Co-funded by the
Tempus Programme
of the European Union

DesIRE

Г.В. ТАБУНЩИК, Т.І. КАПЛІЄНКО, О.А ПЕТРОВА

ПРОЕКТУВАННЯ ТА МОДЕЛЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ СУЧАСНИХ ІНФОРМАЦІЙНИХ СИСТЕМ

НАВЧАЛЬНИЙ ПОСІБНИК

Запоріжжя
Дике Поле
ЗНТУ
2016

УДК 004.41 (075.8)
ББК 32.972-02 я73
Т-12

Рекомендовано як навчальний посібник для студентів напрямків «Програмна інженерія» та «Комп'ютерні науки» на засіданні Вченої Ради Запорізького національного технічного університету від 01.07.2016 р., протокол №13.

Рецензенти:

С. І. Гоменюк *– доктор технічних наук, професор, декан математичного факультету Запорізького національного університету*
В. О.Філатов *– доктор технічних наук, професор, завідувач кафедри штучного інтелекту Харківського національного університету радіоелектроніки*
О.Ф. Тарасов *– доктор технічних наук, професор, завідувач кафедри комп'ютерних інформаційних технологій Донбаської державної машинобудівної академії*

Табунщик Г. В.
Т-12 Проектування та моделювання програмного забезпечення сучасних інформаційних систем / Г. В. Табунщик, Т.І. Каплієнко, О.А. Петрова – Запоріжжя : Дике Поле, 2016. – 250 с.
ISBN 978-966-2752-07-0

Навчальний посібник містить підходи та засоби аналізу, проектування та моделювання програмного забезпечення сучасних інформаційних систем, зокрема, вбудованих систем.

Видання призначене для студентів комп'ютерних спеціальностей вищих навчальних закладів, а також може бути використаним аспірантами, науковими та педагогічними працівниками, фахівцями-практиками.

Навчальний посібник видано за підтримки проекту Tempus 544091-TEMPUS-1-2013-1-BE-TEMPUS-JPCR Розробка курсів з вбудованих систем з використанням інноваційних віртуальних підходів для інтеграції науки, освіти та промисловості в Україні, Грузії, Вірменії.

Проект фінансується при підтримці Європейської комісії. Зміст матеріалу відображає думку авторів та Європейська комісія не несе відповідальності за використання інформації що міститься в навчальному посібнику.

УДК 004.41 (075.8)
ББК 32.972-02 я73

ISBN 978-966-2752-07-0

© Запорізький національний технічний університет (ЗНТУ), 2016

ЗМІСТ

PREFACE 5
ВСТУП 6
ЧАСТИНА 1. ОСНОВНІ ПОНЯТТЯ РОЗРОБЛЕННЯ ІНФОРМАЦІЙНИХ СИСТЕМ 8
1 ЖИТТЄВИЙ ЦИКЛ ТА РОЗРОБЛЕННЯ ІНФОРМАЦІЙНИХ СИСТЕМ . . 8
1.1 Етапи життєвого циклу програмного забезпечення 8
1.2 Підходи до розроблення програмного забезпечення 12
1.3 Об’єктно-орієнтована методологія та мова UML 14
1.3.1 Об’єктна модель 14
1.3.2 Історія об’єктно-орієнтованого моделювання та мови UML . . 20
1.4 Agile розробка інформаційних систем 23
1.4.1 Модель керування програмними проектами SCRUM 24
1.4.2 Модель керування програмними проектами Kanban 30
1.5 Практичні завдання 35
1.6 Контрольні запитання 35
1.7 Література до розділу 36
2 ПОВТОРНЕ ВИКОРИСТАННЯ КОДУ. ВІЛЬНЕ ТА ВІДКРИТЕ ПРОГРАМНИЕ ЗАБЕЗПЕЧЕННЯ 38
2.1 Історія вільного програмного забезпечення 39
2.2 Вільне програмне забезпечення 43
2.3 Відкрите програмне забепечення 44
2.4 Практичні завдання 49
2.5 Контрольні запитання 50
2.6 Література до розділу 50
ЧАСТИНА 2. МЕТОДОЛОГІЧНІ ОСНОВИ АНАЛІЗУ, ПРОЕКТУВАННЯ ТА МОДЕЛЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ІНФОРМАЦІЙНИХ СИСТЕМ 51
3 КОНЦЕПТУАЛІЗАЦІЯ СИСТЕМИ 51
3.1 Розроблення концепції системи 51
3.2 Встановлення вимог 53
3.2.1 Виявлення вимог 54
3.2.2 Узгодження вимог 58
3.2.3 Рівні вимог 60
3.2.4 Керування вимогами 62
3.2.5 Бізнес-модель вимог 64
3.2.5 Документ опису вимог 65
3.3 Моделювання бізнес-процесів 67
3.3.1 Моделювання 70
3.3.2 Об’єктний аналіз 72
3.3.3 Класифікація бізнес-процесів 73
3.3.4 Етапи аналізу помилок процесу 75

3.3.5 Аналіз ризиків процесу 76
3.3.6 Складові моделі об’єкта 76
3.3.7 Складний оператор 78
3.4 Практичні завдання 79
3.5 Контрольні запитання 80
3.6 Література до розділу 81
4 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 82
4.1 Модель класів предметної області 82
4.2 Модель станів предметної області 97
4.4 Практичні завдання 101
4.5 Контрольні запитання 103
4.6 Література до розділу 104
5 МОДЕЛЮВАННЯ ПОВЕДІНКИ СИСТЕМИ 105
5.1 Моделювання взаємодії 106
5.2 Практичні завдання 117
5.3 Контрольні запитання 118
5.4 Література до розділу 119
6 МОДЕЛЬ ПРОЕКТУВАННЯ 120
6.1 Модель класів додатку 120
6.2 Модель станів програмних додатків 122
6.3 Шаблони розподілення обов’язків 128
6.3.1 Шаблон Information Expert 128
6.3.2 Шаблон Creator 133
6.3.3 Шаблон Low Coupling 135
6.3.4 Шаблон High cohesion 139
6.3.5 Шаблон Controller 142
6.4 Практичні завдання 147
6.5 Контрольні запитання 149
6.6 Література до розділу 150
7 ПРОЕКТУВАННЯ СИСТЕМИ 151
7.1 Основи проектування систем 151
7.2 Поширені архітектурні стилі 170
7.3 Проектування системи на основі компонентів 175
7.4 Практичні завдання 180
7.5 Контрольні запитання 181
7.6 Література до розділу 182
АЛФАВІТНО-ПРЕДМЕТНИЙ ПОКАЖЧИК 184
Додаток А. Уніфікована мова моделювання 187
Додаток Б Опис плати Raspberri Pi 194
Додаток В Опис STM32F4DISCOVERY 208
Додаток Г Опис системної плати Digilent Basys2 212
Додаток Д Опис VHDL 232
АВТОРИ 245

PREFACE

It is clear that all higher educational institutes have the clear ambition to train students to become excellent and achieving members of their society with an international mind.

Most common form of computer in use today is by far the embedded microcontroller. This controller, combined with embedded software, is referred to as an embedded system. These systems are built into a product for control, monitoring and communication without human intervention. There are some 30 embedded microprocessors per person in developed countries with an average of 250 million lines of code. These controllers are often used in more critical domains of human life, such as medicine, automotive and aerospace applications. A skilled workforce is necessary to design and develop these qualitative systems. The growing market in embedded systems presents all higher educational institutes with a big opportunity to invest educational and research time and effort in this field of expertise.

The cooperation between different partners extends beyond the original project scope, in new projects, in new ideas and new opportunities.

It is very nice that knowledge achieved within the project will be sharing for the students. We hope that it will give them uptodate information and will help them to find their future working places.

Dirk Van Merode,
DesIRE project coordinator

ВСТУП

Стримкий розвиток сучасних інформаційних технологій передбачає наявність у молодих фахівців із інженерії програмного забезпечення та комп'ютерних наук знань та навичок з проектування, моделювання та аналізу програмного забезпечення цих систем.

Ця галузь характеризується постійним розвитком та вдосконаленням багатьох методологій, що містять сукупність моделей, методів та програмного забезпечення. А спрямованість сучасних розробок на підтримку інтернету речей потребує більш глубоких знаннь в галузі програмного забезпечення вбудованих систем. Вимоги ринку та особливості проектів з вбудованими системами обумовлюють використання відкритого програмного забезпечення, при цьому важливо розуміти як це програмне забезпечення можливо використовувати відповідно до ліцензій під якими воно розповсюджуються.

У цьому навчальному посібнику автори постарались найбільш органічно викласти матеріал, щоб підвищити рівень його засвоєння студентами напрямів «Програмна інженерія» та «Комп'ютерні науки та інформаційні технології». Викладений матеріал був апробований під час читання курсів «Проектування та моделювання програмного забезпечення вбудованих систем», «Аналіз вимог до програмного забезпечення» що вивчається студентами напряму «Програмна інженерія», та «Проектування інформаційних систем», що вивчається студентами напряму підготовки «Комп'ютерні науки та інформаційні технології».

Метою книги є надання систематичного огляду до процесів розроблення складних інформаційних систем з використанням сучасних підходів, мов програмування та мови моделювання UML.

Структурно підручник містить дві основні частини. Перша частина присвячена основним поняттям сучасного розроблення інформаційних систем. Наведені існуючі ліцензії відкритого програмного забезпечення. Особливості ітератив-

ного підходу до розроблення. Друга частина повністю містить методологічні основи аналізу проектування та моделювання інформаційних систем.

Видання орієнтоване на студентів комп'ютерних спеціальностей вищих навчальних закладів, а також може використовуватися аспірантами, науковими та педагогічними працівниками, практичними фахівцями.

Матеріал рукопису відповідає змісту програми дисципліни «Проектування та моделювання програмного забезпечення вбудованих систем», і побудований на основі матеріалів лекцій, що викладається в Запорізькому національному технічному університеті. Зокрема, авторами використано матеріали, отримані при виконанні проекту Tempus 544091-TEMPUS-1-2013-1-BE-TEMPUS-JPCR [DesIRE] «Розробка курсів з вбудованих систем з використанням інноваційних віртуальних підходів для інтеграції науки, освіти та промисловості в Україні, Грузії, Вірменії».

Терміни, визначення яких наводяться в тексті виділено ***жирним курсивом***. Кожний розділ закінчується контрольними запитаннями та практичними завданнями, що можуть бути використані як для самоперевірки, так і при підготовці лабораторних, практичних та контрольних завдань.

Для підвищення наочності матеріалуу в тексті використовуються наступні позначення:

– визначення

– практичні завдання

– приклади, контрольні запитання

– рекомендована література

ЧАСТИНА 1. ОСНОВНІ ПОНЯТТЯ РОЗРОБЛЕННЯ ІНФОРМАЦІЙНИХ СИСТЕМ

1 ЖИТТЄВИЙ ЦИКЛ ТА РОЗРОБЛЕННЯ ІНФОРМАЦІЙНИХ СИСТЕМ

1.1 Етапи життєвого циклу програмного забезпечення

Розроблення програмного забезпечення підпорядковане певному життєвому циклу. ***Життєвий цикл*** – це впорядкований набір видів діяльності, здійснюваний та керований у рамках кожного проекту з розробки програмного забезпечення (ПЗ).

На узагальненому рівні життєвий цикл (ЖЦ) може містити лише три етапи:

– аналіз;

– проектування;

– реалізація.

Етап аналізу концентрується на системних вимогах. Вимоги визначаються та специфікуються. Здійснюється розроблення та інтеграція функціональних моделей та моделей даних для системи. Крім того, фіксуються нефункціональні вимоги й інші системні обмеження.

Етап проектування поділяється на два основні підетапи: архітектурне та деталізоване проектування. Зокрема, проводиться уточнення конструкції програми для архітектури клієнт/сервер, що інтегрує об’єкти інтерфейсу користувача та бази даних. Ставляться та фіксуються питання проектування, що впливають на зрозумілість, пристосованість до супроводу та масштабованість системи.

Етап реалізації містить створення коду клієнтських програмних додатків і серверів баз даних.

Коротко кажучи, аналіз вказує на те, що робити; проектування – на те, як за допомогою наявної технології зробити це,

реалізація втілює задумане на попередніх етапах.

Спираючись на вищесказане, на деталізованому рівні ЖЦ можна розділити на наступні етапи:

– концептуалізація системи;
– специфікація вимог;
– проектування архітектури системи;
– деталізоване проектування;
– реалізація;
– інтеграція;
– супровід.

Концептуалізація системи. Розроблення програмного забезпечення починається з бізнес-аналітиків або користувачів, які придумують додаток і формулюють первинні вимоги до нього. Результатом цього етапу є документ, що містить викладення вимог. Це більшою мірою текстовий документ з деякими неформальними діаграмами та таблицями. Цей документ, як правило, не має формальних моделей, винятком можуть бути певні прості та широко відомі нотації, що можуть бути легко сприйняті замовниками.

