більш-менш широкого спектра проблемних галузей. Таким чином, конструкції мов програмування відіграють роль ПВК.

Конструкції мов програмування високого рівня – це специфікації, котрі мають реалізацію в асемблері.

Постійна частина абстракції відповідає семантичному опису конструкцій мови на рівні метасимволів, змінна частина подана термінальними символами конструкцій.

Прихована частина абстракції містить подробиці реалізації конструкції в асемблері, наприклад, проміжні дані тощо.

Відображення специфікації в реалізацію виконується цілком автоматично за допомогою компілятора.

Спеціалізація конструкцій здійснюється шляхом підставлення термінальних виразів (наприклад, арифметичних або умовних) в граматичні конструкції мови.

Селекція підходящої мови програмування серед мов-претендентів здійснюється шляхом експертного зіставлення конструкцій мови-претендента з інтуїтивним уявленням щодо функціональних та виконавчих властивостей тих програмних систем, які потрібно розробити.

Інтеграція окремих конструкцій кількох мов у нову мову програмування здійснюється як результат неформалізованої інтелектуальної діяльності автора-експерта. Прикладом може бути інтеграція конструкцій мов Алгол-60, Фортран, Кобол у мови PL/1, Алгол-68, Паскаль та ін.

7.3.2 Компоненти вихідного коду

Цим терміном назвемо фрагменти коду вихідною мовою програмування, які створювалися і запам'ятовувалися спеціально для повторного використання.

До цієї категорії належить абсолютна більшість ПВК, котрі на сьогодні масово використовуються. Бібліотеки функцій та класів для багатьох мов програмування налічують тисячі компонент.

Природа ПВК значною мірою залежить від вихідної мови. У багатьох із сучасних мов програмування передбачено спеціальні засоби для подання абстрактних типів даних, абстракцій управління, процедур, модулей, пакетів, функцій, класів тощо.

Головна проблема у створенні бібліотеки ПВК – знайти точні специфікації абстракції для компонент. Без цього користувачу для вибору ПВК буде важко зрозуміти призначення компоненти, а іноді для такого розуміння навіть доведеться вдатися до аналізу власне коду компоненти, що зведе нанівець переваги повторного використання.

Розробник бібліотеки ПВК має забезпечити специфікацію абстракції, яка стисло подає поведінку компоненти, схему її класифікації та пошуку. Найбільшого успіху досягли бібліотеки з абстракцією "одним словом", що є однозначно зрозумілим у межах проблемної галузі, наприклад, матриця та SIN у числових обчисленнях, абстракції схем запам'ятовування Буча стек, черга, дерево в системному програмуванні. Використання компонент вихідного коду часто вимагає роботи з усіма частинами абстракції – фіксованою і змінною, прихованою і видимою.

Спеціалізація компонент може відбуватися кількома способами. Найдавніший з них і досі має застосування – це пряме редагування вихідного коду, для чого доводиться доходити до подробиць низького рівня. У цьому разі ефективність застосування ПВК досить мала.

Більш ефективний підхід – проектування родових компонент з передбачуваними правилами адаптації до вимог конкретного застосування, яке може виконуватися автоматично. Тоді замість прямого редагування коду розробникові достатньо визначити родові параметри як приховану частину абстракції специфікації.

Наприклад, для сортування певних сутностей параметрами можуть бути типи елементів, що підлягають сортуванню, їхня часткова упорядкованість, схема індексування, яку було застосовано під час накопичення їх. Процес спеціалізації при цьому може виконуватися автоматично, як, наприклад, під час використання макрозасобів з параметрами і відомих родових пакетів АДА.

Селекція ПВК вимагає спеціальних способів і зусиль як з боку організаторів бібліотек вихідних модулей, так і з боку користувачів. Зазвичай застосовуються поширені в бібліотечній справі способи, як, наприклад, ієрархічна рубрикація, використання індексів ключових слів, котрі характеризують зміст бібліотечної компоненти, неформальні анотації. Для користувача пошук компоненти при цьому зводиться до пошуку рубрики, що відповідає характеру завдання, яке він має вирішувати (наприклад, обчислювальні функції, робота з потоками даних, виведення на екран тощо), пошуку серед визначених тем рубрики і вивчення неформальних анотацій.

Для інтеграції ПВК більшість компіляторів з мов програмування мають автоматичні засоби складання модулів та вирішення можливої колізії імен.

7.3.3 Класи об'єктів та абстрактні класи

Згідно з парадигмою об'єктного підходу, наведеною в п. 3.3.1, програмна система розглядається як спільнота взаємодіючих об'єктів, істотними ознаками яких є:

– стан, поданий сукупністю атрибутів;

– поведінка;

– спроможність вступати у відношення з іншими об'єктами;

– спроможність посилати одне одному повідомлення.

Об'єкти з однаковими істотними ознаками об'єднуються в класи. Клас, таким чином – це абстракція властивостей, поведінки та інтерфейсів об'єктів. Інтерфейс є видимою частиною абстракції, що визначає правила взаємодії об'єкта із зовнішнім світом, тобто фактично правила його використання.

Об'єкти можуть визначатися на стадії аналізу вимог до розроблення системи, тоді вони відповідають реальним об'єктам предметної області або її поняттям. Повторне використання таких об'єктів на ранніх стадіях життєвого циклу розробки програм у цьому разі не залежить від середовища реалізації системи. У процесі реалізації властивості об'єкта, як правило, відображаються в атрибутах відповідних баз даних, а поведінка об'єкта відображається у визначені для нього операції або методи. Інтерфейси між об'єктами подаються тими повідомленнями, які вони можуть посилати.

Для об'єктно-орієнтованих мов програмування розроблено представницькі бібліотеки ПВК як класів об'єктів. Наприклад, відомі бібліотеки C++ налічують тисячі повторно використовуваних класів. Істотним відношенням класів об'єктів є відношення успадкування. Два класи об'єктів перебувають у відношенні успадкування, якщо один з них (спадкоємець) володіє всіма істотними ознаками другого (успадкованого) класу і має деякі додаткові властивості, поведінку або інтерфейс. За допомогою механізму успадкування природним чином локалізуються фіксована частина абстракції (подана успадкованим класом, який називають суперкласом) і її змінна частина (подана класом-спадкоємцем, який називають підкласом). Отже, внесення змін до ПВК або її настроювання здійснюються як визначення потрібного підкласу, що спрощує супровід розробленої системи.

Суперкласи, які визначено не повністю і їх не можна самостійно використати, названо абстрактними класами. Для використання їх має бути додано підкласи, що успадковують всі властивості, характерні суперкласу, але мають додатково визначені деталі.

Прикладами абстрактних класів служать так звані контейнерні класи – структури зберігання даних, для кожного з яких визначено правила запам'ятовування або видачі чергового елементу контейнерного класу,

однак при цьому реалізація цих операцій залежить від типів елементів, що заповнюють "контейнери". Поширеними прикладами контейнерів можуть бути списки, черги, стеки, матриці, таблиці. Механізм контейнерів реалізовано в C++ у формі так званих шаблонів (templates), створено спеціальні бібліотеки ПВК такого типу.

7.3.4 Абстрактні архітектури програмних систем

Всяка архітектура визначається складом частин, з яких будується ціле, і способом їхньої композиції в ціле. *Архітектура програмної системи (programming system architecture)* визначається в термінах компонент, які виконують обчислення, та інтерфейсів між ними. Архітектура – це відображення правил декомпозиції складності проблеми.

Спектр задач проблемної галузі, що допускають схожі способи розв'язання їх, називають доменом предметної галузі. *Абстрактна архітектура (abstract architecture)* – це декомпозиція спільного вирішення для виділеного спектра завдань домену на підсистеми або ієрархію підсистем, на кожному рівні якої фіксуються можливі варіанти виділених параметрів та обмежень, котрим відповідають варіації складу виділених компонент.

Таким чином, абстрактні архітектури визначають глобальну структуру проектованої системи і функціональність її стабільних головних складових, що може розглядатись як фіксована частина абстракції підходу до вирішення виділеного спектра завдань.

Абстрактні архітектури звичайно будуються для вузьких проблемних галузей, що дозволяє досягти високої спільності підходів до вирішення властивих галузі завдань і, відповідно, високого рівня абстракції ПВК.

Прикладами ПВК абстрактних архітектур є:

– системи управління базами даних;

– архітектура компілятора, в яку можна вбудовувати різні лексичні й синтаксичні аналізатори, генератори кодів;

– архітектура експертної системи, основаної на правилах, тощо.

Спеціалізація абстрактної архітектури здійснюється шляхом довизначення варіацій компонент. Наприклад, спосіб обчислення певного показника в системі управління польотом літака може бути довизначено залежно від фізичних особливостей приладів, які встановлено на конкретному борту літака.

Селекція потрібної архітектури здійснюється шляхом аналітичного зіставлення характеристик проблемної галузі, що покривається абстрактною архітектурою, і характеристик тієї проблемної галузі, для якої створюється програмна система.

Інтеграція абстрактних архітектур у складні системи полегшується

чіткою специфікацією їхніх інтерфейсів, завдяки якій кожна абстрактна архітектура може розглядатися, у свою чергу, як підсистема архітектури вищого рівня.

7.3.5 Генератори прикладних застосувань

Генератори прикладних застосувань працюють подібно до компіляторів для мов програмування: вхідні специфікації автоматично транслюються в потрібний об'єктний код (в тому числі в код мови програмування високого рівня). Особливість генераторів полягає в тому, що використовувані ними специфікації, як правило, є абстракціями дуже високого рівня для дуже вузьких доменів проблемних галузей, що дозволяє забезпечити повторне використання абстракцій як даних, так і процедур та архітектури проекту в цілому.

Абстракція в генераторі прикладних застосувань належить до родових знань про домен проблемної галузі і має для користувача вигляд специфічних для домену різних моделей обчислень і даних.

Фіксована частина абстракції визначається тими складовими застосувань, які користувач не може змінювати в процесі генерації; змінна частина визначається тими параметрами, що задаються користувачем.

Комбінація фіксованої частини і діапазону змінних частин визначає спектр систем, які можна реалізувати за допомогою генератора. Ця міра дістала назву покриття домену генератором.

Розробник програмних систем працює винятково на дуже високому рівні абстракції. Усі деталі реалізації генератора є для користувача прихованою частиною абстракції, що позбавляє його необхідності (а також можливості) бачити, розуміти або модифікувати вихідний результат генератора.

Селекція або вибір генератора – це пошук такого генератора, чиє покриття домену включає вимоги постановки потрібного завдання. До нинішнього часу в обчислювальній практиці використовується лише невелика кількість дуже спеціалізованих генераторів з вузьким покриттям домену. Однак тенденція формалізації все нових і нових предметних галузей є такою, що можна очікувати появи великих бібліотек генераторів. Пошук у них генератора, відповідного потрібному завданню, має здійснюватися в термінах характеристик домену.

Спеціалізація. Розробник програмної системи спеціалізує генератор, задаючи її специфікації.