Специфікація вимог. На даному етапі *аналітик* ретельно досліджує та безжалісно переформульовує вимоги, конструюючи моделі, виходячи з концепцій системи. Він повинен тісно працювати із замовником, щоб досягти розуміння завдання, тому що визначення, отримані на попередньому етапі, рідко виявляються повними або коректними. ***Аналітична модель*** – це стисла й точна абстракція того, що саме повинна робити система. Аналітична модель не повинна містити ніяких рішень щодо реалізації. Наприклад, клас *Window* в аналітичній моделі системи керування вікнами для робочої станції повинен бути описаний у термінах видимих атрибутів і операцій.

Аналітична модель складається з двох частин: моделі предметної області (*domain model*) – опису об’єктів реального світу, що відображає система, та моделі програмного додатка (*application model*) – опису видимих користувачеві частин самого додатка. Наприклад, для додатка біржового маклера об’єктами предметної області можуть бути акції, облігації,

торги й комісія. Модель предметної області, у свою чергу, складається з моделі класів та моделі взаємодії. Об'єкти моделі додатка можуть керувати здійсненням торгів і відображати результати. Гарна модель повинна бути доступною для розуміння та критики з боку експертів, які не є програмістами.

Проектування архітектури системи. Команда розробників продумує стратегію вирішення завдання на вищому рівні, визначаючи архітектуру системи. На цьому етапі визначаються концепції, що послужать основою для прийняття рішень на наступних етапах проектування. Проектувальник системи повинен вибрати параметри системи, відповідно до яких буде проводитися оптимізація, запропонувати стратегічний підхід до завдання, провести попередній розподіл ресурсів. Наприклад, проектувальник може вирішити, що будь-які зміни зображення на екрані робочої станції повинні бути швидкими та плавними, навіть при переміщенні та закритті вікон. На підставі цього рішення він може вибрати відповідний протокол обміну та стратегію буферизації пам'яті.

Проектування класів. Проектувальник класів уточнює аналітичну модель відповідно до стратегії проектування системи. Він пророблює об'єкти предметної області й об'єкти моделі додатка, використовуючи однакові об'єктно-орієнтовані концепції та позначення, незважаючи на те, що ці об'єкти лежать у різних концептуальних площинах. Мета проектування класів полягає в тому, щоб визначити, які структури даних й алгоритми потрібні для реалізації кожного класу. Наприклад, проектувальник класів повинен вибрати структури даних і алгоритми для всіх операцій класу Window.

Реалізація. Відповідальні за реалізацію займаються перекладом класів і відносин, що утворилися на попередньому етапі, на конкретну мову програмування, втіленням їх у базі даних або в апаратному забезпеченні. Ніяких ускладнень на цьому етапі бути не повинно, тому що всі відповідальні рішення вже були прийняті на попередніх етапах. У процесі реалізації необхідно використовувати технології розроблення програмного забезпечення, щоб відповідність коду проекту була очевидною, а система залишалася гнучкою та розширю-

ваною. У нашім прикладі група реалізації повинна написати код класу Window будь-якою мовою програмування, використовуючи виклики функцій або методи графічної підсистеми робочої станції.

Об'єктно-орієнтовані концепції діють протягом усього життєвого циклу програмного забезпечення, на етапах аналізу, проектування й реалізації. Ті самі класи будуть переходити від одного етапу до іншого без будь-яких змін у нотації, хоча на останніх етапах вони істотно обростуть деталями.

Етап інтеграції. Інтеграція модулів ретельно планується спочатку ЖЦ ПЗ. Програмні компоненти для окремої реалізації повинні бути ідентифіковані на ранніх стадіях аналізу системи. Порядок реалізації повинен дозволяти якомога плавнішу збільшувану інтеграцію.

Головна трудність полягає в існуванні взаємних зворотних залежностях між модулями. Хороший проект системи відрізняється мінімальною зв'язаністю (*low coupling*) модулів. Все одно, час від часу виникають взаємозалежні модулі, що не можуть функціонувати ізольовано.

Об'єктно-орієнтовані системи повинні бути спроектовані під інтеграцію. Кожний модуль повинен бути як найбільш незалежним. Залежності між модулями необхідно ідентифікувати та мінімізувати на етапах аналізу та проектування. Якщо система спроектована неякісно, етап інтеграції призведе до хаосу та поставить під загрозу весь проект.

Етап супроводження. Цей етап наступає після успішного постачання замовникові готової програмної системи.

Супроводження складається з трьох стадій:

– підтримання експлуатації;

– адаптивне супроводження;

– супроводження для поліпшення.

Підтримання експлуатації зв'язана з рутинними завданнями супроводу, що необхідні для підтримки системи в стані готовості до використання користувачами та експлуатаційним персоналом. *Адаптивний супровід* зв'язаний з відстеженням та аналізом роботи системи, налагодженням її функціональних можливостей відносно змін зовнішнього середовища та

адаптацією системи для досягнення заданої продуктивності та пропускної спроможності. Під *супроводом для поліпшення* розуміють перепроектування та модифікацію системи для задоволення нових або суттєво змінених вимог.

Коли супровід проекту стає недоцільним, його слід згорнути.

Концепції індивідуальності, класифікації, поліморфізму та спадкування діють протягом усього процесу розробки.

Усі ці етапи можна використати як для водоспадного процесу, де ці етапи повинні виконуватися в зазначеному вище порядку, так і для ітераційного підходу, в якому кожна складова частина розробляється вдекілька етапів.

Деякі класи не входять до аналітичної моделі. Вони з'являються пізніше, на етапах проектування або реалізації. Наприклад, структури даних, подібні до дерев, хеш-таблиць і зв'язних списків, рідко з'являються в реальному світі та зазвичай невидимі для користувачів. Проектувальники додають їх у систему для того, щоб забезпечити підтримку обраних алгоритмів. Об'єкти структур даних існують усередині комп'ютера та не є безпосередньо спостережуваними.

Тестування як окремий етап ЖЦ не розглядається, тому що воно дуже важливе, але повинне бути частиною системи контролю якості, що застосовується протягом усього ЖЦ. Розробники повинні порівнювати аналітичні моделі з реальністю, перевіряти проектувальні моделі на наявність помилок різних видів, а не тільки тестувати коректність реалізації. Виділення всіх операцій з контролю якості в окремий етап коштує дорожче та є менш ефективним.

1.2 Підходи до розроблення програмного забезпечення

Революція в програмному забезпеченні викликала низку значних змін у способах роботи програмних продуктів. Зокрема, значно збільшилися інтерактивні можливості програм.

Розрізняють структурний і об'єктно-орієнтований підходи до розроблення ПЗ.

Структурний підхід до розроблення систем набув широкого поширення в 80х роках. Цей підхід заснований на двох

методах: *діаграмах потоків даних* (*data flow diagrams*) для моделювання процесів і *діаграмах сутність-зв'язок* (*entity relationship diagrams*) для моделювання даних.

Структурний підхід є функціонально-орієнтованим і розглядає *DFD*-діаграми як рушійну силу розроблення ПЗ. У зв'язку з поширенням баз даних, значення *DFD*-діаграм у структурній розробці знизилося, і підхід став більш орієнтованим на дані, і, відповідно, акцент при розроблені ПЗ змістився на *ERD*-діаграми.

Поєднання *DFD* і *ERD* діаграм дає відносно повні моделі аналізу, що фіксують всі функції та дані системи на необхідному рівні абстракції незалежно від можливостей апаратного та програмного забезпечення. Потім модель аналізу перетвориться впроектну модель, що зазвичай виражається в поняттях реляційних баз даних. Після цього слідує етап реалізації.

Недоліками цього підходу є:

– він є послідовним і трансформаційним, а не ітеративним підходом з нарощуванням можливостей;

– він направлений на постачання негнучких рішень;

– він передбачає розроблення «з чистого аркуша» і не підтримує повторного використання компонент.

Об'єктно-орієнтований підхід набув поширення в 1990-х роках. Асоціація виробників ПЗ *Object Management Group* (*OMG*) затвердила як стандартний засіб моделювання цього підходу мову *UML*.

Порівняно зі структурним підходом об'єктно-орієнтований підхід більшою мірою орієнтований на дані – він розвивається довкола моделей класів. Зростаюче значення використання в мові *UML* претендентів сприяє незначному зсуву акцентів від даних до функцій.

До найбільш важливих категорій додатків, для яких потрібна об'єктна технологія, відносяться обчислювальна обробка для робочих груп і системи мультимедіа.

Об'єктний підхід до розроблення систем слідує ітеративному процесу з нарощуванням можливостей. Єдина модель конкретизується на етапах аналізу, проектування та реалізації.

Об’єктно-орієнтований підхід призводить до виникнення низки труднощів:

– оскільки етап аналізу проводиться на ще вищому рівні абстракції і якщо серверна частина рішення з реалізації передбачає використання реляційної бази даних, семантичний розрив між концепцією і її реалізацією може бути значним;

– керування проектом складно здійснювати;

– висока складність рішення, що у свою чергу позначається на таких характеристиках ПЗ, як пристосованість до супроводу і масштабованість.

1.3 Об’єктно-орієнтована методологія та мова ***UML***

Сутність об’єктно-орієнтованого підходу (ООП) полягає, перш за все, у двох моментах.

З одного боку, він надає розробнику інструмент, що дозволяє описати завдання й істотну частину реалізації проекту в термінах, що характеризують предметну область, а не комп’ютерну модель.

З іншого боку, об’єктно-орієнтоване програмування надає розробникам гнучкий потужний універсальний інструмент, не пов’язаний з якимось певним класом задач.

1.3.1 Об’єктна модель

Як концептуальна основа об’єктно-орієнтованого аналізу та проектування виступає об’єктна модель.

Об’єктну модель складають чотири головні компоненти:

– абстрагування;

– інкапсуляція;

– модульність;

– ієрархія.

Абстрагування дозволяє виділити суттєві характеристики деякого об’єкта, що відрізняють його від усіх інших видів об’єктів. Абстракція чітко визначає концептуальні кордони об’єкта з точки зору спостерігача.

Інкапсуляція – це процес відокремлення одне від одного елементів об'єкта, що визначають його будову.

Ієрархія – це впорядкування абстракцій, засіб класифікації об'єктів та систематизації зв'язків між об'єктами.

Модульність – це подання системи у вигляді сукупності відокремлених сегментів, зв'язок між якими забезпечується за допомогою зв'язків між класами, що визначаються в цих сегментах.

Розглянемо всі ці компоненти більш докладно.

Абстрагування. Абстрагування є одним із основних методів, що використовуються для вирішення складних завдань. Абстракція виділяє істотні характеристики певного об'єкта, що відрізняють його від інших видів об'єктів і, таким чином, чітко визначає його концептуальні кордони з точки зору спостерігача.

Наприклад. Автомобіль має корпус і колеса, причому корпуси кожного автомобіля дуже відрізняються один від одного.

Під абстрактними моделями, що використовуються в ООП, розуміють моделі, які характеризуються найбільш загальними властивостями актуальними для практичних випадків.

ООП характеризується наявністю двох основних видів абстракцій:

– тип даних об'єктів природи (***клас***) – розширення вихідних типів мови програмування, що визначається програмістом;

– екземпляр класу (***об'єкт***) – змінна класу.

Об'єкт має стан, поведінку та ідентичність.

Стан об'єкта характеризується набором його властивостей (атрибутів) і поточними значеннями кожної з цих властивостей (автомобіль заведений, фари включено).

Стан об'єкта – результат його поведінки, тобто виконання впевній послідовності характерних для нього дій.

Ідентичність – це така властивість об'єкта, що відрізняє його від усіх інших об'єктів того ж типу.

Наприклад. Розглянемо абстракцію «Лампочка». Для лампочки визначено, як мінімум, два стани: ввімкнуто та вимкнуто. Для того щоб з'ясувати її поточний

стан, нам потрібно подивитись на лампочку. Процес вмикання та вимикання лампочки характеризує поведінку об'єкта. Всі лампочки мають приблизно однакову поведінку, хоча їх реалізації можуть відрізнятися.

На рівні абстрактної моделі кожній операції відповідає метод класу (рис. 1.1).

Лампочка
+Вімкнути() : void +Вимкнути() : void +ПеревіритиСтан() : bool

Рисунок 1.1 – Абстракція електричної лампочки

Клас визначає загальні властивості об'єктів (тобто тип). Коли покупець приходить до магазину та просить дати йому електричну лампочку, то висловлюючись термінами ООП, він оперує ім'ям класу (вказує тип предмета, що його цікавить). Продавець видає конкретну лампочку – об'єкт класу.

Отже, абстрагування відповідає за зовнішню поведінку об'єкта – так поводяться всі об'єкти класу.

Інкапсуляція. Інкапсуляція тісно пов'язана з абстракцією. Якщо абстрагування спрямоване на спостереження поведінки об'єкта, то інкапсуляція займається внутрішнім пристроєм і тому визначає чіткі межі між різними абстракціями.

Інша важлива властивість інкапсуляції – можливість приховування реалізації. Інкапсуляція завжди припускає можливість обмеження доступу до даних класу. З одного боку, це дозволяє спростити інтерфейс класу, показавши найбільш істотні для зовнішнього користувача дані та методи. З іншого боку, приховування реалізації забезпечує можливість внесення змін уреалізацію класу без зміни інших класів.

Наприклад. Для класу «Лампочка» рівень реалізації зв'язаний зі способом уявлення стану лампочки. У нашому прикладі ми ввели логічну змінну. Знак «мінус» у атрибута *стан* показує, що ззовні доступ до нього закритий (рис. 2.2). Це значить, що змінити значення змінної *стан* можуть тільки методи класу.

Лампочка
-стан : bool = false
+Вімкнути() : void +Вимкнути() : void +ПеревіритиСтан() : bool

Рисунок 1.2 – Клас «Лампочка»

Отже робимо висновок, що інкапсуляція ізолює інтерфейс від реалізації, вона зв’язана з керуванням доступом до даних класу. Але важливо мати на увазі, що інкапсуляція не забезпечує повної закритості класу.

Види структурних ієрархій. Відомо, що вивчення і організація складних технічних систем істотно спрощується, якщо слідувати принципам функціонально-ієрархічної декомпозиції.

Ієрархічна декомпозиція й ієрархічна організація ПЗ утворюють один із основних систематичних методів подолання складності останнього.

Структурна ієрархія «частина-ціле». Структурна ієрархія побудована за принципом «*частина-ціле*» називається ***агрегацією***.

Агрегація є найпростішою реалізацією однієї з базових ідей, що лежать в основі ООП: створений клас в ідеалі повинен являти собою фрагмент корисного коду, доступного для повторного використання.

Варіант агрегації, при якому відношення володіння чітко виражене, а час життя частин і цілого збігаються, називається композицією. Проста агрегація на відміну від композитної агрегації припускає, що одна і та ж «частина» може одночасно належати різним «цілим».

Відношення композиції утворюють, наприклад, автомобіль як ціле, а двигун, трансмісія, гальмова система як його частини (рис. 2.3).

Механізм агрегування надає програмістові гнучкий засіб організації ієрархії класів.

Структурні ієрархії «is-a» і «is-like-a». Ідея спадкування випливає з бажання мати зручний засіб, що б дозволив

створювати клас, дані і поведінка якого в деякій частині схожі з існуючим класом.

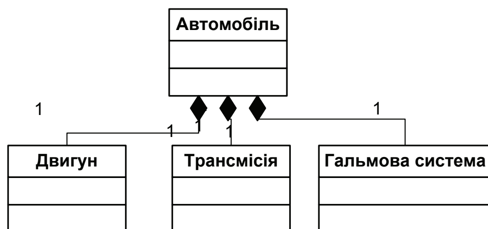

Рисунок 1.3 – Приклад композиції

Спадкування є найпоширенішою в об'єктно-орієнтованих програмах формою відносини узагальнення. Узагальнення означає, що об'єкти класу нащадка можуть використовуватися скрізь, де допустимі об'єкти батьківського класу. Зворотне в загальному випадку неправильно.

Наприклад, коли нам потрібно створити клас «Кольорова лампочка», що, крім стану, має ще колір, не варто створювати новий клас. Треба створити клас-нащадок від класу «Лампочка», якій містить специфічний атрибут колір та специфічні методи (рис. 2.4).

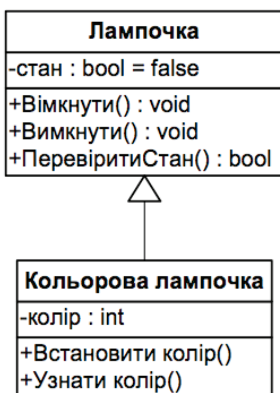

Рисунок 1.4 – Приклад спадкування

Об'єкт класу «Кольорова лампочка» успадкує всі дані та методи батьківського класу (рис.2.5).

О : Кольорова лампочка
стан : bool = false колір : int

Рисунок 1.5 – Об´єкт класу «Кольорова лампочка»

Створюючи базовий тип (базовий клас), програміст висловлює найбільш загальні ідеї щодо об’єктів, з яких конструюється програма. У похідних класах програміст уточнює відмінність уреалізації конструкцій базового класу.

При конструюванні нового класу програміст висловлює найбільш загальні ідеї щодо об’єктів, з яких від базового створюється новий тип. Цей тип включає всі дані і методи базового типу. Відповідність типу похідного класу типу базового класу є одним із фундаментальних принципів реалізації концепції успадкування в рамках об’єктної моделі.

Поведінку об’єктів, для яких викликаються видозмінені методи, інтерфейс, до яких заявлено в базовому класі, називають поліморфною, а механізм, що забезпечує таку поведінку, називають поліморфізмом.

Ієрархія і модулі. Під модульністю будемо розуміти уявлення системи у вигляді сукупності відокремлених сегментів, зв’язок між якими забезпечується за допомогою зв’язків між класами, що визначаються в цих сегментах. При цьому виділяють два рівні модульності: фізичний та логічний.

Фізичний рівень має справу зі структурою файлів і каталогів, уяких зберігаються окремі модулі, що становлять проект.

Механізм забезпечення реалізації модульності програмних проектів на основі логічної реалізації програмних об’єктів зазвичай пов’язаний з використанням простору імен.

Простір імен – це іменована або неіменована відкрита область дії імен.

Побудова модуля завжди припускає процес логічного групування, оголошень та визначень у зв’язку з розв’язуваними завданнями йспільністю використовуваних даних.

Простору імен надають механізм відображення логічного групування програмних об’єктів засобами мови програмування.

1.3.2 Історія об'єктно-орієнтованого моделювання та мови UML

Мови для об'єктно-орієнтованого моделювання почали з'являтися в період із середини 1970 року й кінця 1980-х років, коли різні розроблювачі експериментували з різними підходами до об'єктно-орієнтованого аналізу й проектування.

В основі *UML* лежить кілька об'єктно-орієнтованих методів, кожний з яких спочатку був орієнтований на підтримку окремих етапів об'єктно-орієнтованого аналізу й проектування (ООАП):

– метод Граді Буча (Grady Booch), умовна назва *Booch* (*Booch'91*, *Booch Lite, Booch'93*), вважався найбільш ефективним на етапах проектування й розроблення програмних систем;

– метод Джеймса Румбаха (*James Rumbaugh*), *Object Modeling Technique* (*OMT*, пізніше *OMT-2*), оптимально підходив для аналізу процесів оброблення даних у інформаційних системах;

– метод Айвара Джекобсона (Ivar Jacobson), *Object-Oriented Software Engineering* (*OOSE*), містив засоби подання прецедентів, що мають істотне значення на етапі аналізу вимог при проектуванні бізнесів-додатків.

Історія розвитку *UML* датується 1994 р., коли почалася інтеграція/уніфікація вищевказаних методів. Проект уніфікованого методу (*Unified Method*) версії 0.8 був опублікований ужовтні 1995 р.

Всі питання розроблення й супроводу мови *UML* сконцентровані в рамках консорціуму *OMG (Object Management Group*). Хоча *OMG* був створений з метою розроблення пропозицій щодо стандартизації об'єктних і компонентних технологій *CORBA*, мова *UML* набула статусу другого стратегічного напрямку в роботі консорціуму. Улистопаді 1997 р. *OMG* оголосив *UML* стандартною мовою об'єктно-орієнтованого моделювання та взяв на себе обов'язок з його подальшого розвитку. Група фахівців забезпечує публікацію описів наступних версій мови *UML* і запитів пропозицій *RFP* (*Request For Proposals*) по його стандартизації. Статус мови *UML* ви-

значений як відкритий для всіх пропозицій щодо доопрацювання йудосконалення. У 2003р. як результат розгляду набору запитів *RFP* 2000р. був опублікований опис мови *UML* 2.0, що містить інфраструктуру *UML*, мова обмежень об'єктів (*Object Constraint Language – OCL*), суперструктуру *UML* і формат обміну даними.

Основними ініціативами консорціуму *OMG* у рамках роботи над проектом *UML* є:

1. Моделювання систем реального часу.
2. Визначення моделі виконання – точної специфікації поведінки моделей, що підтримуються *UML*.
3. Оброблення даних підприємства – визначені так звані профілі, що описують способи створення великих розподілених паралельних систем підприємства.
4. Визначення процесу розроблення програмного забезпечення – специфіковані структури визначення процесів розробки програмного забезпечення.
5. Стандарт зберігання даних.
6. Зіставлення технології *CORBA* та мови *UML*.
7. Формат *XMI (Metadata Interchange*) для обміну моделями *UML* у текстовому форматі [9].

Існує консорціум партнерів *UML* (D*igital Equipment Corp., HP, Intellicorp, IBM, ICON Computing, Microsoft, Oracle, Rational Software* іінші), що забезпечують уточнення нотації, удосконалення й доповнення мови, а також супровід розробки інструментальних засобів підтримки. Особливе місце займає компанія *Rational Software Corporation*, що реалізувала *Rational Rose 98* – один з перших інструментальних CASE-засобів, уякому була підтримана мова *UML*. Необхідно відзначити увагу компанії Microsoft до технології, що базується на мові *UML*, на основі якої створена інформаційна модель (*UML Information Model*), призначена для створення стандартного інтерфейсу між засобами розроблення додатків і засобами візуального моделювання. На даний момент *Rational Software*

Corporation офіційно входить до складу *IBM* (http://www-306.ibm.com/software/rational/).

Мова *UML* призначена для вирішення таких завдань:

1. Надати в розпорядження користувачів готову до використання виразну потужну мову візуального моделювання, що дозволяє розробляти осмислені моделі й обмінюватися ними.
2. Передбачити внутрішні механізми розширення й спеціалізації базових концепцій мови.
3. Забезпечити максимальну незалежність проекту створення програмного забезпечення від конкретних мов програмування й процесів розробки.
4. Забезпечити формальну основу для однозначної інтерпретації мови.
5. Стимулювати розширення ринку об'єктно-орієнтованих інструментальних засобів створення програмного забезпечення.
6. Інтегрувати кращий практичний досвід використання мови йреалізації програмних засобів його підтримки.

Принципи використання *UML* специфіковані в *Rational Unified Process* (*RUP*) – розвинутій методиці створення програмного забезпечення, оформленої у вигляді бази знань, фізично розміщеної на web (http://www-306.ibm.com/software/awdtools/rup/support/) і оснащеною пошуковою системою.

Основним документом з *UML* є [http://www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF], де описана метамодель *UML* і дуже мало уваги приділяється семантиці мовних конструкцій. Опис поточної версії мови *UML* і приклади розроблення програмних систем з використанням *CASE*-засобу *Rational Rose* можна знайти на web-вузлі *OMG* (www.omg.org); модифікації мови *UML* і його новітні версії – на вузлі www.celigent.com/uml.

UML створювалася як мова моделювання загального призначення для застосування в таких «дискретних» галузях, як програмне забезпечення, апаратні засоби й цифрова логіка. Структури *UML* дозволяють фіксувати різноманітні рішення з відображення:

1) функціональності системи;
2) динамічної та статичної структури системи;
3) організації елементів системи;
4) реалізації системи.

Популярності набуває використання *UML* при проектуванні баз даних. Завдяки відкритості (наявності в мові механізмів розширення) він надає потужний інструментарій для вирішення завдань інших галузей, наприклад, бізнесу-моделювання.

1.4 Agile розробка інформаційних систем

Ітеративна розробка – це технічний підхід до створення програмних систем, покладений в основу опису об'єктно-орієнтованого аналізу та проектування.

У рамках цього підходу розроблення виконується у вигляді декількох короткострокових міні-проектів фіксованої тривалості, що називаються *ітераціями*. Кожна ітерація містить свої власні етапи аналізу вимог, проектування, реалізації та завершується тестуванням, інтеграцією та створенням робочої системи.

Перші ідеї ітеративного процесу називалися «проектування по спіралі й еволюційною розробкою».

Переваги ітеративного розроблення:

– своєчасне усвідомлення можливих технічних ризиків, осмислення вимог, завдань проекту та зручності використання системи;

– швидкий та помітний прогрес;

– ранній зворотний зв'язок, можливість обліку побажань користувачів і адаптації системи. У результаті система більш задовольняє реальні вимоги керівників і споживачів;

– керована складність;

– отриманий при реалізації кожної ітерації досвід можна методично використовувати для поліпшення самого процесу розроблення.

Agile – гнучка методологія розроблення, – це концептуальний каркас, в рамках якого виконуються проекти з

розроблення програмного забезпечення. Основна ідея всіх гнучких моделей полягає в тому, що процес, який застосовується у розробці ПЗ, повинен бути адаптивним. Вони декларують своєю вищою цінністю орієнтованість на людей та їх взаємодію, а не на процеси і засоби.

1.4.1 Модель керування програмними проектами SCRUM

Методологія *Scrum* була вперше озвучена в роботі Хіротака Такеучи й Ікуджіро Нонаки, опублікованій в *Harvard Business Review*. Джеф Сазерленд використовував цю роботу при створенні методології для корпорації *Easel* у 1993 році, яку за аналогією і назвав *Scrum*, а Кен Швабер формалізував процес для використання в індустрії розроблення програмного забезпечення.

Мета цієї методології – виявити й усунути відхилення від бажаного результату на більш ранніх етапах розроблення програмного продукту.