Зазначимо, що в тому разі, коли функціональні потреби системи, що розробляється, виходять за покриття домену генератора, його ре-використання стає неефективним.

Інтеграція. Як правило, генератор прикладних застосувань виробляє на виході завершену, готову для виконання систему, тому питання

інтеграції для неї не ставиться. В тому разі, коли генератор генерує окремі компоненти систем, правила їхньої композиції визначаються в термінах абстракції високого рівня.

Залежно від вбудованих в реалізацію генератора способів спеціалізації може бути виділено різновиди генераторів, котрі набули широкого поширення. Назвемо їх.

Генератори програм оброблення даних у бізнесі. Специфікації для таких генераторів визначають схеми баз даних, а їхнє оброблення визначають на так званих мовах четвертого покоління. Прикладами таких генераторів є:

а) управління даними – базується на одній моделі даних; наприклад, на реляційній або ієрархічній;

б) генератори форм документів – базуються на асоціативному пошуку даних (так звані генератори звітів);

в) графічні генератори звітів – аналогічні текстуальним, але на виході дозволяють отримувати графічне подання результатів (даних) у формі графіків, діаграм тощо.

Спеціалізація може включати обмеження на окремі елементи даних та їхні відношення, обмеження на управління доступом і вироблення кута зору (view) на бази даних.

Генератори експертних систем. Експертні системи – це системи, які включають і застосовують для розв'язку задач домену експертні знання.

Абстракціями для них є узагальнені способи розв'язання задач у проблемних галузях шляхом оброблення знань незалежно від їхньої природи, яка може суттєво відрізнятися, як, наприклад, при діагностиці людей і двигунів. Спосіб розв'язання задач є фіксованою частиною абстракції генератора експертних систем, а змінною частиною (за якою він спеціалізується) – експертні знання про конкретну область діагностики;

Парсери й компілятори компіляторів. Їхня основна абстракція – це регулярні вирази для генерації лексичних аналізаторів і контекстно-незалежні граматики для генерації парсерів, що узагальнюють знання про принципи дії й алгоритми лексичних та інших аналізаторів, видавання помилок, генерацію коду тощо. Їхня спеціалізація відбувається за конкретними лексичними та граматичними правилами.

7.3.6 Абстрактні домени

Розвитком ідеї абстрактної архітектури є каталогізація доменів масового вжитку й узагальнення їхніх властивостей як поняття “абстрактний домен” [2]. Виконуючи аналіз подібності та розбіжностей сукупності доменів, експерти можуть помітити аналогії для таких, на перший погляд, відмінних речей, як продаж театральних квитків, резервування місць на рейси літаків або розподіл деталей на оброблення

для верстатів заводського конвеєра. Помітивши аналогії в структурі знань різних із синтаксичного погляду доменів, експерти можуть ввести абстракції об'єктів, котрі можна конкретизувати у відповідні об'єкти аналогічних доменів. Вимоги до систем, що належать до цих доменів, може бути подано в термінах відповідного абстрактного домену.

Автори пропонують набір абстракцій проблем, які слугують базисом для генерації специфічних для конкретного домену сценаріїв складання та перевірки вимог, використання їх під час експлуатації, складання вимог у формі юридичного контракту природною мовою. Підхід є першою систематичною спробою покриття простору (моделювання) інженерії вимог у сукупності доменів, котрі обслуговують менеджмент у бізнесі. Гіпотетично вони мають опиратися на ментальні моделі природних категорій когнітивних наук. Це є підхід, що обіцяє бути ефективним.

Простір моделей систем об'єктів становлять 13 базових ієрархій. Верхній рівень моделі системи об'єктів кожної ієрархії визначається з використанням базових: поведінки, складу об'єктів, агентів, структури домену для одного класу проблем. Спеціалізація кожної із цих моделей (у формі систематичного додавання різних типів знань — станів, мети, подій) генерує простір до 200 листів-вузлів моделі систем об'єктів. Кожний вузол-лист визначається з використанням станів, переходів у стани, подій, об'єктів, агентів, структури домену, абстрактних відношень між об'єктами й агентами, передумов переходів у стани, властивостей та атрибутів об'єктів. Перелічимо 13 моделей верхнього рівня:

а) повернення ресурсів – — двоспрямоване передавання ключових об'єктів від структури, що зберігає, до клієнтської і навпаки, наприклад, бібліотеки;

б) забезпечення ресурсами – односпрямоване передавання ресурсів від структури, що зберігає, до клієнтської та контроль за необхідністю поповнення ресурсами структури, що зберігає, наприклад, складів;

в) використання ресурсів – односпрямоване передавання від структури, що зберігає, до клієнтської для оброблення;

г) композиція елементів – агрегування із синтезом нового; наприклад, складання деталей у продукт;

д) декомпозиція на елементи;

е) розміщення ресурсу – моделює зміни стану ресурсу щодо інших транзакцій, наприклад, стани квиткових кас;

ж) логістика – комплексна діяльність з планування, наприклад, маршрутів парку машин;

и) спостереження за об'єктами – відстеження руху об'єкта в просторі чи у фізичній структурі, наприклад, радарне;

к) побудова сполучення між об'єктами – пов'язування об'єктів, які

володіють інформацією, в топологію зв'язків;

л) управління агента об'єктом – подання команд та керуючих компонент, до яких агенти посилають повідомлення, що викликають зміни в поведінці об'єктів, наприклад, контроль суден у порту;

м) моделювання домену – комплексні засоби моделювання для агента-людини, наприклад, військовика;

н) оброблення ділянок — діалогові зміни стану ділянки, наприклад, обробка слів або креслень;

п) перегляд об'єкта – домени, в яких об'єкти не змінюють стану, але переглядаються агентами – людьми або комп'ютерами.

Перелік абстрактних доменів, безперечно, може бути поповнено, якщо певна спільнота професіоналів може узагальнити сукупність розв'язаних нею проблем у формі абстрактного домену. Прикладом домену масового використання, досвід роботи з яким дозволяє побудувати його абстракцію, є задачі так званої аналітичного оброблення даних, чи не наймасовіші у сфері економічних завдань, зокрема АСУ, здавна відомі під назвою отримання розрізів даних [3, 4].

Для згаданих вище доменів можна навести чимало прикладів завдань, несхожих зовні, але вирішення яких вимагає маніпулювання об'єктами з аналогічною поведінкою. У межах цих доменів побудовано сотні працюючих програмних систем, і тому побудова для них класифікацій є проблемою скоріше організаційною, бо є достатня кількість експертів і достатня сукупність готових об'єктів для експертизи.

7.3.7 Патерни

Тип ПВК, який позначається терміном “патерн”, є принципово відмінним від тих, котрі розглянуто дотепер і котрі, певною мірою, належали до програмних модулів чи їх узагальнень.

Патерни є абстракцією спілкування (взаємодії) певної сукупності об'єктів у кооперативній діяльності, для якої визначено абстрактних учасників, їхні ролі, взаємовідносини та розподіл відповідальності.

Патерн фіксує певне узагальнене рішення, яке дозволяє спілкуватися розробникам та замовникам і розмірковувати щодо проблеми та її розв'язання. Це крок до стандартних інженерних шляхів застосування успішних розв'язань до відомих проблем, як це ведеться у сталих інженерних дисциплінах (наприклад, для автомобіля відомо про певні набори з’єднання вузлів, щоб одержати спортивний варіант, або високо-комфортний, або для поганих доріг тощо). Інакше кажучи, це шлях до ревикористання корисного досвіду, який передається неформально за допомогою якісної, ментальної моделі опису знань.

Цей термін справді походить з архітектури. Патерн у контексті програмної інженерії є позначенням проблеми, яка виникає в багатьох

контекстах, при цьому ядро її розв'язання можна подати так, щоб повторно використовувати його для більшості з них. Патерни описують окрему повторювану проблему проекту як визначену наперед схему її розв'язання шляхом специфікації абстракції її складових, їхніх ролей та взаємовідносин і відповідальності. Найчастіше патерни виражають фундаментальну парадигму структурування системи, як, наприклад, архітектури клієнт – сервер чи замовник – посередник – послуга.

Патерни мають певний контекст застосування; зазвичай можна визначити і мотиви доцільності застосування патерну, й аргументи, що суперечать їм. І кожне з рішень має певну вартість, а опис патерну має визначати цю вартість.

Відомо кілька схем опису патернів, у більшості з них визначається проблема, яку розв'язує взаємодія, контекст, в якому сформульовано проблему та підхід до її розв'язання. У наведеній схемі відображено не стільки технічні ідеї, скільки соціальні аспекти прийнятих рішень: проблема відображає мотиви розробки, контекст характеризує явища, реакцією на які є розв'язання проблеми та аргументи проти розв'язання.

Спеціалізація патернів відбувається шляхом конкретизації (довизначення) об'єктів, котрі беруть участь у взаємодії.

Селекція (добір) патернів, адекватних системі, що будується, відбувається неформально, шляхом вивчення та зіставлення каталогів патернів.

Інтеграція обраних патернів у нову розробку відбувається неформально, найчастіше сукупність патернів для певного домену не інтегрується в мову проектування, а скоріше – це зібрання окремих одиниць, кожна з яких відповідає певній схемі взаємодії, котру можна конкретизувати для певних об'єктів розробки.

Відомо кілька таких зібрань, окремі з яких вже широко застосовуються в процесі розроблення програмних систем, і не лише як певні готові рішення, які можна використати повторно, а і як засоби обговорення прийнятих рішень та визначення їх [5].

Об'єднання патернів у зібрання може відбуватися за класифікаційними ознаками, зокрема за сферою застосування їх. Так, патерни проектування можна умовно поділити на три групи.

Креативні патерни (від англ. create – створювати) відображають процеси створення екземплярів об'єктів. Нижче наведено приклади таких патернів:

- ініціатор – це відповідь на проблему, яка полягає в тому, що конкретний тип об'єкта часто змінюється; контекст визначається як багаторазова потреба породжувати екземпляри об'єкта; рішення – інтерфейс створення об'єктів відокремлюється від власне створення, яке вирішується шляхом визначення підкласу, котрий конкретизує, що саме створюється;

- трансформер – це відповідь на проблему організації перегляду файлів та трансформування їх у різні формати; рішення – відокремити механізм перегляду від алгоритмів створення нових форматів;

- прототип – це відповідь на проблему породжувати екземпляр об'єкта за заданим зразком; рішення – визначити тип за екземпляром прототипу і створити новий об'єкт як копію прототипу.