Методологія *Scrum* визначає:

1. Правила, за якими повинен плануватися і управлятися список вимог до продукту, з метою досягнення максимальної прибутковості від реалізованої функціональності;
2. Правила планування ітерацій для забезпечення максимальної зацікавленості команди в отриманні результату;
3. Основні правила взаємодії учасників команди для максимально швидкої реакції на непередбачені робочі ситуації;
4. Правила аналізу і коригування процесу розроблення для вдосконалення взаємодії всередині команди.

Кожну ітерацію можна описати так: «Плануємо – Фіксуємо – Реалізуємо – Аналізуємо». За рахунок фіксування вимог на протязі однієї ітерації та зміни довжини ітерації можна керувати балансом між гнучкістю та плановим керуванням розроблення.

Scrum фокусується на постійному визначенні пріоритетних завдань, ґрунтуючись на бізнес цілях, що збільшує корисність і прибутковість проекту на його ранніх стадіях. Зважаючи

на те, що при ініціації проекту його прибутковість визначити майже неможливо, *Scrum* пропонує концентруватися на якості розроблення і до кінця кожної ітерації мати проміжний продукт, який можна використовувати.

Наприклад, результатом ітерації може бути каркас сайту, який можна показати на презентації.

Методологія *Scrum* орієнтована на те, щоб оперативно пристосовуватися до змін у вимогах, що дозволяє команді швидко адаптувати продукт до потреб замовника.

Девіз *Scrum* – «аналізуй та адаптуй»: аналізуйте те, що отримали, адаптуйте те, що вже розроблено до реальних потреб, а потім аналізуйте знову. Чим менше формалізму, тим більш гнучко та ефективно можна працювати, – це основний принцип даної методології. Але це не означає, що формальних процесів не повинно бути зовсім, їх має бути достатньо для організації ефективної взаємодії і керування проектом. Формальна частина *Scrum* складається з трьох ролей, трьох практик і трьох основних документів.

В *Scrum*-проектах відділяють наступні ролі:

1. **Власник продукту** (*Product Owner*) – людина, що формулює вимоги програмістам. Зазвичай власник продукту є представником або довіреною особою замовника, а для компаній, що випускають коробкові продукти, він являє собою ринок, на якому реалізується продукт. Власник продукту повинен скласти бізнес план, який показує очікувану дохідність і план розвитку до вимог, відсортованими за коефіцієнтом окупності інвестицій. Виходячи з наявної інформації, власник продукту готує перелік вимог, відсортований за значимістю;

2. *Scrum*-**майстер** (*Scrum Master*) – забезпечує максимальну працездатність і продуктивність команди, чітку взаємодію між усіма учасниками проекту, своєчасне вирішення всіх проблем, що гальмують або зупиняють роботу будь-якого члена команди, відгороджує ко проходження процесу всіх учасників проекту. Ця людина має бути одним з членів команди розроблення і брати участь у проекті як розробник. Він відповідає за своєчасне вирішення поточних проблем.

3. *Scrum*-**команда** (*Scrum Team*) – група, що складається з п'яти-дев'яти самостійних, ініціативних програмістів. Перше завдання цієї команди – поставити реально досяжну, прогнозовану, цікаву і значущу мету для ітерації. Друге завдання – зробити все для того, щоб ця мета була досягнута у відведені терміни і з заявленою якістю. Мета ітерації вважається досягнутою тільки в тому випадку, якщо всі поставлені завдання реалізовані, весь код написаний за певними проектом «стандартам кодування» (*coding guidelines*), програма протестована повністю, а всі знайдені дефекти усунені. Програмісти цієї команди повинні вміти оцінювати та планувати свою роботу, працювати в команді, постійно аналізувати та поліпшувати якість взаємодії та роботи. В обов'язки всіх членів *Scrum*-команди входить участь у виборі мети ітерації і визначення результату роботи. Вони повинні робити все можливе і неможливе для досягнення мети ітерації в рамках, визначених проектом, ефективно взаємодіяти з усіма учасниками команди, самостійно організовувати свою роботу, надавати власнику робочий продукт в кінці кожного циклу.

Scrum містить наступні документи: **журнал продукту** (*Product Backlog*), **журнал спринту** (*Sprint Backlog*) та **графік спринту** (*Burndown Chart*).

На початку проекту власник продукту готує **журнал продукту** (табл. 1.1) – перелік вимог, відсортований за значимістю, а *Scrum*-команда доповнює цей журнал оцінками вартості реалізації вимог. Перелік повинен містити функціональні та технічні вимоги, необхідні для реалізації продукту. Найбільш пріоритетні з них повинні бути досить детально прописані, щоб їх можна було оцінити і протестувати. Про своєчасну деталізацію вимог має дбати власник продукту і надавати необхідний обсяг у потрібний час. У цьому сенсі програмісти є замовниками вимог для власника продукту. Надалі інші вимоги повинні поступово уточнюватися та деталізуватися до такого ж рівня. Головне, щоб у команди завжди був достатній обсяг підготовлених до реалізації вимог.

Графік спринту дозволяє також власнику продукту спостерігати за ходом ітерації – якщо загальний обсяг робіт не

зменшується щодня, значить, щось йде не так. Під час сесії планування команда знаходить і оцінює завдання, які треба виконати для успішного завершення ітерації. Сума оцінок всіх завдань в журналі спринту є загальним обсягом роботи, який треба виконати за ітерацію. Піс ля завершення кожного завдання *Scrum*-майстер перераховує обсяг роботи, що залишилася, і зазначає це на графіку спринту. Тільки в тому випадку, якщо по закінченні ітерації у журналі спринту не залишилося незавершених завдань, ітерація вважається успішною. Графік спринту використовується як допоміжний інструмент, що дозволяє коригувати роботу для завершення ітерації вчасно, з працюючим кодом і необхідною якістю.

Таблиця 1.1. Приклад журналу продукту

Номер вимоги	**Опис вимоги**	**Цінність бізнесу (ум. од.)**	**Пріоритет**	**Високорівнева оцінка (часи)**
FE1	Покупець може зареєструватися на сайті	10.000	1	20
FE2	Покупець може ввести свої персональні дані	12.000	6	36
FE3	Покупець може побачити список доступних виробів	16.000	10	30
FE4	Покупець може купити виріб	100.000	20	48
FE5	Покупець може робити пошук виробів	80.000	30	32
FE6	Покупець може підписатись на новини	30.000	40	66

К *Scrum*-**практикам** відносяться: спринт (*Sprint*), скрам (*Daily Scrum Meeting*) та демонстраційне засідання (*Sprint Review Meeting*).

1. Підготовка до першої ітерації, званої **спринт** (*Sprint*), починається після того, як власник продукту розробив **журнал продукту**.

При плануванні ітерації відбувається детальне розроблення сесій планування спринту (*Sprint Planning Meeting*), яке починається з того, що власник продукту, *Scrum*-команда і *Scrum*-майстер перевіряють план розвитку продукту, план релізу та перелік вимог, *Scrum*-команда перевіряє оцінки вимог, переконується, що вони досить точні, щоб почати працювати, вирішує, який обсяг роботи вона може успішно виконати за спринт, ґрунтуючись на розмірі команди, доступному часі та продуктивності. *Scrum*-команда повинна вибирати перші за пріоритетом вимоги з журналу продукту. Після того як *Scrum*-команда зобов'язується реалізувати обрані вимоги, *Scrum*-майстер починає планування спринту. *Scrum*-команда розбиває вибрані вимоги на завдання, необхідні для його реалізації. Ця активність в ідеалі не повинна займати більше чотирьох годин, і її результатом є **журнал спринту**. Необхідно, щоб всі учасники команди взяли на себе зобов'язання з реалізації обраної мети.

2. Після закінчення планування починається ітерація. Кожен день *Scrum*-майстер проводить «**скрам**» (*Daily Scrum Meeting*) – п'ятнадцятихвилинної нарада, мета якої – досягти розуміння того, що сталося з часу попередньої наради, скоригувати робочий план до реалій сьогодення і позначити шляхи вирішення існуючих проблем. Кожен учасник *Scrum*-команди відповідає на три питання: що я зробив з часу попереднього скрама, що мене гальмує або зупиняє в роботі, що я буду робити до наступного скрама? У цій нараді може брати участь будь-яка зацікавлена особа, але тільки учасники *Scrum*-команди мають право приймати рішення.

Правило обґрунтовано тим, що вони давали зобов'язання реалізувати мету ітерації, і тільки це дає впевненість у тому, що вона буде досягнута. На них лежить відповідальність за їх власні слова, і, якщо хтось з боку втручається і приймає рішення за них, тим самим він знімає відповідальність за результат з учасників команди.

В кінці кожного спринту проводиться демонстраційне засідання (Sprint Review Meeting) тривалістю не більше чотирьох годин. Спочатку Scrum-команда демонструє власнику продукту зроблену протягом спринту роботу, а той у свою чергу веде цю частину наради і може запросити до участі всіх зацікавлених осіб. Власник продукту визначає, які вимоги з журналу спринту були виконані, і обговорює з командою і замовниками, як краще розставити пріоритети в журналі продукту для наступної ітерації. У другій частині мітингу проводиться аналіз минулого спринту, який веде Scrum-майстер. Scrum-команда шукає використані в останньому спринті позитивні і негативні способи спільної роботи, аналізує їх, робить висновки і приймає важливі для подальшої роботи рішення. Scrum-команда також визначає програми, які можуть працювати краще, і шукає шляхи для збільшення ефективності подальшої роботи. Потім цикл замикається, і починається планування наступного спринту.

Час між ітераціями – це час прийняття основоположних рішень, що впливають на хід всього проекту. Під час спринту ніякі зміни ззовні не можуть бути зроблені. Після того як команда дала зобов'язання реалізувати журнал спринту, він фіксується, і зміни в ньому можуть бути зроблені тільки з таких причин:

- Scrum-команда протягом ітерації отримала кращу уяву про вимоги і потребує додаткових завдань для успішного завершення ітерації;
- знайдені дефекти, які треба обов'язково виправити для успішного завершення ітерації;
- Scrum-майстер і Scrum-команда можуть вирішити, що невеликі зміни, які не впливають на загальний обсяг робіт, можуть бути реалізовані в зв'язку з виниклою у власника продукту необхідністю.

Виходячи з того що журнал спринту не може бути змінений ззовні під час ітерації, потрібно вибирати його довжину, ґрунтуючись на стабільності вимог. Якщо вимоги стабільні, змінюються або доповнюються рідко, то можна вибрати шеститижневий цикл. У цьому випадку заощаджується час на пе-

ремикання команди з активного розроблення на планування та демонстраційні мітинги. Якщо вимоги часто змінюються і доповнюються, потрібно відштовхуватися від двотижневого циклу, в будь-якому випадку довжина ітерації – це величина експериментальна.

Недоліки *Scrum*:

1. Складно домогтися активної участі від кожного розробника і злагодженої колективної роботи в команді.

2. Складно залучити постачальника вимог до активної участі в проекті, зацікавити його динамікою розвитку продукту, дати можливість бути активним вболівальником і спонсором команди.

Проте, незважаючи на це, використання методології *Scrum* в проектах дозволяє:

– використовувати весь технічний багаж, накопичений компанією, тому що головний напрям методології *Scrum* направлений на керування проектами і не задає ніяких технічних практик;

– з високою якістю та в рамках бюджету реалізувати великий обсяг функціональності та специфікації, які були відсутні на момент початку проекту;

– забезпечити максимальну бізнес-цінність виробленого продукту, за рахунок того, що практично всі реалізовані функції активно використовуються відвідувачами;

– досягнути високу супровідність коду (можливість внесення змін з мінімальними трудовитратами) – вартість змін, внесених до продукту, практично еквівалентна вартості розроблення аналогічних функцій продукту на початку проекту, що рідко буває у RUP чи MSF моделях виробництва, для яких характерний експонентний ріст вартості змін по мірі виконання проекту.

1.4.2 Модель керування програмними проектами Kanban

З початку 70-х років у Японії, а потім і в інших країнах набула великого поширення система «*Канбан*», що є механіз-

мом організації безперервного гнучкого виробничого потоку і функціонує практично без страхових запасів. Традиційна концепція організації виробництва, як відомо, спрямована на запобігання простоям та організацію безперервного потоку з обов'язковим створенням страхового запасу. Японська концепція ґрунтується на практично повній відмові від страхових запасів. Більше того, менеджери навмисне дають робітникам змогу повністю випробувати на собі наслідки простоїв. В результаті весь персонал постійно зайнятий виявленням причин збоїв у виробництві та пошуком шляхів підвищення надійності та запасу міцності системи управління. Після виявлення та усунення причин простоїв керівники ще більше скорочують страховий запас, породжуючи додаткові зусилля з покращення організації виробництва з боку всього персоналу. В умовах системи «*Канбан*», на відміну від традиційного підходу, виробник не має завершеного плану й графіка виробництва, а жорстко пов'язаний конкретним замовленням споживача. Він не взагалі оптимізує свою роботу, а в межах замовлення. Конкретного графіка роботи на декаду, місяць він не має. Кожен зайнятий у технологічному ланцюгу робітник знає, що він вироблятиме продукцію тільки тоді, коли карта «*Канбан*» з його продукції відкріплена від контейнера на складі, тобто коли продукція фактично надійшла на наступну стадію обробки.