Структурні або архітектурні патерни визначають композиції класів об'єктів. Їхніми типовими прикладами є:

- адаптер – це відповідь на проблему пристосовування інтерфейсу до смаку персонального клієнта. Контекст – смаки часто змінюються. Рішення – визначити абстракцію інтерфейсу як суперклас, а настроювання на персональні вимоги клієнта – як підклас;

- *фасад(facade)* – це відповідь на проблему локалізації в підсистемах властивостей, що мають високу вірогідність змін. Контекст – найбільшу вірогідність змін мають об'єкти інтерфейсу. Рішення – передбачити можливі зміни в підсистемі та зосередити їх в одному класі;

- пошарова структура – це відповідь на проблему дотримування певного рівня абстракції в осмисленні проблеми на кожній стадії життєвого циклу розробки програмної системи (див. рис. 4.2.) Контекст – наявність розробників з різними рівнями кваліфікації, що зумовлює здатність виконувати роботу на відповідних рівнях абстракції. Рішення полягає в декомпозиції системи на ієрархію шарів, для кожного з яких встановлюється стандартизований інтерфейс між компонентами попереднього та наступного шарів. Кожен із шарів є сукупністю віртуальних машин, відповідних певному рівню абстракції, а компонента є реалізацією такої машини. Для кожного шару можна визначити певний спектр компонент, взаємозамінних та сумісних щодо їхніх інтерфейсів (засобів включення), але відмінних з погляду реалізації цих інтерфейсів (наприклад, різні прилади в авіоніці можуть вимірювати ту саму величину).

Найчастіше спектр компонент – це перелічена множина компонент, і спектри можуть бути вкладеними один в одний. Кожна з компонент експортує свій інтерфейс до віртуальної машини верхнього шару й імпортує інтерфейс віртуальної машини нижнього шару, тобто трансформує перший у другий, але при цьому інкапсулює відображення. Спектри та компоненти можна визначити як складові граматики, множина речень якої характеризує сімейство систем, котрі може бути породжено на базі патерну. При цьому може бути накладено певні допоміжні обмеження стосовно правил породження.

Поведінкові патерни визначають взаємодію класів та їхніх екземплярів (як розподіл абстрактних ролей) і їхню поведінку. Приклади:

- ланцюг повноважень – це відповідь на проблему роз'єднання

відправників повідомлень від тих, хто їх обробляє. Контекст – потреба в роздільному функціонуванні перших і других. Рішення – повідомлення передається від об'єкта до об'єкта, доки не дійде до того, хто виконає його обробку;

- клієнт/сервер/сервіс – це відповідь на проблему інкапсулювання подробиць реалізації програмних послуг, щоб вивільнити клієнтів від необхідності знати їх (згідно з принципом приховування інформації, про який ішлося в розділі 3). Контекст – сервіс може бути запрошено паралельно й одночасно багатьма клієнтами. Рішення – побудова двох класів. Перший з них – сервіс – інкапсулює послідовність виконання послуги, підтримує стани виконання послуги і передає події, котрі виникають у процесі виконання послуги. Другий – сервер – інкапсулює ресурси та послуги сервісів і визначає абстрактний інтерфейс з клієнтом, конкретизація якого ініціює конкретне виконання конкретного сервісу.

Підсумовуючи зазначене вище, можемо сказати, що патерни є спробою висловити інтуїтивні рішення, набуті досвідом окремих провідних спеціалістів як формалізовані абстракції, конкретизацію яких можна виконати широким колом спеціалістів за умов, відповідних їхнім вимогам.

Зазначимо, що наведені приклади можуть декому здатися давно знайомими, і так воно і є, але новим є подання їх у вигляді чітко визначених правил взаємодії об'єктів.

7.3.8 Каркаси (Frameworks)

Каркас проблеми визначає готову структуру головних складових частин, на які має поділятися система, побудована для розв'язання проблеми. ПВК типу каркас [6] виникли як розвиток об'єктно-орієнтованого підходу до розроблення програмних систем [16].

Згідно з парадигмою згаданого підходу каркас об'єднує множину класів об'єктів, які взаємодіють між собою для досягнення множини цілей, котрі розв'язують проблему. Водночас каркас є високорівневою абстракцією проекту реалізації програмної системи.

За способом спеціалізації каркаси поділяють на два різновиди.

Каркас типу "біла скринька" має у своєму складі абстрактні класи, які неповно визначають мету та інтерфейси компонент. Щоб трансформувати такий каркас у конкретну прикладну систему, достатньо породити шляхом наслідування конкретні класи й доповнити при цьому визначення відповідних методів. Але цей процес потребує знання про внутрішню побудову каркаса, що суттєво утруднює його використання.

Каркас типу "чорна скринька" – це структура, окремі вузли якої означено як такі, визначення котрих відкладено (так звані "гарячі плями").

На відміну від попереднього типу каркаса, для заповнення "гарячих плям" пропонується спектр можливих альтернативних класів – претендентів зайняти місце у "гарячій плямі". Кожна пляма відповідає одному з аспектів змінюваності системи. Вибираючи один з альтернативних класів, ми отримуємо реалізацію відповідного йому варіанта системи.

Отже, в проектуванні системи передбачається її здатність змінюватися за окремими аспектами, фіксуються позиції можливих "збурень" у конструкції системи і передбачаються можливі варіанти. Створення системи виглядає як визначення конкретної конфігурації компонент, відповідних "гарячим плямам". Визначення конкретного варіанта системи проводиться як вибір "з полиці" потрібних компонент.

При цьому від конструктора конкретного варіанта вимагається лише знати перелік "гарячих плям", суть аспектів, варіантність яких визначається цими "плямами", і суть доступних варіантів за кожним із згаданих аспектів. Проектування системи у формі каркаса потребує від її конструктора спрямувати увагу на майбутні зміни, передбачити перелік таких змін, забезпечити максимальну локалізацію ефекту від втілення їх і в такий спосіб продовжити термін життя системи. Мабуть тому вживання каркасів стало поширюватись у новітніх технологіях програмної інженерії.

Селекція (добір) каркасів відбувається як неформальний, інтуїтивний процес зіставлення можливостей каркаса з вимогами цільової розробки.

Інтеграція каркасів можлива тільки шляхом включення їх як; підсистем у цільові розробки.

Прикладом каркаса є система банківських розрахунків, для якої визначено дві "гарячі плями": компонент ідентифікації рахунка клієнта (зазвичай у кожній країні діють відповідні стандарти) та компонент реакції на прохання клієнта зняти певну суму з його рахунка, якщо така сума перевищує наявну суму рахунка (окремі банки можуть у такому разі кредитувати клієнтів певних категорій або на певних умовах).

7.4 Створення повторно використовуваних компонент

7.4.1 Вияв потенційних ПВК

Раніше у підрозділі 7.1 було зазначено, що тільки систематичне повторне використання має перспективи стати ефективним засобом творення програм, тож у цьому розділі буде йти мова про компоненти як результат діяльності, спеціально спрямованої на можливість використання їх у багатьох прикладних застосуваннях.

Будемо називати групу подібних програмних продуктів, які належать до одного сегмента ринку (конкурують на ньому або доповнюють одне одного) лінією продуктів [7].

Лінія продуктів є найбільш перспективною нішею використання ПВК. Поряд з нею ПВК можуть використовуватись у кількох підсистемах одного прикладного застосування, у різноцільових застосуваннях, між якими можна виявити певні аналогії, як в абстрактних доменах (див. п. 7.3.6.).

Як засвідчують експерти, процес систематичного повторного використання може бути започатковано тільки після того, як буде створено певну кількість конкретних систем програмування з наведених вище різновидів, тобто буде напрацьовано базу для узагальнення (програмістські "легенди" показують, що ця кількість дорівнює в середньому трьом). За цих умов стандартний шлях виявлення ПВК становлять такі етапи:

- аналіз спільності та розбіжностей у створених досі системах;
- зіставлення вимог до нової системи з можливостями створених;
- прогнозування можливих змін у вимогах (до тих систем, що аналізуються) та виникнення потреб у нових системах;
- пошук компонент – кандидатів для повторного використання.

У попередніх розділах ми подали процес розроблення програмних систем як послідовне створення низки моделей (аналізу вимог, проекту, реалізації, тестування), сукупність яких дозволяє трансформувати побажання замовника в працюючу програмну систему.

Готова система постає перед тим, хто має зрозуміти й проаналізувати її як послідовність формалізованих моделей у відповідних нотаціях, властивих тому чи іншому методу (як-от подані у розділах 3 – 5 діаграми UML, І. Джекобсона чи С. Шлеєр та С. Меллора), а також як неформальні описи, коментарі до моделей та рекомендацій із застосування для користувачів.

Кожний з елементів згаданих моделей може бути потенційним кандидатом на повторне використання – актор, сценарій, інтерфейси між ними, об'єкти сценарію та відношення між ними, об'єкти, класи та підсистеми проекту, об'єкти та класи реалізації, діаграми переходів у стани та їхні фрагменти тощо. Перетворення названих елементів у ПВК відбувається шляхом узагальнення й перетворення їх на абстрактних акторів, абстрактні сценарії, абстрактні класи, підсистеми тощо.

При подальшому проектуванні абстрактні сценарії та підсистеми можуть втілюватися в такі категорії ПВК, як генератори, каркаси, класи об'єктів, тоді як більш детальні елементи моделей (класи, інтерфейси тощо) частіше відображаються в компоненти вихідного коду (див. підрозділ 7.3.). Ясно, що чим раніше на етапах життєвого циклу буде виявлено кандидата у ПВК, тим меншою буде відповідна йому когнітивна відстань (див. підрозділ 7.3.), тим більший виграш від ПВК можна чекати в майбутньому.

Побудова певної абстракції має на меті розширення спектра її придатності шляхом реалізації узагальненого розв'язання проблеми і

забезпечення засобів подальшої спеціалізації абстрактного елементу на кожний конкретний випадок застосування ПВК. Інакше кажучи, ми вважаємо кандидатами у ПВК такі елементи, для яких можна прогнозувати змінність (варіантність) певних властивостей і передбачити спектр можливих змін, адаптація до яких якраз і створює можливість повторного використання. Тому пошук кандидатів у ПВК є не лише визначенням необхідних абстракцій, а також і визначенням точок і характеру прогнозованих змін та відповідних механізмів адаптації до них.

Можна назвати властивості проекту, для яких доцільно зафіксувати варіантність сценарію та акторів [8]. Це такі властивості.

Варіантність інтерфейсів для категорій користувачів або для програмних систем. Приклади:

а) керівники різних рівнів потребують різнорівневих подробиць для видачі показників;

б) прикладні застосування відрізняються тим, що використовують різні СУБД;

в) різні інсталяції потрібні для окремих підрозділів організації, котрі працюють на різних конфігураціях устаткування.

Робота з кількома взаємозамінними типами об'єктів-сутностей. Приклади:

а) різноманітні типи приладів, котрі можуть вимірювати або регулювати однакові показники на конкретному "борту" літака в різних системах вимірювання, як-от відстань у кілометрах чи милях, пальне у літрах чи галонах тощо;

б) матриці з різною щільністю заповнення елементів (діагональні, стрічкові, розріджені тощо).

Функції, виконання яких в окремих випадках необов'язкове. Приклади:

а) клієнт, відправивши поштою оплату покупки, може інколи обумовити зворотне повідомлення про надходження платні;

б) під час тестування можна видавати чи не видавати контрольні значення;

в) дані в процесі введення можна контролювати чи не контролювати.

Обмеження на застосування або на правила бізнесу. Приклади:

а) в Україні правила обчислення податків часто змінюються;

б) правила перевірки прав доступу клієнта до банківського рахунка постійно вдосконалюються;

в) обчислення зарплати має свої особливості в окремих організаціях, а вимоги контролюючих органів до відповідної звітності також є змінним чинником.

Усунення помилок. Приклади:

а) коригування на основі гранично допустимих даних;

б) перехід до стану, що передував тому, в якому знайдено помилку

функціонування.

Гнучке настроювання на масштаби або швидкість оброблення, тобто необхідність регулювання співвідношення пам'ять/ швидкість. Прикладом може бути ситуація, коли час очікування відповіді на запит до системи визначається залежно від посади того, хто запитує, тобто відрізняється для керівників різних рівнів (чим вищий начальник, тим менше він згоден чекати відповіді на свій запит, а прискорення реакції системи здебільшого досягається за рахунок збільшення ресурсу пам'яті).

7.4.2 Специфікація варіантності вимог

Дослідивши елементи моделей готових систем, які було розроблено раніше, та визначивши потенційних кандидатів на роль ПВК, позначимо тепер конкретні місця можливих змін у наявних поданнях моделей кандидатів та способи адаптації згаданих моделей до таких змін.

Будемо називати *точками варіантності* текстуально виділені позиції в поданні моделей компоненти, для яких прогнозується можливість зміни і передбачаються засоби адаптації до них.

Точки варіантності позначаються на моделях. Точка в овалі сценарію позначає прогнозовану змінність певних властивостей або функцій сценарію.

Приклад. На рис. 7.1 подано об'єкт "Банківський рахунок" для деякої банківської системи. Система ідентифікації рахунків банку в кожній країні має свої особливості – в Японії вона відрізняється від прийнятої в Швейцарії, в Польщі – свої особливості тощо.

Точка в прямокутнику класу – змінність його атрибутів або операцій тощо.

Точкою варіантності V *1* на рисунку позначено прийняту для даної прикладної системи стратегію перевірки правильності вказаного ідентифікатора розрахункового рахунка, варіант якої залежить від того, в якій країні використовується система. Об'єкт "Банківський рахунок" має дві точки варіантності – V*1 і* V*2,* друга з яких позначає ту обставину, що прийнята стратегія поведінки банку підлягає змінам, якщо клієнт, вичерпавши свій кредит, намагається зняти гроші з рахунка.

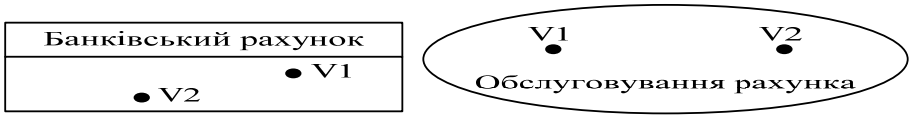

Рисунок 7.1 – Подання варіантів на елементах моделей

Для певних видів змінності широко застосовуються так звані патерни як штампи забезпечення локальності змін (див. підрозділ 7.3). Нагадаємо, що терміном “патерн” позначаються стандартні об'єктно-

орієнтовані рішення для стандартних проблем, описані стандартним способом. Патерни фіксують переважно перевірені часом штампи взаємодії об'єктів. Відзначимо низку таких штампів із числа наведених у джерелі [5], які корисні для деталізації сценаріїв у побудові ПВК:

– відокремлення інтерфейсу від обробки робить систему стійкою до змін інтерфейсу як найбільш динамічного аспекту вимог. Кілька можливих варіантів інтерфейсу подаються за допомогою патерну, названого обсервером, за яким інтерфейс уявляється як абстрактний об'єкт та сукупність об'єктів, котрі його успадковують (рис. 7.2).

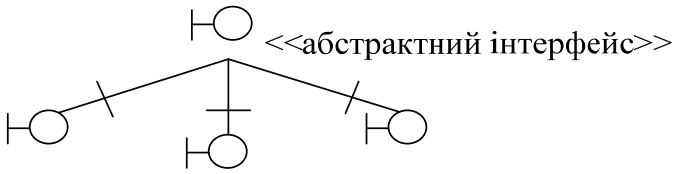

Рисунок 7.2 – Варіантність інтерфейсів

– варіантність використовуваних баз даних можна подати за допомогою патерну, названого мостом, за яким абстракція об'єкта (узагальнення доступу до баз даних) відокремлюється від своєї реалізації (звернення до конкретної СУБД);

– розширення функції може бути подане патерном декоратора, за яким складний сценарій розбивається на сукупність простіших, кожний з яких є розширенням функції абстрактного об'єкта, названого декоратором (приклад об'єкта-декоратора подано на рис. 7.3).

Рисунок 7.3 – Варіантність функцій

– якщо до системи входять кілька досить великих підсистем, котрі мають велику кількість класів об'єктів, доцільно використати патерн фасаду, за яким інтерфейс з підсистемою доцільно зосередити в спеціальному класі. Поняття фасаду відіграє велику роль у процесі повторного використання, тому зупинимося на ньому докладніше.

Звичайно, підсистему можна розглядати як систему компонент з узгодженою взаємодією, які можуть спільно функціонувати, успадковувати одне одного. Для користувача така система виглядає як певний комплекс послуг, опис яких власне і подається як фасад.

Ми кажемо, що прикладна система ревикористовує готові

компоненти і системи компонент шляхом імпорту їхніх властивостей. Нагадаємо, що властивості, які імпортуються, становлять видиму частину абстракції.

При цьому для кожного випадку ревикористання ПВК звичайно імпортує, а прикладна система, відповідно, експортує тільки потрібну в даному разі частину властивостей, якими володіє ПВК. Таким чином, співвідношення видимої і прихованої частин абстракції може змінюватися для кожного випадку ревикористання.

Видима для даного випадку сукупність властивостей є власне фасадом. Фасад являє собою механізм доступу до тих властивостей системи компонент, які потрібні для конкретного ревикористання. Наприклад, система компонент, що реалізує доступ до баз даних, може мати фасадом чотири стандартних операції – "читати," "писати", "створити", "знищити". Деякі важливі операції, такі, як перевірка правильності при введенні, відновлення, підтримка мультидоступу можуть, залежно від використання, належати до прихованої частини або бути виведеними на фасад. Інші внутрішні властивості баз даних може бути інкапсульовано.

На фасад може бути виведено будь-які знання про ПВК, наприклад, її сценарій, інструкції з використання тощо. Фасад – це погляд на систему компонент того, хто її використовує. Побудовано фасад так, щоб втілити відомий принцип Парнаса "приховування непотрібних знань".

Одна система компонент може мати кілька фасадів – якість фасаду визначає успіх ревикористання. Фасад має бути стабільним, а те, що в нього не ввійшло, може змінюватися (наприклад, використання конкретного сервера СУБД). Приклад фасаду подано на рис. 7.4 в нотації UML.

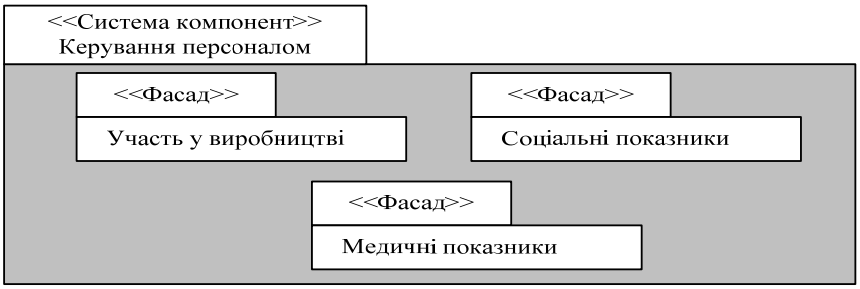

Рисунок 7.4 – Приклад фасаду

7.4.3 Конкретизація варіантності вимог

Конкретизація варіантної точки може здійснюватися як вибір однієї з визначених наперед можливостей (як, наприклад, згадувана вище система ідентифікації банківського рахунка) або як довизначення користувачем адаптованого до конкретного застосування екземпляра ПВК.

Звичайно, фасад системи містить відомості про наявність варіантних точок і засобів їхнього довизначення.

Будемо називати вирішенням варіантної точки ПВК механізм визначення конкретного варіанта використання ПВК та її адаптації до такого використання.

Можна навести типові механізми вирішення варіантної точки ПВК, тобто її адаптації або настроювання.

Механізм *успадкування (inheritance)*. Окремі властивості поведінки суперкласу конкретизуються в підкласі. Точний зміст цього процесу значною мірою визначається конкретною об'єктно-орієнтованою мовою програмування. На рівні стадії аналізу вимог вирішення варіантності шляхом наслідування позначається на моделі сценаріїв відношеннями "розширює" та "використовує".

Відношення розширення сценарію (див. підрозділ 3.3) доцільно застосовувати, якщо функції та поведінка одного сценарію (суперкласу) доповнюється функціями та поведінкою іншого (підкласу). Це розширення може бути подано явною варіантною точкою (наприклад, якщо деталізацію певного рішення відкладено на наступну фазу життєвого циклу). Або неявною, коли посилаються на окремий документ, в якому наводяться правила розширення.

Звичайно, сценарій уточнюється за кілька ітерацій, на кожній з них визначений раніше сценарій може бути деталізовано і доповнено новими функціями. Таке доповнення зручно відобразити в моделі сценаріїв відношенням розширення. Розширення, по можливості, не повинно змінювати розширюваний сценарій, тоді досягається стабільність сценарію-суперкласу, бо він не залежить від своїх розширень.

Відношення використання доцільно застосовувати, якщо один сценарій включається повністю (без будь-якого настроювання) до іншого. Важливо розрізняти кілька самостійних сценаріїв та кілька варіантів того самого сценарію.

Механізм конфігурації. Механізм конфігурації застосовується тоді, коли точка варіантності позначає можливий вибір із сукупності передбачених наперед взаємозамінних компонент (у тому числі на рівні сценаріїв). Приклад вирішення варіантності шляхом конфігурації наведено на рис. 7.1, де показано об'єкт "Банківський рахунок" з двома варіантними точками, для кожної з яких передбачено вибір однієї з передбачених альтернативних можливостей, що зумовлено конкретними обставинами використання об'єкта.

Механізм конфігурації, як і успадкування, безпосередньо використовується у ПВК, що належать до категорії каркасів (див. п. 7.3.8.)

Включення тих чи інших компонент або систем компонент (елементів конфігурації) в загальну композицію прикладної системи для складних великих ПВК може породити велику кількість версій такої

композиції, залежно від конкретних обставин застосування. Тому управління конфігурацією розглядається як окрема гілка в дереві знань програмної інженерії, і ступінь автоматизації цього процесу вважається навіть як один з показників оцінки технологічної зрілості організації розробників.