Конкретний графік послідовності праці одержують лінії кінцевого складання, вони розкручують клубок інформації у зворотному напрямі. Іншими словами, графіки виробництва не переглядаються, а тільки формуються рухом карток «*Канбан*». Виробництво постійно перебуває в стані надбудови, відбувається його системне коригування під зміни ринкової кон'юнктури.

Методика розроблення програмного забезпечення ***Канбан*** була введена у практичне використання Девідом Андерсеном у 2007 році. Багато з цих практик та підходів використовувалися різними *Agile* командами, перш ніж були описані як єдине ціле.

Нововведення полягали в тому, що було введено завдання «в процесі». Це робилося і раніше іншими *Agile*-командами,

але в Канбан існує всім відоме обмеження на кількість робочих завдань, які можуть виконуватися в один час. Ця межа зазвичай досить низька – ліміт приблизно дорівнює числу розробників в команді або трохи менший.

Основні положення Канбан наступні:

1. Візуалізація потоку робіт:

– розбиття роботи на частини, кожна частина виписується на картку і прикріплюється до стіни;

– розбиття усіх завдань на стовпчики для розділення по стадіям розробки.

2. Обмеження незавершеної роботи (НЗР) (*work*-in-progress) визначається можлива кількість незавершених пунктів на кожній стадії робочого процесу.

3. Вимірювання часу виконання завдання (*lead* time) (середньої тривалості часу для завершення одного пункту, іноді звану «оперативним часом» (*cycle* time)), оптимізація процесу, щоб звести час виконання завдання до мінімуму і зробити його настільки прогнозованим, наскільки це можливо.

Канбан – це не конкретний процес, а система цінностей. Його можна описати однією простою фразою – «Зменшення виконується в певний момент роботи (*work in progress*)».

Різниця між *Канбан* і Scrum:

– немає таймбоксов (ні на завдання, ні на спринти);

– завдання більше і їх менше;

– оцінки термінів на задачу опціональні або взагалі їх немає;

– «швидкість роботи команди» відсутня і враховується тільки середній час на повну реалізацію завдання.

Команда для роботи використовує дошку.

Наприклад, вона може виглядати класично, як на рис. 1.6. Де дошка розділена на 3 складових.

Наприклад, вона може мати і більше стовпців (стовпці зліва направо):

Цілі проекту. Сюди можна помістити високорівневі цілі проекту.

Черга завдань. Тут зберігаються завдання, які готові до того, щоб почати їх виконувати. Завжди для виконання

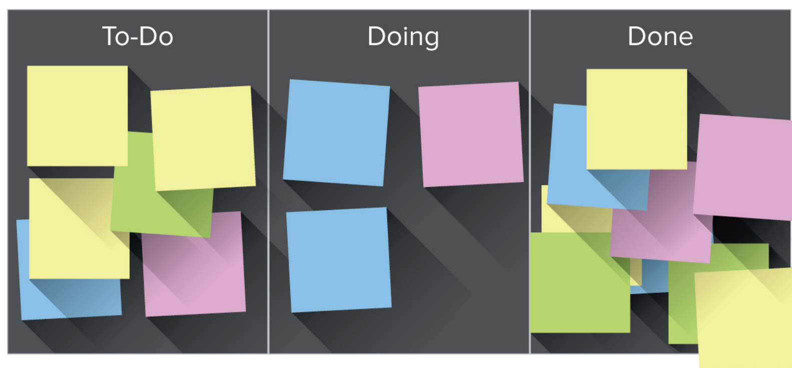

Рис. 1.6. – Приклад Канбан дошки

береться зверху, сама пріоритетна задача і її картка переміщується в наступний стовпець.

Опрацювання дизайну. Цей та інші стовпці до «Закінчено» можуть змінюватися, тому що саме команда вирішує, які кроки проходить завдання до стану «Закінчено».

Наприклад, в цьому стовпці можуть перебувати завдання, для яких дизайн коду або інтерфейсу ще не ясний і обговорюється. Коли обговорення закінчені, завдання пересувається в наступний стовпець.

Розробка. Тут завдання перебуває до тих пір, поки розробка не завершена. Після завершення вона пересувається в наступний стовпець. Або, якщо архітектура не вірна або не точна – завдання можна повернути в попередній стовпець.

Тестування. У цьому стовпці завдання знаходиться, поки воно тестується. Якщо знайдені помилки – повертається в Розробку. Якщо ні – пересувається далі.

Деплоймент. У всіх проектів різне поняття деплоймент, наприклад, викласти нову версію продукту на сервер або помістити код в репозиторій.

Закінчено. Сюди стікер потрапляє лише тоді, коли всі роботи по завданню закінчені повністю.

В будь-якій роботі трапляються термінові завдання. Для таких можна виділити спеціальне місце (наприклад «Прискорити»). В Прискорити можна помістити один рядок до

завдання і команда повинна почати її виконувати негайно і завершити якомога швидше. У такій черзі може бути тільки одна така задача, якщо з'являється ще одна – вона повинна бути додана в «Черга завдань».

Під кожним стовпцем вказуються цифри, які позначають кількість завдань, які можуть бути одночасно в цих стовпцях. Цифри підбираються експериментально, але вважається, що вони повинні залежати від числа розробників в команді.

Наприклад, якщо є 8 програмістів в команді, то в рядок «Розробка» можна помістити цифру 4. Це означає, що одночасно програмісти будуть робити не більше 4-х завдань, а значить у них буде багато причин для спілкування та обміну досвідом. Якщо поставити туди цифру 2, то 8 програмістів, що займаються двома завданнями, можуть простоювати або втрачати занадто багато часу на обговореннях. Якщо поставити 8, то кожен буде займатися своїм завданням і деякі завдання будуть затримуватися на дошці надовго, а головне завдання Канбан – це зменшення часу проходження завдання від початку до стадії готовності.

Підсумовуючи, переваги *Канбан* полягають у тому, що:

1. Легко визначаються проблемні області.

2. Зростає продуктивність роботи за рахунок того, що члени команди техпідтримки самостійно організують свою роботу, використовуючи канбан-дошку, а менеджер лише спрямовує зусилля на приорітезацію великих проектів і вирішення виникаючих проблем.

Недоліки *Канбан*:

1. Зі зменшенням НЗР з'являються обмеження – для менш пріоритетних проектів не вистачає ресурсів, що призводить до скорочення кількості проектів на команду.

2. Канбан накладає менше обмежень, ніж *Scrum*, тобто керівництво отримує більше параметрів для налаштування. Це може бути як недоліком, так і перевагою, залежно від ситуації.

3. Канбан – це ще більш «гнучка» методологія, ніж *Scrum* і *XP*, тому вона не підійде командам та проектам, не готовим до гнучкої роботи.

1.5 Практичні завдання

Завдання 1. Напишіть реферат на одну із тем:

– призначення та використання діаграм потоків даних;

– проектування та аналіз систем з використанням IDEF технологій;

– моделювання бізнес-процесів;

– методології структурного аналізу;

– функціонально-орієнтований аналіз.

Завдання 2. Зробіть порівняльну характеристику структурного та об’єктно-орієнтованого підходів; функціонально-орієнтованого та об’єктно-орієнтованого.

Завдання 3. Дослідить характеритики підходу ScrumBan. Порівняйте його з підходами Scrum та Канбан.

1.6 Контрольні запитання

1. Які етапи містить життєвий цикл програмного забезпечення?
2. На якому етапі життєвого циклу здійснюється розроблення та інтеграція функціональних моделей та моделей даних для системи?
3. Який етап містить написання коду клієнтських програм?
4. На яких методах заснований структурний підхід?
5. Які недоліки структурного підходу?
6. На що орієнтований об’єктно-орієнтований підхід?
7. Яка мова затверджена як стандарт для об’єктно-орієнтованого підходу?
8. З чого складається аналітична модель?
9. В чому полягає мета проектування?
10. Порівняйте Scrum та Канбан.

1.7 Література до розділу

1. Брауде Э. Технология разработки программного обеспечения [Текст] / Брауде Э. – СПб.: Питер, 2004. – 655с. ил.
2. Пышкин Е. В. Основные концепции и механизмы объектно-ориентированного программирования [Текст] / Пышкин Е. В. – СПб.: БХВ-Петербург, 2005. – 640 с.: ил.
3. Соммервилл И. Инженерия программного обеспечения; пер. с англ. [Текст] / Соммервилл И. – М. : Изд. дом «Вильямс», 2002. – 624 с.
4. Константайн Л. Разработка программного обеспечения; пер. с англ. [Текст] / Л. Константайн, Л. Локвуд.– СПб.: Питер, 2004. – 592 с.
5. Коплиен Дж. Мультипарадигменное проектирование для С++ [Текст] / Коплиен Дж. – СПб. : Питер, 2005. – 235 с.
6. Bahrami, A. Object Oriented Systems Developoment? Irwin McGrawHill, 1999. – 412 pp.
7. Object Management Group: [Електрон. ресурс]. – Режим доступу:http://www.omg.org/.
8. Буч Г. Язык UML. Руководство пользователя [Текст]/ Г. Буч, Д. Рамбо, А. Джекобсон. – М.: ДМК Пресс, 2001. – 257 с.
9. Грэхем И. Объектно-ориентированные методы. Принципы и практика; пер. с англ. [Текст] / ГрэхемИ. – М.: Издательский дом «Вильямс», 2004. – 880 с.
10. Рамбо Дж. UML: специальный справочник. [Текст] / Дж. Рамбо, А. Якобсон, Г. Буч. – СПб.: Питер, 2002. – 656 с.
11. Элиенс А. Принципы объектно-ориентированной разработки программ; пер. с англ. [Текст] /Элиенс А. – М.: Издательский дом «Вильямс», 2002. – 496 с.

12. Брагина, Т.И. Сравнительный анализ итеративных моделей разработки программного обеспечения [Текст] / Т.И. Брагина, Г.В. Табунщик // Радиоелектроніка. Інформатика. Управління. – 2010. – № 2. – С. 130 – 139.
13. Брагина, Т.И. Анализ подходов к управлению рисками в программных проектах с итеративным жизненным циклом [Текст] / Т.И. Брагина, Г.В. Табунщик // Радиоелектроніка. Інформатика. Управління. – 2011. – №2 – С. 120-124.
14. Брагіна, Т.І. Розробка засобів інтеграції / Т.І. Брагіна // Системи обробки інформації. Вип.8 (106). – 2012. – С. 127-130.
15. Брагина, Т.И. Оценка прогресса разработки программных проектов в JIRA / Т.И. Брагина, К.В. Задорожная // Тиждень науки – 2013: зб. тез доп. щоріч. наук.-практ. конф. викладачів, науковців, молодих учених, аспірантів, студентів ЗНТУ (Запоріжжя, 20–25 квіт. 2013 р.). – Запоріжжя: ЗНТУ, 2013. – С. 270-271.
16. Табунщик, Г.В. Інженерія якості програмного забезпечення:навчальний посібник / Г.В. Табунщик, Р.К. Кудерметов, Т.І. Брагіна. – Запоріжжя: ЗНТУ, 2013. – 176 с.

2 ПОВТОРНЕ ВИКОРИСТАННЯ КОДУ. ВІЛЬНЕ ТА ВІДКРИТЕ ПРОГРАМНИЕ ЗАБЕЗПЕЧЕННЯ

Повторне використання коду (англ. *code reuse*) – методологія проектування комп'ютерних та інших систем, яка полягає в тому, що система (комп'ютерна програма, програмний модуль) частково або повністю повинна складатися з двох частин, написаних раніше компонентів і/або частин іншої системи, і ці компоненти повинні застосовуватися більш ніж один раз (якщо не в рамках одного проекту, то хоча б різних). Повторне використання – основна методологія, яка застосовується для скорочення трудовитрат при розробці складних систем.

Існують як суто практичні, так і філософські аргументи на користь розвитку вільного програмного забезпечення.

Відкриті програми дозволяють розробникам аналізувати вихідний код, знаходити можливості для його вдосконалення і використовувати його в своїх продуктах. Користувачі по всьому світу можуть вносити свій вклад в розвиток різними способами, від програмування до написання і перекладу документації, а також звітів про помилки. Вільний обмін ідеями є рушійною силою інновацій.

Згідно маніфесту *GNU*, люди мають моральне право на доступ до удобочитаем поданням комп'ютерних програм.

Термін «*вільний*» має два значення: «незалежний» і «безкоштовний». Виголошуючи промови з імпровізованої трибуни, прихильникам вільного программного забезпечення (ПЗ) часто доводиться явно пояснювати, мають вони на увазі, що ПЗ можна використовувати і поширювати вільно, або що його можна отримати безкоштовно. При цьому будь-який студент знає, що всі продукти мають собівартість, тобто хтось витратив певні кошти на їх виробництво, навіть якщо їх можна отримати і використовувати безкоштовно.

Невільне програмне забезпечення ще називається власницьким або пропрієтарним, від «proprietary». Важливо не плутати умовно-безкоштовні та безкоштовні (freeware) програми з вільними, це різні речі.

Рисунок 2.1 – Типи ліцензій програмного забезпечення

До вільних програм відносяться:

– програми з відкритим вихідним текстом
– програми суспільним надбанням – це програми, на які не поширюється виключне авторське право
– програми з авторським лівому
– пільні програми без авторського лева
– програми з нерозважливо ліберальної ліцензією
– напіввільні програми
– невільні програми
– безкоштовні програми
– умовно-безкоштовні програми
– замовні програми
– комерційні програми

Тому мета цього розділу надати пояснення до всіх цих визначень.

2.1 Історія вільного програмного забезпечення

Обмін програмним забезпеченням існував стільки ж скільки і самі програми. На початку розвитку комп’ютерів, ви-

робники вважали, що конкурентні переваги можуть бути вилучені лише з вдосконалення апаратних засобів і відтак не ставилися до ПЗ як до бізнес активу. Хоча цей ранній період і нагадує сучасну культуру вільного програмного забезпечення, він відрізняється за двома ключовими моментами. Поперше, на той момент не існувало майже ніякої стандартизації серед апаратних засобів – це був час розквіту інновацій в комп'ютерному дизайні, але різноманітність комп'ютерних архітектур означала, що одне ПЗ було несумісне іншим. Таким чином, програми, написані для однієї машини зазвичай було не можливо використовувати на іншій. Програмісти намагалися спеціалізуватися в одній архітектурі або сімействі архітектур (так само як на сьогоднішній день програмісти більш спеціалізуються в певній мові програмування або їх сімействі, в повній впевненості, що їх знання можна буде перенести на будь-яку обчислювальну платформу, з якою їм доведеться працювати). Через те, що особисті знання людини зазвичай ставилися до одного виду комп'ютерів, накопичення знань призводило до того, що ці комп'ютери ставали більш привабливими для людини. Отже, в інтересах виробника було поширювати машинно-залежний код і знання якнайширше.

З розвитком галузі, одночасно відбулося кілька взаємопов'язаних змін. Велика різноманітність апаратних платформ, поступово поступилася місцем кільком безперечним лідерам – переможцям, які володіли найкращою технологією, найкращим маркетингом або комбінацією обох. У той же час, і не зовсім випадково, розробка так званих мов програмування «високого рівня», привела до того, що, будь-хто міг один раз написати програму, на одній мові, і потім автоматично перевести («скомпілювати») її в код, виконуваний на будь-якому типі комп'ютерів. Виробники, для котрих обладнання було єдиним видом активу, відчули майбутнє падіння прибутку. Чиста обчислювальна потужність ставала взаємозамінним товаром, а ПЗ перетворювалося в нову межу розділу. Продаж програмного забезпечення, та ставлення до нього як до невід'ємної частини продажу обладнання, було досить хорошою стратегією на майбутнє.

На практиці це призвело до того, що виробники стали набагато суворіше охороняти авторські права на свій код. Якщо користувачі просто продовжували б вільно обмінюватися між собою кодом та змінювати його, то вони могли б незалежно реалізувати якісь удосконалення, які продавалися постачальником як «додаткова цінність». Постачальники вжили заходів, які або забороняють користувачеві доступ до коду і виконуються на його машині, або наполягають на угоді про нерозголошення, що зробило обмін корисною інформацією неможливим.

У той час як світ необмеженого обміну кодом повільно згасав, в голові принаймні одного програміста викристалізувалася ідея контрреакції. Річард Столлмен працював в лабораторії Штучного Інтелекту в Массачусетському технологічному інституті в 1970-х і на початку 80-х, він відмовився від лабораторії і почав проект *GNU*, а також заснував Фонд вільного програмного забезпечення (*FSF*). Метою *GNU* була розробка абсолютно безкоштовної і відкритої комп'ютерної операційної системи і набору прикладних програм, користувачам яким ніколи б не заборонялося досліджувати код або поширювати внесені ними зміни. Одночасно з роботою над новою операційною системою, Столмен розробив авторську ліцензію, умови якої гарантували, що його код завжди буде безкоштовним.

За допомогою багатьох програмістів, серед котрих деякі поділяли ідеї Столлмена, а деякі просто хотіли побачити деяку кількість доступного безкоштовного коду, проект *GNU* почав випускати безкоштовні заміни більшості найважливіших компонентів операційної системи. У зв'язку з уже широко поширеною стандартизацією в комп'ютерному апаратному та програмному забезпеченні, стало можливим використовувати заміни від *GNU* для не-безкоштовних систем, і багато хто так і робив. Текстовий редактор від *GNU* (*Emacs*) і компілятор *Сі* (*GCC*) були особливо успішні, і отримали велику і віддану армію шанувальників не на ідеологічній основі, а просто через їх технічні характеристики. Приблизно до 1990 року, *GNU* створив велику частину безкоштовної операційної системи, за винятком ядра – частини, з якої починається завантажен-

ня комп'ютера, і яка відповідає за управління пам'яттю, диском та іншими системними ресурсами.

На жаль, для проекту *GNU* була обрана архітектура ядра, яку виявилося реалізувати важче, ніж передбачалося. Отримана затримка не дозволила Фонду вільного програмного забезпечення випустити першу версію повністю безкоштовної операційної системи. Замість них, заключна частина була вставлена на своє місце Лінусом Торвальдсом, фінським студентом факультету комп'ютерних наук, який, за допомогою добровольців з усього світу, закінчив ядро, використавши більш консервативний дизайн. Він назвав його Linux, і коли воно було зеднане з існуючими *GNU* програмами, результатом була повністю безкоштовна операційна система. У перший раз користувачі могли запустити комп'ютер і працювати без використання будь-якого пропрієтарного програмного забезпечення.

У світі вільного програмного забезпечення, відбувалося ще безліч інших подій, і деякі з них були явно ідеологічними, як і проект *GNU* Столлмена. Одним з таких подій став Програмний пакет університету Берклі (Berkeley Software Distribution) (*BSD*), послідовна переробка операційної системи Unix, яка до початку 1970-х була в деякій мірі пропрієтарним дослідницьким проектом в *AT&T*, виконуваним програмістами Каліфорнійського Університету в Берклі. Група *BSD* не робила ніяких відкритих політичних заяв про те, що програмістам потрібно об'єднатися і обмінюватися один з одним, але вони втілювали цю ідею зі смаком і ентузіазмом, координуючи сильно розподілену розробку, де Unix-утиліти командного рядка і бібліотеки коду, а іноді і саме ядро операційної системи, були переписані з нуля добровольцями. Проект *BSD* став першим прикладом не-ідеологічної розробки вільного програмного забезпечення, а так само послужив тренувальним майданчиком для багатьох розробників, які хотіли продовжувати працювати, щоб залишатися дійовими особами світу в світі вільного програмного забезпечення.

Навіть не перераховуючи кожен проект і кожну ліцензію окремо, з упевністю можна сказати, що на початку 1980-х,

існувало багато вільного програмного забезпечення, доступного під широким набором ліцензій. Різноманітність ліцензій відображала відповідна різноманітність мотивів. Дехто ж із програмістів, які обрали *GNU GPL* були набагато менше проваджені ідеологією, ніж передбачалося проектом *GNU*.

2.2 Вільне програмне забезпечення

Програма вільна, якщо у її користувачів є чотири свободи:

– свобода виконувати програму в будь-яких цілях (свобода 0).

– свобода вивчати роботу програми і модифікувати програму, щоб вона виконувала ваші обчислення, як ви побажаєте (свобода 1). Це передбачає доступ до вихідного тексту.

– свобода передавати копії, щоб допомогти своєму ближньому (свобода 2).

– свобода передавати копії своїх змінених версій іншим (свобода 3). Цим ви можете дати всьому співтовариству можливість отримувати вигоду від ваших змін. Це передбачає доступ до вихідного тексту.

Можливість виправлення помилок і покращення програм – найважливіша особливість вільного і відкритого програмного забезпечення, що неможливо для користувачів закритих приватних програм навіть при виявленні в них помилок і дефектів, кількість яких, як правило, невідома нікому.

Тільки программа, яка задовольняє всі чотири перераховані принципи може вважатися вільною програмою, тобто гарантовано відкритою, доступною для модернізації та виправлення помилок і дефектів, не має обмежень на використання та поширення. Потрібно підкреслити, що ці принципи обумовлюють тільки доступність початкового програмного коду для загального використання, критики та поліпшення, і права користувача, який отримав вихідний код програми, але ніяк не обумовлюють пов’язані з поширенням програм грошові відносини, в тому числі не припускають і безоплатності. ВПЗ цілком можна поширювати на платній основі, проте дотримуючись при цьому критерії свободи: кожному користувачеві

надається право отримати вихідні тексти програм без додаткової плати (за винятком ціни носія), змінювати їх і поширювати далі. Будь-яке програмне забезпечення, користувачам якого не надається такого права, є невільним – «пропрієтарним» (від англ. Proprietary) незалежно від будь-яких інших умов.

Відкритий доступ до вихідних текстів програм є ключовою ознакою вільного ПЗ, тому запропонований трохи пізніше Еріком Реймондом термін open source software (ПЗ з відкритим вихідним текстом) деяким представляється навіть більш вдалим для позначення даного феномена.

Відмінність між відкритим та вільноим ПЗ полягає в основному в пріоритетах. Прихильники терміна «відкритий ресурс» роблять акцент на ефективність відкритих початкових кодів як методу розробки, модернізації та супроводу програм. Прихильники терміна «вільний ресурс» вважають, що саме права на вільне поширення, модифікацію і вивчення програм є головною перевагою вільного відкритого ПЗ.

Програми з відкритим вихідним кодом, як правило, випускаються під вільною ліцензією.

Яскравим прикладом ефективності ВПЗ служить історія розвитку операційної системи *Linux*.

2.3 Відкрите програмне забепечення

У 1998, коаліція програмістів, яка згодом стала Комітетом по Програмному Забезпеченню з Відкритим Вихідним Кодом (*Open Source Initiative* (*OSI*)), створила на заміну терміну «вільний» термін *з відкритим вихідним кодом*. [9] *OSI* відчував не тільки те, що термін «вільне програмне забезпечення» міг збити з пантелику, але і що метушня навколо терміна» вільний «була лише ознакою більш глибокої проблеми: необхідна була маркетингова програма, яку можна запропонувати корпоративному світу і яка поклала б край розмовам про «моралі» та «соціальні блага»в залах засідань корпорацій.

Ліцензії відкритого програмного забезпечення (*OSS*) повинні відповідати 10 критеріям, таким чином, щоб вважатися з відкритим вихідним кодом:

1. Вільне розповсюдження: Ліцензія не повинна обмежувати будь-яку сторону від продажу або програмного забезпечення як складової частини збірки програмного забезпечення, що містить програми з кількох різних джерел. Ліцензія не повинна вимагати авторського гонорару або іншої платні за такий продаж.

2. Вихідний код: Програма повинна включати в себе вихідний код і повинна допускати поширення у вигляді вихідного коду, а також в компільованою формі. Там, де будь-яка форма продукту не поширюється з вихідним кодом, має бути добре розрекламовану засобом отримання вихідного коду не більше розумної вартості відтворення переважно, скачування через Інтернет без пред'явлення звинувачень. Вихідний код повинен бути кращою формою, в якій програміст міг би модифікувати програму. Навмисне заплутування вихідного коду не допускається. Проміжні форми, такі як висновок препроцесора або перекладача не є допустимим.

3. Похідні роботи: Ліцензія повинна допускати модифікації і похідні роботи і повинні дозволити їм поширюватися на тих же умовах, що і ліцензія оригінального програмного забезпечення.

4. Цілісність авторського вихідного коду: Ліцензія може обмежувати вихідний код від поширення в модифікованій формі тільки якщо вона дозволяє розповсюдження «файлів патчів» з вихідним кодом для цілей модифікації файлів в процесі побудови, Ліцензія повинна явно дозволяти розповсюдження програмного забезпечення, створеного на основі модифікованого вихідного коду. Ліцензія може вимагати, щоб похідні роботи мали ім'я або номер версії від оригінального програмного забезпечення.

5. Відсутність дискримінації щодо осіб або груп: Ліцензія не повинно призводити до дискримінації по відношенню до будь-якої особи або групи осіб.

6. Відсутність дискримінації щодо областей діяльності: Ліцензія не повинна обмежувати кого-небудь від використання програми в конкретній галузі діяльності. Наприклад, вона не може обмежувати програму від використання в бізнесі, або від використання в якості генетичних досліджень.

7. Поширення ліцензії: права, прикладені до програми, повинні застосовуватися до всіх, кому програма перерозподіляється без необхідності виконання додаткових ліцензій цими сторонами.

8. Ліцензія не повинна ставитися до продукту: права, прикладені до програми не повинні залежати від програми, що є частиною конкретного поширення програмного забезпечення. Якщо програма витягується з цього розподілу і використовують або розповсюджують відповідно до умов ліцензії на здійснення програми, всі сторони, яким програма перерозподіляється повинні мати ті ж права, як і ті, які надаються разом з початковим розподілом програмного забезпечення.