Механізм настроювання за параметрами. Варіантна точка може означати, що вирішення варіантності передбачено виконувати шляхом задания параметрів настроювання. В сучасних системах програмної інженерії настроювання за параметрами може бути також достатньо складним. Для конструкції процедур у класичних мовах програмування настроювання за параметрами потребувало заміни формального значення параметра на вказане фактичне, причому всюди, де було текстуально локалізовано формальне значення, тобто реалізація зводилася до пересилання фактичних значень на поля пам'яті, відведені для формальних параметрів. У сучасних засобах створення програмних систем використовується цілий спектр достатньо складних типів настроювання за параметрами.

Серед них відзначимо такі:

а) параметром значення змінної або список значень. Настроювання здійснюється шляхом встановлення відповідних значень змінних;

б) параметром є ім'я або список імен. Настроювання здійснюється шляхом підставляння імені як фактичного значення параметра замість входження формального параметра до вихідного тексту програми;

в) параметром є явна умова прийняття рішення та умова обмеження. Настроювання здійснюється шляхом підставляння умовного виразу замість входження формального параметра до вхідного тексту програми;

г) параметром є неявна умова прийняття рішення або умова обмеження, тлумачення якої визначається станом бази знань. Настроювання вимагає виведення явної умови на підставі бази знань і наступної підстановки як у випадку 3;

д) параметром є вираз, який визначається шляхом генерації. Настроювання потребує спеціального генератора;

е) параметром є організація носіїв даних. Настроювання потребує виклику відповідних серверів (серверів баз даних, файлів тощо);

ж) параметрами є імена послуг та функцій, що використовуються. Настроювання здійснюється шляхом звернень до відповідних ПВК.

и) параметром є специфікація сервера, що використовується. Настроювання здійснюється шляхом реалізації сервера;

к) параметром є специфікація інтерфейсу. Настроювання здійснюється шляхом використання інструментальних засобів конструюван ня інтерфейсу;

л) параметрами є показники середовища виконання. Настроювання здійснюється шляхом виконання серверів інсталяції;

м) параметрами є необхідні виміри під час виконання ПВК. Настроювання здійснюється шляхом генерації коду;

н) параметрами є показники якості виконання (швидкодія, надійність тощо). Настроювання здійснюється шляхом генерації коду;

п) параметром є клас об'єкта, що використовується в родовому класі (в шаблоні або контейнерному класі). Настроювання здійснюється спеціальними засобами мови програмування;

р) параметром є певне поняття проблемної галузі. Настроювання здійснюється шляхом виведення його тлумачення з бази знань.

Перелік можливих випадків вирішення варіантності за допомогою визначення параметрів постійно нарощується з появою нових засобів генерації коду.

Коли ми розв'язуємо питання про доцільність використання системи компонент, ми зіставляємо її варіантні точки з тими, яких потребує створювана нами система. Якщо вони збігаються, ми імпортуємо всю систему компонент, як на рис. 7.5. Якщо ні – шукаємо інших кандидатів на використання.

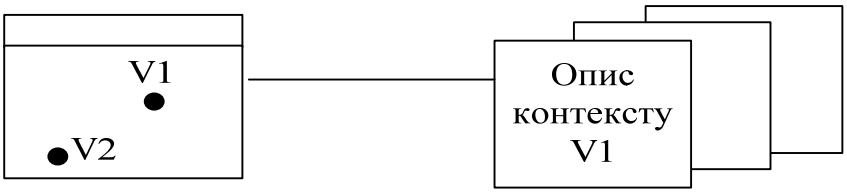

Рисунок 7.5 – Випадок використання системи компонент

Якщо на рівні прикладної системи ми фіксуємо лише один варіант конфігурації, то здійснюємо спеціалізацію відразу, бо підтримка варіантності в проекті завжди потребує ресурсів пам'яті і часу, а якщо варіантів не вистачає, додаємо нові, як на рис. 7.6.

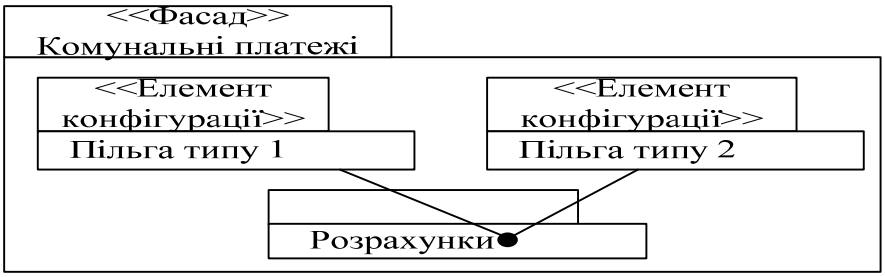

Рисунок 7.6 – Варіанти конфігурації

Коли ми створюємо систему компонент для повторного використання, то, як ми вже зазначали, умовою успіху є можливість потенційного користувача швидко зрозуміти, яку з його проблем покриває дана система компонент. Документування ПВК за допомогою фасаду та

варіантних точок з визначеним механізмом вирішення їх дає лаконічне уявлення про властивості ПВК, визначальні для її використання. Інакше кажучи, для того щоб використати систему компонент, ми маємо якомога раніше врахувати всі можливості, які вона надає, тобто розв'язати проблему її використання на стадіях аналізу та проектування, формулюючи сценарії використання нової системи в термінах фасаду системи компонент (його об'єктів, точок варіювання тощо).

Рисунок 7.7 ілюструє цю тезу. Видима частина – незаштрихована, біла.

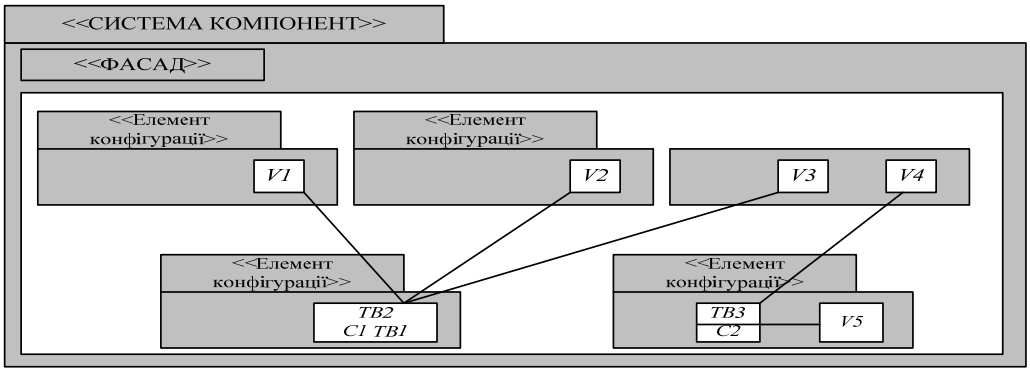

Рисунок 7.7 – Варіанти конфігурацій

Приклад. Новий банківський консорціум, використовуючи можливості Internet, організовує "банк вдома" – тобто електронний банк, який обслуговує фінансування маркетингу, умови котрого швидко змінюються, тому реалізується не монолітна структура, а система як композиція компонент, що допускають адаптацію. Можлива розподілена архітектура клієнт-сервер або система рівноправних компонент. Клієнтські компоненти працюють на персональних машинах або через Internet. Малі оплати можуть здійснюватися через Internet. "Банки вдома" мають спілкуватися з іншими фінансовими об'єктами, можливо, програмне забезпечення для "банку вдома" може мати спільні ПВК з кредитними, страховими, телефонними та іншими компаніями.

Понятійна база для цього домену включає цілу низку понять, що відображають типово банківські операції ("зареєструвати", "вкласти", "зняти з рахунку", "перевести на інший рахунок") та об'єкти, з якими вони оперують (позичка, кредит, акції, вклад, закладна, накладна, страховка, вексель, портфель цінних паперів, гроші готівкою, електронні гроші, чек, відсотки тощо). Тоді рис. 7.7 може бути ілюстрацією до цього прикладу, якщо вважати:

- *V1* позначає елемент конфігурації, який здійснює блокування видачі, якщо рахунок вичерпано;
- *V2* позначає елемент конфігурації, який здійснює підвищення

відсоткової ставки, якщо кредит перевищує залишки рахунка;

- *V3* позначає елемент конфігурації, який здійснює застосування гнучкої стратегії кредитування з урахуванням ризику;
- *C1* позначає елемент конфігурації, який здійснює ідентифікацію рахунка клієнта;
- *TB1* є варіантною точкою, яка визначає середовище клієнтських компонент;
- *TB2* є варіантною точкою, яка визначає поведінку банку, якщо замовлена виплата перевищує залишки на рахунку;
- *TB3* є варіантною точкою для ідентифікації особистого рахунка клієнта;
- *C*2 є модуль контролю ризику.

Незаштрихована частина рисунка відповідає видимій інформації.

7.4.4 Проектування ПВК

Базою для пошуку компонент і систем компонент є сукупність напрацьованих прикладних систем та готових систем компонент. Означена сукупність розглядається як єдина система, яку будемо називати суперсистемою. До її фасаду включається тільки та інформація, яка стосується використання кожної із складових суперсистеми.

Домовимося, що суперпозиція акторів і сценаріїв складових суперсистеми є першим наближенням акторів та сценаріїв власне суперсистеми (рис. 7.8).

Рисунок 7.8 – Використання суперсистеми

Наступним кроком є узагальнення включених сценаріїв та акторів і побудова абстрактних сценаріїв та акторів суперсистеми в такий спосіб, щоб сценарії та акторів кожної із складових систем було поділено на дві категорії – на ту, що є конкретизацією сценарію чи актора суперсистеми шляхом вирішення точки варіантності, та на ту, що є індивідуальною властивістю конкретної складової суперсистеми і не використовуються в інших складових (рис.7.9).

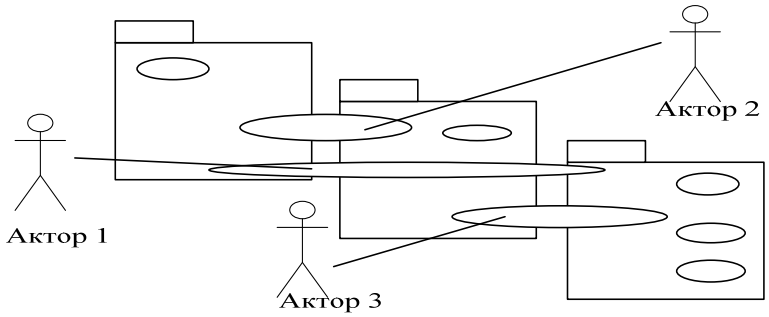

Рисунок 7.9 – Приклад розташування сценаріїв у суперсистемі

Перші з них є кандидатами у ПВК. Щоб перетворити їх на справжні ПВК, потрібно вдосконалити їхні інтерфейси, можливо, більш чітко визначити їхні фасади і правила та обмеження застосування. Якщо не вдається абстрагувати сценарій у цілому (тобто виділити ПВК-кандидатів на стадії аналізу вимог до систем), проводимо дослідження моделей проектів систем, включених нами до складу суперсистеми з метою виявлення тих її елементів, які можуть вважатися кандидатами у ПВК (абстракції класів, моделей переходів у стани тощо).