9. Ліцензія не повинна обмежувати інше: Ліцензія не повинна накладати обмеження на інше програмне забезпечення, яке розповсюджується разом з ліцензійним програмним забезпеченням. Наприклад, ліцензія не повинна наполягати на тому, що всі інші програми що розповсюджуються на тому ж середовищі, повинні бути з відкритим вихідним кодом.

10. Ліцензія повинна бути технологічно нейтральним: Жодне з положень ліцензії не може ґрунтуватися на будь-якої індивідуальної технології або стилі інтерфейсу

Розглянемо найбільш розповсюджені ліцензії ПЗ.

1. **Універсальна громадська ліцензія** *GNU* **(***GNU* **General Public License,** *GPL***)**.

GNU General Public License (універсальна загальнодоступна ліцензія *GNU*, або відкрита ліцензійна угода *GNU*), мабуть, найбільш популярна ліцензія на вільне програмне забезпечення, створена в рамках проекту *GNU* в 1988 році. Її також скорочено називають *GNU GPL*, або просто *GPL*, якщо з контексту зрозуміло, що мова йде саме про цю ліцензію (існують і інші ліцензії, що містять слова «general public license» в назві). Друга версія цієї ліцензії була випущена в 1991 році, третя версія, після багаторічної роботи і тривалої дискусії – в 2007 році.

Мета *GNU GPL* – надати користувачеві програми такі права, які за замовчуванням заборонені законом про авторські права, а також гарантувати, що і інші користувачі всіх похід-

них (змінених) програм отримають точно такі ж розширені права. Вносячи будь-які зміни у відкритий програмний код, розробник зобов'язується надалі надавати свої вихідні коди кожному користувачеві на першу вимогу. При цьому автори знімають з себе будь-яку відповідальність за те, як буде використовуватися їх продукт і до яких наслідків може привести його використання. Єдине, що явно забороняється, -це закриття вихідних кодів після їх модифікації.

В цьому і полягає принцип успадкування прав, або копілефт (транслітерація англійського «copyleft»), який був придуманий Річардом Столлманом.

2. **Програмна лицензія университету Берклі (*Berkeley Software Distribution*, сокращенно *BSD*)**.

Також надає право необмеженого використання в сторонніх розробках, але, на відміну від *GPL*, дозволяє в подальшому зробити продукт закритим. Тобто ліцензія *BSD* накладає менше обмежень на користувача, ніж звичайний копірайт. Тому в певному сенсі використання цієї ліцензії ближче до переміщення програми в категорію суспільного надбання.

Права на вихідний дистрибутив *BSD* офіційно належать піклувальникам університету Каліфорнії (Regents of the University of California) – керуючому органу університету Каліфорнії. Причина в тому, що *BSD* був розроблений в кампусі Берклі університету Каліфорнії. Ця вказівка початкових прав збереглась в сучасних версіях *BSD (NetBSD, FreeBSD, OpenBSD, DragonFly BSD)*.

В даний час ліцензії типу BSD є одними з найпопулярніших ліцензій для вільного програмного забезпечення і використовуються для багатьох програм (крім *BSD*-версій *UNIX*, для яких ліцензія *BSD* була спочатку створена).

3. **Вільні ліцензії *Artistic, Affero, Apache, LGPL* (*Lesser* *GPL* – полегшена версія *GPL*), MIT/X11, ZPL та ін**. в тій чи іншій мірі доповнюють (полегшуючи або підсилюючи) викладені вище вимоги вільних ліцензій.

4. ***Mozilla Public License* – *MPL*.**

Використовується в якості ліцензії для *Mozilla Suite, Mozilla Firefox, Mozilla Thunderbird* та інших програм, розроблених

в рамках проекту *Mozilla*. Вона також була адаптована іншими розробниками, в особливості Sun Microsystems, в якості ліцензії (*Common Development and Distribution License*) для *OpenSolaris*, версії Solaris з відкритими вихідними кодами.

Вихідний код, скопійований або змінений під ліцензією *MPL*, повинен бути ліцензований за правилами *MPL*. На відміну від більш суворих вільних ліцензій, код під ліцензією *MPL* може бути об'єднаний в одній програмі з закритими файлами.

Різні частини дистрибутивів програмного забезпечення можуть підпадати під умови різних ліцензій, ситуацію також ускладнює необхідність ретельної перевірки пакетів, що входять у дистрибутив, на можливість використання їх в рамках інших ліцензій. Існують дистрибутиви, в основі яких лежить як вільні, так і комерційні компоненти, не призначені для вільного розповсюдження. Такі комерційні дистрибутиви *GNU/Linux* часто не можуть вільно копіюватись та розповсюджуватись. Втім, розбиратися у всіх цих тонкощах – завдання вже самих авторів дистрибутивів, а не користувачів. Кожна вільна операційна система супроводжується спеціальною ліцензійною угодою, в якій і роз'яснюються права та обов'язки як самих розробників, так і користувачів. Розуміючи складність процесу, багато розробників відразу ділять свої дистрибутиви на комерційні (тобто ті, в які входять «закриті» компоненти) та некомерційні (повністю на вільному ПЗ) – наприклад, *Red Hat Fedora*.

Тексти перерахованих вище ліцензій знаходяться у вільному доступі в мережі Internet і можуть бути вільно скопійовані і роздруковані користувачем. У тих країнах, де це не суперечить місцевому законодавству, вільні ліцензії мають чинності без підпису і печатки ліцензіара – в тому числі в електронному вигляді. Ліцензія обов'язково присутня у вигляді посилання на текст в коді відкритого програмного продукту. При купівлі так званої коробкової версії дистрибутива ліцензійна угода буде у неї вкладено у вигляді спеціально надрукованого документа. Угоду у вигляді текстового файлу, завжди можна знайти і викачати при завантаженні образів програмних продуктів у форматі ISO з Internet.

Всі перераховані ліцензії мають силу лише англійською мовою (рис. 2.2).

2.4 Практичні завдання

1. Зробить порівняльний аналіз різних ліцензій до програмного забезпечення.
2. Розробить прототип нової ліцензії що дозволить ліцензувати вільне українське програмне забезпечення. Розробіть дорожню карту взаємодії з Creative Commons.
3. Проведіть аналіз існуючого програмного забезпечення для протоколу *BLE 4.0*.

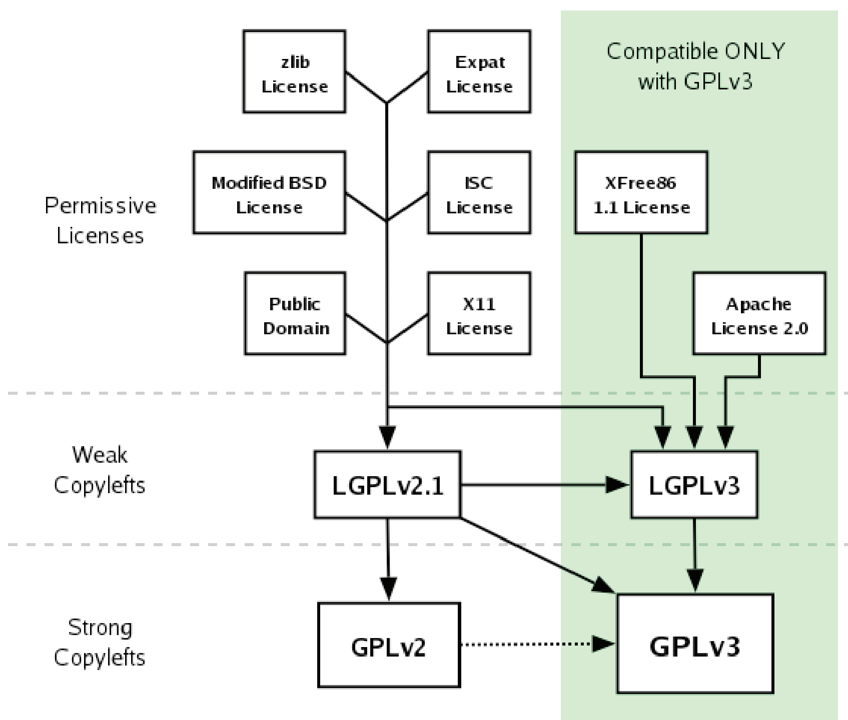

Рисунок 2.2 – Зв'язок ліцензій програмного забезпечення

4. Проведіть аналіз вільного програмного забезпечення на C++ та Python для оброблення відеопотоку. Проведіть аналіз можливості їх сумісного використання.

2.5 Контрольні запитання

1. Кто був засновником підходу до вільного розповсюдження програмного забезпечення?

2.6 Література до розділу

1. Повторное использование кода посредством кодоориентированной разработки и моделирования. Режим доступа [Электронный ресурс]: http://www.ibm.com/developerworks/ru/library/r-reuse-code-centric-development-modeling/
2. Sojer M. Reusing Open Source Code Value Creation and Value Appropriation Perspectives on Knowledge Reuse, 2011. Gabler Verlag | Springer Fachmedien Wiesbaden GmbH – 306 p.

Частина 2. МЕТОДОЛОГІЧНІ ОСНОВИ АНАЛІЗУ, ПРОЕКТУВАННЯ ТА МОДЕЛЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ІНФОРМАЦІЙНИХ СИСТЕМ

3 КОНЦЕПТУАЛІЗАЦІЯ СИСТЕМИ

3.1 Розроблення концепції системи

Концептуалізація системи – це зародження додатка.

У більшості випадків ідеї, на яких ґрунтуються нові системи, є продовженням уже існуючих ідей.

Існує кілька способів пошуку концепцій нових систем:

– нова функціональність – можна додати функціональність в існуючу систему;

– модернізація – зняття обмежень або універсалізація роботи системи;

– спрощення – надання звичайним людям можливості займатися тим, чим раніше займалися тільки фахівці;

– автоматизація ручних процесів;

– інтеграція – об'єднання функціональності різних систем;

– аналогії – пошук аналогій в інших предметних областях і дослідження їх на наявність корисних ідей;

– глобалізація – вивчення культури й ділової практики інших країн та впровадження їх досвіду.

Перед тим як вкладати кошти та час у розробку, потрібно оцінити можливість створення системи, витрати й ризики, пов'язані з її розробленням, потребу в системі та відношення виграшу до витрат.

Робота над системою починається з розроблення концепції. Гарна концепція повинна містити наступну інформацію:

– для кого призначений додаток – визначаються зацікавлені особи, які є 2-х типів: спонсори та користувачі.

– яке завдання буде вирішувати додаток – визначаються функції, які буде мати система;

– де буде використовуватися система – визначається де буде використовуватися система, чи буде вона незалежною, чи буде доповнювати вже існуючі системи;

– коли буде потрібна система. Для нового додатка важливо два аспекти, зв’язаних з часом. По-перше, це час за який система може бути розроблена з урахуванням обмежень щодо вартості та ресурсів. По-друге, це час, за який система може бути розроблена, щоб задовольнити вимоги бізнесу. Треба переконатися, що оцінка часу, отримана з урахуванням технологічних можливостей, відповідає потребі бізнесу;

– чому потрібна система – підготовлюють економічне обґрунтування системи;

– як буде працювати система – розглядаються можливості використання різних архітектур.

Наприклад, потрібно розробити систему «Банкомат».

Концепція: необхідно розробити програмне забезпечення, що дозволить клієнтам одержувати доступ до банківської комп’ютерної системи та виконувати операції без участі банківських службовців.

1. Для кого призначений додаток?

Банкомати виробляються кількома компаніями. Тому тільки виробник автомата може компенсувати витрати на створення програмного забезпечення банкомата.

2. Яке завдання буде вирішувати додаток?

ПЗ банкомата повинне працювати як на банк, так і на клієнта. З погляду банку, програмне забезпечення підвищує рівень автоматизації та скорочує обсяг ручної роботи.

З погляду клієнта, банкомати повинні бути поширені повсюдно та цілодобово доступні.

ПЗ повинне бути простим у використанні та зручним. Система повинна бути надійною та захищеною, оскільки вона працює з грошима.

3. Де буде використовуватися система?

Актуальний засіб для всіх фінансових установ.

4. Чому потрібна система?

Новий продукт дозволить виробнику бути більш конкурентоспроможним.

5. Як буде працювати система?

Планується реалізувати трирівневу архітектуру, відокремивши інтерфейс користувача від програмної логіки, а логіку від бази даних.

Після того як ідея кристалізується, формулюються вимоги до системи, що будуть описувати мету та загальний підхід до створення системи.

3.2 Встановлення вимог

Мета встановлення вимог полягає в тому, щоб дати розгорнуте визначення функціональних, а також нефункціональних вимог, які учасники проекту очікують затвердити в реалізованій і розгортуваній системі.

Вимоги визначають послуги, очікувані від системи (формулювання сервісів) та обмеження, яким система повинна підлягати (формулювання обмежень). Ці формулювання сервісів можна об'єднати в кілька груп: одна із груп описує межі системи, інша – необхідні бізнес-функції (функціональні вимоги), а третя – необхідні структури даних (вимоги до даних).