У процесі побудови моделі проекту суперсистеми визначаються й уточнюються такі аспекти:

– взаємодія прикладних систем, охоплених суперсистемою;

– класи ПВК-проектування, типи підсистем, утворені з них системи компонент;

– фасади систем компонент та ті їхні властивості, які імпортуються за допомогою фасадів;

– розподіл функцій сценаріїв, визначених для моделі суперсистеми, між охопленими нею прикладними системами та системами компонент.

Треба зауважити, що на першій стадії архітектурної роботи компоненти і фасади не можуть бути визначені одразу. Вони уточнюються разом із стабілізацією архітектури.

Модель проекту суперсистеми визначається щодо шарів, наведених на рис. 4.2, кожна підсистема належить одному шару і залежить тільки від компонент цього шару або шару, який розташовано нижче.

Наступний крок – виявлення загальнозначущих акторів. Якщо суперсистема взаємодіє з певними акторами, то конкретні прикладні застосування та підсистеми, для яких вона є узагальненням, мають також взаємодіяти з ними. Підсистеми нижнього рівня, які імпортує система, не експортують своїх акторів, бо актор – це зовнішня відносно до системи сутність, а імпортовані властивості є внутрішніми відносно до підсистем, які їх використовують.

Коли ми говорили про складові суперсистеми, то мали на увазі системи, які на сучасному рівні (з побудовою всіх необхідних моделей

аналізу, проекту тощо) реалізують окремі прикладні застосування та відповідні системи компонент. Але, окрім таких складових, для побудови суперсистеми часто необхідно враховувати так звані успадковані системи.

Успадкованою системою (system inheritance) називають діючу систему, створену застарілими (небажаними або навіть невідомими для команди розробників) засобами й технологіями проектування, яку, однак, усе одно експлуатують для управління значущими даними бізнесу і підтримки процесів бізнесу, бо вона задовільно виконує свої функції, а її модифікація потребує чималих коштів.

Прикладом може бути база даних великого обсягу, зібрана за допомогою застарілої СУБД, дані якої все ще потрібні.

Суперсистема, яка використовується як узагальнення спектра завдань домену, має враховувати і ті завдання, які поданоуспадкованими системами.

Для досягнення цієї мети почнемо з того, що побудуємо оболонку, яка інкапсулює послуги успадкованої системи й забезпечує взаємодію з нею в термінах сучасних моделей розроблення систем.

Така оболонка виконує роль транслятора інтерфейсів успадкованої системи в інтерфейси інших складових суперсистеми. Тоді ця оболонка розглядається як система компонент і вбудовується в суперсистему. Для неї будується фасад, через який експортуються її послуги на всіх фазах життєвого циклу нових систем, котрі взаємодіють з успадкованою системою. Нові системи, які її використовують, розглядають її як підсистему з віддаленими об'єктами, що визначаються оболонкою і якими нова система маніпулює шляхом звернення до віддаленої підсистеми. Для однієї успадкованої системи може бути побудовано кілька оболонок і кілька фасадів. На рис. 7.10 подано приклад використання успадкованої системи розрахунків, зробленої мовою програмування Кобол.

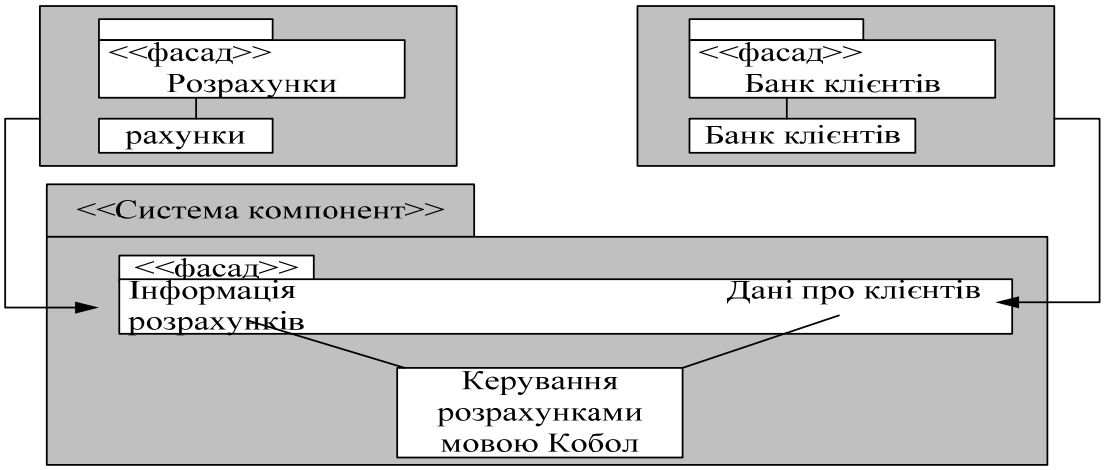

Рисунок 7.10 – Приклад використання успадкованої системи

7.5 Інформаційне забезпечення повторного використання

Якими б корисними не були створені ПВК, їх можуть знайти їхні потенційні користувачі тільки при умові, що хтось поінформує їх про наявність та властивості ПВК (див. підрозділ 7.1).

Назвемо апарат інформування потенційних користувачів про наявність, властивості та умови використання ПВК *інформаційним забезпеченням повторного використання (dataware reuse)*. Його основне завдання – навігація користувачів у просторі колекцій ПВК, котрі розрізняються за призначенням, категоріями (див. підрозділ 7.3) та поданням, належністю до різних доменів (проблемної галузі). Мета такої навігації – знайти готові рішення, що відповідають вимогам цільової системи, яку має створити користувач.

Згадаємо як організовано інформаційне обслуговування відвідувачів бібліотеки. Кожний примірник книги (об'єкта зберігання) має свою інформаційну картку в каталозі бібліотеки, котра є інформаційною моделлю книги і виконує роль так званого пошукового образу – моделлю відомостей, за якими інформаційна система виконує обслуговування, зіставляючи поданий читачем так званий пошуковий запит (відомості про об'єкти, які він шукає) з тими пошуковими образами, що відповідають діючим об'єктам зберігання. Так само для ПВК як об'єктів зберігання має бути побудовано відповідні каталоги.

Успіх пошуку звичайно забезпечується якістю каталогів, зокрема інформаційною моделлю, на якій їх побудовано.

Інформаційна модель ПВК, за допомогою якої має бути подано інформаційний образ окремої ПВК у каталозі (пошуковий образ ПВК), орієнтована на розуміння людиною функцій ПВК (інколи досить складних) та на можливість зіставлення їх з власними потребами. Очевидно, таке завдання набагато складніше від бібліотечного пошуку, коли в ролі інформаційної моделі звичайно виступає ім'я автора, назва або жанр книжки тощо.

Можна назвати кілька відомих підходів до побудови інформаційної моделі ПВК [9, 10]:

– пошуковим образом ПВК може бути список ключових слів, які найчастіше згадуються в тексті ПВК. Це так званий спосіб вільного індексування;

– пошуковий образ ПВК може бути створено на базі заздалегідь побудованої онтології домену проблемної галузі (див. п. 3.2.1.). Це спосіб ієрархічної класифікації, прикладом якого є універсальна десяткова класифікація – УДК, що використовується, зокрема, для каталогізації наукових публікацій: окремим галузям знань присвоюються унікальні номери, так само робиться для їхніх підгалузей до певної глибини, між якими існує відношення рід – вид. Тоді кожна публікація в науковому

виданні маркується (кажуть, індексується) послідовністю номерів, розділених крапками, що позначають відповідні галузь, підгалузь, розділ тощо і до тематики яких належить дана публікація. Наприклад, індекс УДК681.3 позначає розділ кібернетики "системи програмування", і якщо хтось хоче знайти наукові публікації з даної теми, він може вказати індекс УДК у пошуковому запиті, якщо в бібліотеці є тематичний каталог, пошукові образи котрого мають у своєму складі індекс УДК. УДК є найпоширенішою міжнародною системою класифікації для точних, технічних та природничих наук у світі. Онтології доменів також є прикладами ієрархічних класифікацій, хоча треба зазначити, що онтологію певного домену може бути створено як кілька ієрархій, кожну зі своєю основою;

– пошуковий образ ПВК може складатися з визначених наперед порцій (фасет), кожна з яких належить до окремого аспекту розгляду ПВК. Це є фасетний спосіб. Усередині фасети інформацію можна структурувати на окремі поля, значення яких можуть належати до певної онтології або вільного (наперед не визначеного) списку дескрипторів (термінів), або до одного з наперед визначених альтернативних значень. Тобто, фасетний спосіб може поєднувати в собі можливості двох наведених вище способів.

Кожний із способів, який може бути обрано для побудови інформаційної моделі ПВК, потребує, з одного боку, певного обсягу та ступеня кваліфікації праці, що вкладається попередньо для створення каталогу, а з іншого боку, забезпечує досягнення тієї чи іншої якості пошуку. Будемо вважати критеріями якості пошуку, по-перше, можливість отримати (внаслідок пошуку в каталозі) інформацію, повністю або частково відповідну (кажуть, релевантну) інформаційному запиту, а по-друге, швидкість отримання відповіді.

Спосіб вільного індексування може бути реалізовано з мінімальними витратами праці фахівців при мінімальній кваліфікації. ПВК, що вносяться до каталогу, можуть індексуватися автоматично тими термінами, які згадуються в їхніх описах. Але при цьому є імовірність, що терміни, які було використано для опису ПВК її автором, і терміни, котрі потенційний користувач буде вживати для вираження своєї потреби під час пошуку підходящої ПВК, не збігаються через розбіжність та неузгодженість онтологій, якими користується кожна з діючих сторін.

Спосіб ієрархічних класифікацій потребує попереднього проведення аналізу проблемних галузей експертами високої кваліфікації, здатними побудувати онтології, що відображають специфіку завдань доменів. Отримані ними онтології можуть служити базою для утворення ієрархічних меню, котрі керують формулюванням запитів на пошук ПВК. У цьому разі істотно знижується ефект неузгоджених термінів, а пошук зводиться до вибору однієї з поданих можливостей. Однак цей підхід потребує інвестицій як для початкової побудови онтологій, так і для

супроводження та модифікації їх.

Фасетний спосіб також вимагає попередньої експертної підготовки. До його переваг слід віднести можливість звуження кола пошуку за рахунок багатоаспектних критеріїв пошуку.

Слід зазначити, що витрати на побудову онтологій має бути виправдано успішним пошуком, без якого взагалі немає сенсу говорити про повторне використання. Справді, нині маємо бібліотеки модулів мовами програмування, які налічують тисячі об'єктів зберігання. При цьому їх каталоги структуровані переважно лише назвами доменів використання (наприклад, обчислювальні функції, функції роботи з екранами, функції введення-виведення тощо), до того ж каталоги бібліотек фірм-розробників можуть використовувати різні назви для однакових доменів.

На сьогодні існує досить значна множина доменів, досвід роботи з якими дозволяє визначити їхні онтології, стандартизація котрих потребує переважно організаційних зусиль, після чого поняття, що входять до їхнього складу, можуть однозначно трактуватися. Саме така робота тепер проводиться у світі [11].

Джерелом знань для побудови онтологій можуть бути формалізовані вимоги на розроблення ПВК як програмних систем, способи отримання яких розглянуто нами в розділах 3,4,5.

Наявність спектра моделей для відображення різних аспектів вимог (властивості класів об'єктів, їхніх зв'язків, динаміка поведінки тощо) дає підстави для поділу пошукового образу ПВК на фасети. Ще одним джерелом виявлення фасет є традиції побудови окремих інструментів для окремих аспектів створення програмних систем (таких, як інтерфейси користувача, композиції компонент, розподілена обробка тощо), що дозволяє ставити у відповідність таким аспектам окремі фасети пошукового образу ПВК. Такий підхід дає можливість визначити незалежні напрями адаптації ПВК і спростити розуміння їхніх властивостей.

Інформаційно-пошукові системи зберігання й пошуку пошукових образів ПВК мають спеціальну назву репозиторіїв компонент повторного використання.

Контрольні запитання і завдання

1. Назвіть дві характерні властивості повторно використовуваних компонент (ПВК).
2. Назвіть головні процеси життєвого циклу розробки, властиві систематичному повторному використанню.
3. Охарактеризуйте основні властивості ПВК, поєднання яких визначає її категорію.

4. Охарактеризуйте основні властивості ПВК для окремих категорій:

а) мов програмування;

б) генераторів програм;

в) бібліотек програм;

г) каркасів;

д) абстрактних доменів.

5. Як виявляється й подається варіантність проектних рішень під час створення ПВК?

6. Якими механізмами конкретизуються варіанти рішень у реальних проектах?

7. Охарактеризуйте підходи до організації пошуку потрібних ПВК.

8. Визначте процес проектування ПВК.

9. Наведіть методи побудови інформаційного забезпечення ПВК.

10. Що лежить в основі створення ПВК ?

11. Назвіть головну проблему у створенні бібліотеки ПВК.

12. Наведіть приклади ПВК абстрактних архітектур.

13. Що таке селекція або вибір генератора?

14. Визначте типи та приклади різновидів генераторів широкого поширення.

15. Назвіть базові ієрархії моделей систем об`єктів.

16. Які аспекти визначаються та уточнюються в процесі побудови моделі проекту?

17. Що таке успадкована система?

18. Визначте відомі підходи до побудови інформаційної моделі ПВК.

19. Назвіть основні кроки проектування ПВК.

20. Як використовується механізм успадкування для вирішення варіативної точки ПВК?

ПІСЛЯМОВА

Навчальний посібник має обмежений обсяг, що не дозволяє висвітлити в ньому усі теоретичні та практичні питання проектування ПЗ сучасних систем управління. Ця галузь комп'ютерної науки зараз швидко розвивається, помножуючи щороку удвічі об'єми теоретичного матеріалу та інструментальних засобів розробки. Тому даний навчальний посібник можна вважати тільки загальним введенням до дисципліни проектування програмних систем і програмного забезпечення.

Посібник дозволяє студентам самостійно вивчати такі розділи курсу "Проектування програмних засобів систем управління", як "Базові поняття програмної інженерії", "Інженерія вимог", "Основи об'єктно-орієнтованого проектування програмних систем", "Метод UML як стандартний засіб моделювання в програмній інженерії" та "Повторне використання в програмній інженерії".

На жаль, в посібник не увійшли інші важливі розділи цієї дисципліни, а саме: "Розширення методу UML для моделювання програмних систем реального часу", "Інструментальні засоби підтримки методу UML", "Проектування ПЗ інформаційних та інформаційно-управляючих систем", "Проектування ПЗ систем управління бізнес-процесами". Цей матеріал студенти вивчають на лекційних заняттях та під час виконання лабораторних робіт, а також в рамках самостійної роботи над відповідними методичними матеріалами.

Матеріал навчального посібника буде використовуватися студентами при вивченні інших спеціальних дисциплін на старших курсах, а саме: "Системне програмування", "Проектування комп'ютерних систем управління", "Цифрові системи". Крім того, посібник буде корисним тим студентам, що виконуватимуть за певною тематикою бакалаврські дипломні роботи та дипломні проекти спеціаліста.

Література

Література до розділу 1

1. McConnel S.,Tripp L. Professional Software Engineering: Fact or Fiction? // IEEE SOFTWARE. – 1999. –Nov. – Dec. – P. 13–18.

2. David L., Parnas Software Engineering Programs Are Not Computer Science Programs // IEEE SOFTWARE. – 1999. – Nov. – Dec. – P. 19–30.

3. http:/www.swebok.org.html.

4. Jackson M. Software requirements & specifications. – Wokingham, England: Addison-Wesley, ACM Press Books, 1995. – 228 p.

5. Merriam-Webster's New Collegiate Dictionary, 10th Edition.

6. Брукс П. Мифические человеко-месяцы. – М.: Мир, 1972. – 234 с.

7. Jotterbarn D., Miller K., Rogerson S. Software Engineering Code of Ethics is Approved // Communications of the ASM. – 1999. – V. 42. – № 10. – P. 102–107.

Література до розділу 3

1. http:/www-ksl.stanford.edu.html

2. Шлеєр С., Меллор С. Объектно-ориентированный анализ: моделирование мира в состояниях. – К.: Диалектика, 1993. – 240 с.

3. Jacobson I., Griss M., Jonsson P. Software Reuse. – N.-Y.: Addison-Wesley, 1997. – 497 p.

4. http:/www.rational.com.uml.html

5. Андон А. И., Яшунин А. Е., Резниченко В. А. Логические модели интеллектуальных информационных систем. – К.: Наук. думка, 1999. –320 с.

Література до розділу 5

1. http:/www.rational.com.uml

2. Фаулер М., Скотт К . UML в кратком изложении. – М.: Мир, 1999. – 200 с.

3. Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя. – М.: ДМК, 2000. – 430 с.

4. Боггс Ч., Боггс М. UML Rational. – М: Лори, 2000. – 561 с.

Література до розділу 7

1. Бабенко Л. П. Повторное использование в программной инженерии // Кибернетика и системный анализ. –1999. – № 2. – С. 37–48.

2. Sutclife A. and Maiden N. The Domain Theory for Requirement Engineering //IEEE Trans, on Software Engrg. –1998. – V. 24. – № 3. – P. 174–190.

3. Бабенко Л. П. и др. Адаптивные компоненты повторного использования в системах генерации программ // Кибернетика и системный анализ. – 1991. – № 5. – С. 145–159.

4. Сахаров А. Л. Принципы проектирования и использования многомерных баз данных // СУБД. – 1996. – № 3. – С. 44–59.

5. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приёмы объектно-ориентированного проектирования. Патерны проектирования – СПб: Питер, 2001. — 368 с.

6. Schmid H. A. Creating Applications from Components: A Manufacturing Framework Design // IEEE Software, 1996. – November. – P. 67–75.

7. Keepence B., Mannioni M. Using patterns in Model Variability in Product Families // IEEE Software –1999. – № 3. – P. 10.2–108.

8. Jacobson I., Griss M., Jonsson P. Software Reuse. – N.-Y.: Addison-Wesley, 1997. – 497 p.

9. Бабенко Л. П., Поляничко С. Л. Модели классификации объектов программной инженерии // Проблемы программирования. – Вып. 1. – К.: Ин-т программных систем НАН Украины, 1997. – С. 25–32.

10. Лаврищева Е. М., Грищенко В. Н. Сборочное программирование. – К.: Наук. думка, 1991. – 216 с.

11. http:/www-ksl.Stanford.edu.html

12. БенАри М. Языки программирования. Практический сравнительный анализ. – М.: Мир, 2000. – 366 с.

ДОДАТОК А

Глосарій

Абстрактна архітектура *(abstract architecture)* – декомпозиція рішення для виділеного спектра завдань домену на підсистеми або ієрархію підсистем, на кожному рівні якої фіксуються можливі варіанти виділених параметрів і обмежень, що визначають відповідні варіації складу виділених компонент.

Агрегація *(aggregation)* – об'єднання кількох понять у нове поняття, істотні ознаки нового поняття при цьому можуть бути або сумою ознак компонент або суттєво новими (відношення агрегації часто ототожнюють з відношенням "частка – ціле").

Активні домени *(active domain)* – домени, які можуть переходити від стану до стану без зовнішніх стимулів, як це робиться з людиною чи атмосферою.

Актори *(actors)* – чинні особи, для яких створюється система.

Аналіз вимог *(requirement analysis)* – відображення функцій системи та її обмежень у моделі проблеми.

Артефакт *(artifact)* – будь-який продукт діяльності фахівців з розробки програмного забезпечення.

Архітектура програмної системи *(program system architecture)* – визначення системи в термінах обчислюваних складових (підсистем) та інтерфейсів між ними, яке відображає правила декомпозиції проблеми на складові.

Валідація (*validation***)** – забезпечення відповідності розробки вимогам її замовників.

Верифікація (*verification***)** – перевірка правильності трансформації проекту в код реалізації.

Відмова (*failure***)** – відхилення програми від функціонування або неможливість виконувати функції, визначені вимогами та обмеженнями, тобто перехід програми в нероботоздатний стан у зв'язку з помилками у програмі або аварією в середовищі функціонування.

Гарантія (підтвердження) якості програмного забезпечення *(software warranty)* – дії на кожній стадії життєвого циклу з перевірки і підтвердження відповідності стандартам та процедурам досягнення якості.

Дефект *(fault)* – результат помилок розробника у вхідних або проектних специфікаціях, текстах кодів програм, експлуатаційній документації тощо.

Динамічні методи тестування *(dynamic mode testing)* – виконання програм з метою встановлення причин помилок з очікуваними реакціями на ці помилки.

Діаграма *(diagram)* – графічне подання елементів моделювання системи (класів, сценаріїв, поведінки, послідовності, співробітництва, активності, станів, реалізації, компонент тощо).

Домен предметної галузі *(data domain)* – спектр завдань предметної галузі, які допускають схожі способи їх вирішення.

Екземпляризація *(specimenation)* – залежність між параметризовании абстрактним класом-шаблоном (template) і реальним класом, який ініційовано шляхом визначення параметрів шаблону.

Експлуатація або супроводження *(maintenance)* – дії з використання готової програмної системи.

Ефективність ресурсів *(resource effectiveness)* – атрибути, які показують кількість ресурсів і тривалість використання їх при виконанні функцій програми в програмному забезпеченні.

Ефективність системи *(system effectiveness)* – множина атрибутів раціональності, які показують взаємозв'язок між рівнем виконання програмного забезпечення і кількістю використовуваних ресурсів (засоби, аппаратура та ін.).

Загроза *(threat)* – прояв нестійкості, що порушує безпеку системи.

Захищеність *(protectability)* – атрибут функціональності, який вказує на можливість запобігати несанкціонованому доступу (випадковому або навмисному) до програм і даних.

Зрозумілість (*indelibility*) – атрибут зручності, який визначає зусилля, необхідні для розпізнавання логічних концепцій та умов їх застосування.

Інертні домени *(inertial domain)* – домени, зміна станів яких ніколи не відбувається з ініціативи об'єктів домену, а реалізується під дією зовнішніх факторів.

Інженерія *(engineering)* – застосування наукових результатів, яке дозволяє отримувати користь від властивостей матеріалів та джерел енергії.

Інтервал стабільності *(stability interval)* – інтервал часу, протягом якого не відбувається зміна стану.

Інформаційна система *(information system)* – система, яка проводить збирання, оброблення, зберігання та вироблення інформації людьми з використанням автоматичних процесорів.

Інцидент (*incident)* – абстрактна подія, яка впливає на зміну стану об'єкта.

Каркаси (*frame***)** – різновид абстрактних архітектур або частково визначена абстрактна архітектура з відкладеним довизначенням виділених компонент.

Керування якістю *(quality control)* – комплекс засобів та системної діяльності з планування, забезпечення й оцінювання якості програмного

забезпечення.

Когнітивна відстань (*cognitive distance)* – кількість інтелектуальних зусиль, які необхідно затратити розробникові програмного забезпечення для того, щоб перевести систему, що розробляється, з однієї стадії життєвого циклу в іншу.

Кодування *(coding)* –див.: **Реалізація програмної системи.**

Компонента (*multiplier)* – тип, клас, проектне рішення, документація або інший продукт програмної інженерії, спеціально пристосований для повторного використання.

Компонентна розробка *(component development)* – конструювання програмного забезпечення з конструкцій за каталогом як композиції готових компонентів.

Конфігурація *(configuration)* – склад програмної системи, створений шляхом вибору окремих примірників із визначеного заздалегідь набору варіантів модулів.

Концептуальне моделювання *(conceptual modeling)* – процес побудови моделі проблеми, орієнтованої на її розуміння людиною.

Метрика програм *(program metric)* – система визначених кількісних показників програм та шкали вимірювання їх.

Моделі життєвого циклу – типова схема послідовності робіт на етапах розроблення програмного продукту.

Модель процесів *(model process)* – визначення певних дій, які супроводжують зміни станів об'єктів.

Модель станів *(model stane)* – відображення динаміки змін стану кожного з класу об'єктів, які змінюють свою поведінку.

Модель функції-дані *(data-function model)* – декомпозиція проблеми на послідовність функцій та даних, які обробляються цими функціями.

Модель якості *(quality model)* – чотирирівнева модель, яка відображає характеристики, атрибути (показники), метрики, оціночні елементи програмного забезпечення.

Надійність *(reliability)* – множина кількісних атрибутів, яка вказує на спроможність програмного забезпечення перетворювати вхідні дані у вихідні результати за умов, що залежать від періоду часу (зношення й старіння життя програмного забезпечення не враховується).

Нефункціональні вимоги *(non-functional requirement)* – вимоги, які характеризують організаційні, виконавчі, операційні аспекти роботи програмної системи або аспекти середовища реалізації.

Об'єкти інтерфейсу *(object interface)* – об'єкти сценарію, функцією яких є трансформація повідомлень зовнішніх об'єктів у внутрішні стани сценарію.

Об'єкти керування *(object control)* – об'єкти, які здійснюють

функції перетворення об'єктів інтерфейсу в об'єкти сутності; часто відображають алгоритми оброблення даних у системі.

Об'єкти-сутності *(object substance)* — довгоживучі об'єкти, які відповідають сутностям реального світу і зберігають свій стан після виконання сценарію.

Об'єктно-орієнтована модель *(object-oriented model)* — подання програм системи як сукупності об'єктів, які взаємодіють між собою і мають певні властивості та поведінку.

Оперативність *(efficiency)* – атрибути зручності, які характеризують рівень реакції системи на зусилля користувача при виконанні **операцій** та її операційного контролю.

Пакет (у UML) *(package)* – загальний механізм організації елементів (об'єктів, класів) у групи, починаючи від усієї системи в цілому (стереотип "система") і до підсистем різних рівнів деталізації.

Переносимість *(portability)* – група властивостей, яка забезпечує пристосовність до переносу з одного середовища функціонування в інші, а також зусилля для перенесення програмного забезпечення до нового середовища або мережі.

План тестування *(plan testing)* – опис стратегії, ресурсів і графіка тестування окремих компонент та системи в цілому.

Повторно використовувана компонента *(reuse component)* – фрагмент знань про минулий досвід розроблення систем програмування, який можна адаптувати для створення нових систем і поданий так, що його можуть використовувати не лише його розробники.

Повторне використання *(reuse)* – використання для нових розробок будь-яких порцій формалізованих знань, отриманих під час реалізації завершених розробок програмних систем.

Помилка *(mistake)* – недоліки в операторах програми або в технологічному процесі її розроблення, що призводять до неправильної інтепретації вхідної інформації, а отже, і до неправильного рішення.

Прикладна система *(application system)* – кінцевий продукт програмної інженерії, призначений для виконання конкретних сценаріїв кінцевого користувача.

Програмна інженерія *(program engineering)* – система методів, здатна до масового відтворення засобів та дисципліни планування, розроблення, експлуатації та супроводження програмного забезпечення.

Процес експлуатації *(process running)* – дії з обслуговування системи під час її використання – консультації користувачів, вивчення їхніх побажань тощо.

Процес постачання *(process providing)* – дії під час передавання розробленого продукту покупцю.

Процес придбання *(process gaining)* – дії, що ініціюють життєвий

цикл системи і визначають організацію-покупця автоматизованої системи, програмної системи чи сервісу.

Процес розроблення *(process development)* – дії організації-розробника програмного продукту, які включають інженерію вимог до системи, проектування, кодування та тестування.

Проектування (*designing)* – перетворення вимог у послідовність проектних рішень щодо системи.

Проектування архітектурне *(architecture designing)* – визначення головних структурних особливостей системи, яку будують.

Проектування концептуальне *(conceptual designing)* – уточнення розуміння й узгодження деталей вимог.

Проектування технічне *(technical designing)* – відображення вимог середовища функціонування та розроблення системи, визначення усіх конструкцій як композицій компонент.

Раціональність *(rationality)* – група властивостей, яка характеризує ступінь відповідності використовуваних ресурсів середовища функціонування рівню надійності виконання при заданих умовах застосування програмного забезпечення.

Реактивні домени *(reagent domain)* – домени, зміна стану яких відбувається у зв'язку з відповіддю на певну зовнішню подію.

Рівень якості *(quality level)* – відносна характеристика якості, яка базується на порівнянні фактичних значень показників якості оцінюваного програмного забезпечення з базовими значеннями кращих аналогів або зі встановленими вимогами на розроблення програмного забезпечення.

Система компонент *(system component)* – сукупність взаємоузгоджених компонент, придатна для повторного використання.

Систематичне повторне використання *(system reuse)* – систематична і цілеспрямована діяльність із створення і використання ПВК.

Система якості *(system quality)* – набір організаційних структур, методик, процесів та ресурсів для здійснення керуванням якістю.

Специфікація *(specification)* – подання правил, стандартів, критеріїв якості, обмежень на користування об'єктів.

Спіральна модель *(model spiral)* – модель процесів життєвого циклу розробки, яка дозволяє повертатися до будь-якого попереднього процесу життєвого циклу з метою перероблення елементів зробленого продукту.

Стабільність *(stability)* – атрибути супроводжуваності, які вказують на ризик модифікації системи.

Стан (домену, системи, об'єкта тощо) *(state)* – фіксація певних властивостей на певний момент або інтервал часу.

Статичні методи тестування *(static testing mode)* – аналіз та

розгляд специфікацій компонентів без виконання їх.

Стійкість (*stableness)* – атрибут надійності, який вказує на спроможність компоненти виконувати функції в аномальних умовах (збої апаратури, помилки в даних та інтерфейсах, порушення в діях оператора тощо).

Сценарій *(script)* – один з можливих шляхів використання системи.

Тест *(test)* – програма, призначена для перевірки роботоздатності іншої програми і виявлення в ній помилкових ситуацій.

Тестові дані *(data test)* – дані, які готуються певними генераторами на основі документів програм або користувачем за специфікаціями вимог і які використовуються для перевірки роботи програмної системи.

Тестовність *(testability)* – атрибут програмного забезпечення, який вказує на ступінь проведення валідації та верифікації для виявлення помилок у програмах, невідповідностей їх вимогам та виконаним модифікаціям.

Тестування *(testing)* – спосіб семантичного налагодження (перевірки) програми, що полягає в опрацюванні програмою послідовності контрольних наборів тестів, для котрих відомий результат.

Успадкування *(inheritance)* – конкретизація в підкласі окремих властивостей та поведінки, визначених у суперкласі.

Фасад (в UML) *(facade)* – механізм доступу до тих властивостей системи компонент, які потрібні для ревикористання, або погляд на систему компонент того, хто її використовує.

Функціональні вимоги (*functional requirement)* – вимоги, які визначають складові мети і функції системи.

Функціональність (*functionality)* – це сукупність властивостей, які визначають спроможність програмного забезпечення виконувати перелік функцій у заданому середовищі, котрі відповідають вимогам.

Характеристики якості *(quality characteristics)* – функціональність (functionality), надійність (reliability), зручність (usability), ефективність (efficiency), супроводжуваність (maintainability), переносимість (portability) тощо.

Цілісність (*integrity)* – спроможність системи зберігати стабільність у роботі і не мати ризику.

UML *(Unified Modeling Language)* – мова для специфікації, візуалізації, конструювання та документування артефактів програмних систем, а також для моделювання бізнесу.

Якість програмного забезпечення *(software quality)* – сукупність властивостей, які визначають спроможність програмного забезпечення задовольнити замовника, котрий сформулював ці властивості як вимоги до розробки.

Навчальне видання

Бевз Олександр Миколайович
Папінов Володимир Миколайович
Скидан Юрій Анатолійович

ПРОЕКТУВАННЯ ПРОГРАМНИХ ЗАСОБІВ СИСТЕМ УПРАВЛІННЯ
Частина 1
Основи об`єктно-орієнтованого проектування

Навчальний посібник

Редактор О. Скалоцька

Оригінал-макет підготовлено О. Бевзом

Підписано до друку .
Формат 29,7×42¼ . Папір офсетний.
Гарнітура Times New Roman.
Друк різографічний. Ум. друк. арк. .
Наклад прим. Зам. №

Вінницький національний технічний університет,
Навчально-методичний відділ ВНТУ.
21021, м. Вінниця, Хмельницьке шосе, 95,
ВНТУ, к. 2201.
Тел. (0432) 59-87-36.
Свідоцтво суб`єкта видавничої справи
Серія ДК № 3516 від 01.07.2009 р.

Віддруковано у Вінницькому національному технічному університеті
в комп`ютерному інформаційно-видавничому центрі.
21021, м. Вінниця, Хмельницьке шосе, 95,
ВНТУ, ГНК, к.114.
Тел. (0432) 59-81-59.
Свідоцтво суб`єкта видавничої справи
Серія ДК № 3516 від 01.07.2009 р.