Вимоги необхідно одержати від зацікавлених осіб. Цей вид діяльності називається *виявленням вимог* і здійснюється аналітиком бізнес-процесів (або системним аналітиком).

До зацікавлених осіб в проекті відносяться:

- замовники;
- користувачі;
- аналітики вимог;
- розробники;
- тестери;
- технічні письменники;
- менеджер проекту;
- співробітники юридичного відділу;
- представники промислових організацій;
- співробітники відділу продажів.

Потрібно виконати ретельний аналіз зібраних вимог для виявлення в них повторів і протиріч. Це незмінно приводить

до перегляду вимог та повторного їх узгодження з зацікавленими особами.

Вимоги, задовільні з погляду зацікавлених осіб, документуються. При цьому вимогам дають визначення, класифікують, нумерують і привласнюють їм пріоритети. Структура документа, що описує вимоги, відповідає шаблону, обраному в організації для цієї мети.

Хоча документ, що фіксує вимоги, носить значною мірою описовий характер, цілком можливо включити до нього високорівневу схематичну бізнес-модель. Як правило, *бізнес-модель* складається з моделі меж системи, моделі бізнес-прецедентів і моделі бізнес-класів.

3.2.1 Виявлення вимог

Бізнес-аналітик виявляє вимоги до системи за допомогою консультацій, до участі в яких залучаються замовники, користувачі й експерти в проблемній області. У деяких випадках бізнес-аналітик має достатній досвід у проблемній області, і допомога експерта може не знадобитися. У цьому випадку бізнес-аналітик являє собою різновид експерта проблемної області.

Завдання бізнес-аналітика полягає в тому, щоб об'єднати два набори вимог у *бізнес-моделі*. Бізнес-модель містить *модель бізнес-класів* і *модель бізнес-прецедентів*. *Модель бізнес-класів* – це діаграма класів верхнього рівня, що ідентифікує й зв'язує між собою бізнес-об'єкти. *Модель бізнес-прецедентів* – це діаграма прецедентів верхнього рівня, що ідентифікує основні функціональні будівельні блоки системи.

Методи виявлення вимог розподіляються на традиційні та сучасні.

До традиційних належать:

– інтерв'ювання;

– анкетування;

– спостереження;

– вивчення документів програмних систем.

До сучасних:

– прототипування;

– спільне розроблення програмних додатків (JAD-метод);
– швидке розроблення програмних додатків (RAD-метод).

Використання ***інтерв'ю*** являє собою основний *метод для* збору інформації. Більшість інтерв'ю проводяться із зацікавленими особами, інтерв'ю з якими дозволяють виявити більшою мірою вимоги, що випливають із прецедентів. Якщо бізнес-аналітик не має достатнього досвіду в проблемній області, можна також про інтерв'ювати відповідних експертів.

Проблеми інтерв'ювання:
– замовник може не знати, чого хоче;
– не хоче співпрацювати;
– не виражає вимог.

Існує 2 основних види інтерв'ю:
1. Структуроване (формальне) – слід заздалегідь підготувати питання, 2 категорій: питання з відкритим безліччю відповідей і питання із замкнутим безліччю відповідей.
2. Неструктуроване (не формальне).

Використання анкет або ***анкетування*** (*questionnaires*) – ефективний спосіб збору інформації від багатьох замовників. Звичайно анкети використовуються додатково до інтерв'ю, а не замість них. Виняток можуть становити проекти з низьким ризиком, цілі яких ясно окреслені. Для таких проектів звичайно досить використати анкети із запитаннями, що носять пасивний характер і не відрізняються великою глибиною. В анкетуванні найчастіше використовуються питання з замкнутим списком відповідей.

У загальному випадку, анкетування менш продуктивне, ніж використання інтерв'ю, оскільки до питань або можливих відповідей не можна внести додаткову ясність.

Питання можуть приймати форму:

1. Багатоальтернативні питання – при відповіді на ці питання респондент повинен вказати один або більше відповідей, вибравши їх із прикладеного списку. Крім того, іноді допускаються додаткові коментарі до питань з боку респондента.
2. Рейтингові питання – при відповіді на цей тип питань респондент повинен висловити свою думку щодо висловле-

ного твердження. Для цього можуть бути використані такі рейтингові значення як: абсолютно згоден, згоден, ставлюся нейтрально, не згоден, абсолютно не згоден, не знаю.

3. Питання з ранжируванням – цей тип питань передбачає ранжування відповідей за допомогою привласнення ним послідовних номерів – процентних значень і використання інших засобів упорядкування.

Спостереження

Спостереження: активне – занурення в середу, пасивне – спостереження, питання.

Вивчення документів і програмних систем є неоціненним методом виявлення як вимог типу прецедентів, так і вимог, зв'язаних зі знанням проблемної області. Цей метод використовується завжди, хоча він може стосуватися тільки окремих сторін системи.

Таблиця 3.1 – Вивчення документів і програмних систем

Організаційні документи:	**Системні форми і звіти**
Форми ділових документів (по можливості – заповнені)	Системні моделі аналізу і проектування
Опис робочих процедур	Звіти разом з документацією
Посадові обов'язки	Системні керівництва по експлуатації
Методичні посібники	Призначена для користувача документація
Бізнес-плани	Технічна документація
Схеми організаційних структур	Копії екранів
Внутріофісна кореспонденція	
Протоколи нарад	

Вимоги, що формулюються у вигляді прецедентів (*use case requirements*), виявляються за допомогою вивчення існуючих організаційних документів, системних форм і звітів.

Прототипування (*prototyping*) – це найбільше часто використовуваний сучасний метод виявлення вимог. Програмні прототипи конструюються для візуалізації системи або її частини для замовників з метою одержання їхніх відгуків.

Існують два основні різновиди прототипів:

– *одноразовий прототип* («*throw-away*» *prototype*), що після того, як виявлення вимог завершене, просто відкидається. Розробка «одноразового» прототипу націлена тільки на етап установлення вимог ЖЦ ПЗ. Як правило, цей прототип концентрується на найменш зрозумілих вимогах;

– *еволюційний прототип* (*evolutionary prototype*), що зберігається після виявлення вимог і використовується для створення кінцевого програмного продукту. Еволюційний прототип націлений на прискорення постачання продукту. Як правило, він концентрується на ясно викладених вимогах, так що першу версію продукту можна надати замовникові досить швидко (хоча її функціональні можливості, як правило, неповні).

JAD-метод повністю відповідає своїй назві – це спільна розробка додатків (*Joint Application Development*), здійснювана в ході одного або декількох нарад із залученням всіх учасників проекту. Хоча ми відносимо *JAD*-підхід до сучасних методів виявлення вимог, цей метод був уперше уведений наприкінці 1970-х років компанією *IBM*.

JAD-метод ґрунтується на груповій динаміці. Групові зусилля більш перспективні з погляду одержання кращого вирішення проблем. Групи сприяють підвищенню продуктивності, швидше навчаються, схильні до більш кваліфікованих висновків, дозволяють виключити багато помилок, приймають ризиковані рішення, концентрують увагу учасників на найбільш важливих питаннях, об'єднують людей і т. д.

Метод швидкого розроблення додатків (*Rapid Application Development-RAD*) – це щось більше, ніж метод виявлення вимог – це цілісний підхід до розроблення ПЗ. Як ясно з назви методу, він припускає швидку поставку системних рішень. Технічна перевага відступає на друге місце порівняно зі швидкістю поставки.

Технологія *RAD* містить у собі 5 підходів, перелічених нижче:

– еволюційне прототипування;

– *CASE*-засоби з можливостями генерації програм і циклічною розробкою з переходом від проектних моделей до програми й назад;

– фахівці, що володіють розвиненими інструментальними засобами – *RAD* команда розробників. Кращі аналітики, проектувальники й програмісти, які тільки може залучити організація. Команда працює в рамках строгого часового режиму та розміщується разом з користувачами;

– інтерактивний *JAD*-метод–*JAD*-сесія, під час якої секретарі замінюються бригадою *SWAT*, оснащеною *CASE*-засобами;

– жорсткі часові рамки (*timeboxing*) – метод керування проектом, що відводить команді розробників фіксований період часу для завершення проекту. Цей метод перешкоджає «розповзанню рамок проекту»; якщо проект затягається, то рамки рішення звужуються, щоб дати можливість завершити проект вчасно.

3.2.2 Узгодження вимог

Вимоги, отримані від користувачів, можуть дублюватися або суперечити одне одному. Деякі вимоги можуть бути неясні або нереальні, інші вимоги можуть залишитися нез’ясованими. З цієї причини перш ніж вимоги потраплять до документу опису вимог, їх необхідно узгодити.

При умові що всі вимоги чітко ідентифіковані і пронумеровані можна сконструювати матрицю залежності вимог (матриця взаємодії).

Таблиця 3.2 Матриця залежності вимог

Вимоги	В1	В2	В3	В4
В1	X	X	X	X
В2	Конфлікт	X	X	X
В3			X	X
В4		Перекриття	Перекриття	X

Суперечливі вимоги необхідно обговорити з замовниками і по можливості переформулювати, для пом’якшення протиріч (фіксацію протиріччя, видиму для подальшої розробки, необхідно зберегти).

Перекриваються вимоги так само повинні бути сформульовані заново, що б виключити збіги.

Насправді узгодження й перевірка обґрунтованості вимог здійснюється паралельно з виявленням вимог. Після того як вимоги виявлені, вони піддаються певному рівню перевірки. Для всіх сучасних методів виявлення вимог, що пов'язані з так званою «груповою динамікою», це цілком природно. Як би там не було, після того як виявлені вимоги зібрані разом, вони в кожному разі повинні бути піддані ретельному обговоренню й перевірці.

Після того, як у результаті зняття протиріч і усунення повторів у вимогах, розроблено переглянутий набір вимог, їх необхідно піддати аналізу ризиків і призначити їм пріоритети. Аналіз ризиків спрямований на ідентифікацію вимог, що є потенційними джерелами труднощів у розробці. Призначення пріоритетів необхідне для того, щоб забезпечити можливість без труднощів змінити рамки проекту у випадку виникнення непередбачених затримок.

Вимоги можуть бути «ризикованими» внаслідок впливу різних факторів. Вимогам властиві наступні типові види ризиків:

– технічний ризик, коли вимогу технічно важко реалізувати;

– ризик, зв'язаний зі зниженням продуктивності, коли вимога, будучи реалізованою, може несприятливо позначитися на часі реакції системи;

– ризик, пов'язаний з порушенням безпеки, коли вимога, будучи реалізованою, може створити пролом у захисті системи;

– ризик, пов'язаний з процесом розроблення, коли для реалізації вимоги необхідне використання незвичайних методів розроблення, незнайомих розроблювачам (наприклад, методів формальної специфікації);

– ризик, пов'язаний з порушенням цілісності баз даних, коли вимога не може бути легко перевіреною та може призвести до суперечливості даних;

– політичний ризик, коли вимога може виявитися важкою для виконання із внутрішньополітичних причин;

– ризик, пов'язаний з порушенням законності, коли вимога може призвести до порушення чинних законів або очікуваної зміни закону;

– ризик, пов’язаний з мінливістю, коли вимога може потенційно змінюватися або еволюціонувати протягом процесу розроблення.

В ідеалі *пріоритети* вимогам призначають окремі замовники в процесі виявлення вимог. Потім вони узгоджуються на нарадах і знову змінюються після додавання до них факторів ризику.

3.2.3 Рівні вимог

Вимоги до ПЗ складаються з 3-х рівнів:

- бізнес вимоги;
- вимоги користувачів;
- функціональні вимоги.

До того ж, кожна система має свої нефункціональні вимоги.

Бізнес вимоги містять високорівневі цілі організації або замовників системи. Вони можуть бути записані в документі про спосіб і межах проекту, в якому пояснюється, чому організації потрібна така система, тобто, описані цілі, які організація має намір досягти з її допомогою. Вимоги користувачів описують цілі і завдання, які користувачам дозволить вирішити система. Ці вимоги можуть бути записані в документ про варіанти використання, де вказано, що клієнти зможуть зробити за допомогою системи.

Функціональні вимоги визначають функціональність системи, яку розробники повинні побудувати, щоб користувачі змогли виконати свої завдання в рамках бізнес вимог.

Системні вимоги – це високорівневі вимоги до продукту.

Бізнес правила включають корпоративні політики, урядова постанова, промислові стандарти і обчислювальний алгоритм.

Функціональні вимоги документуються в специфікації вимог до програмного забезпечення, де описується так повно, як необхідно очікуване поведінка системи.Специфікація вимог до ПЗ використовується при розробці, тестуванні, гарантії якості продукту, управлінні проектом і пов’язаним з проектом функцій.

На додаток до функціональним вимогам, специфікація містить нефункціональні, де описані цілі і атрибути якості.

Атрибути якості – це додатковий опис функцій продукту, виражені через опис його характеристик, важливих для користувачів або розробників (легкість і простота використання, легкість переміщення, цілісність, ефективність і стійкість до збоїв).

Обмеження гойдаються вибору можливості розробки зовнішнього вигляду і структури проекту.

Яких вимог не повинно бути:

– деталей дизайну або реалізації;

– даних про планування проекту;

– відомостей про тестування.

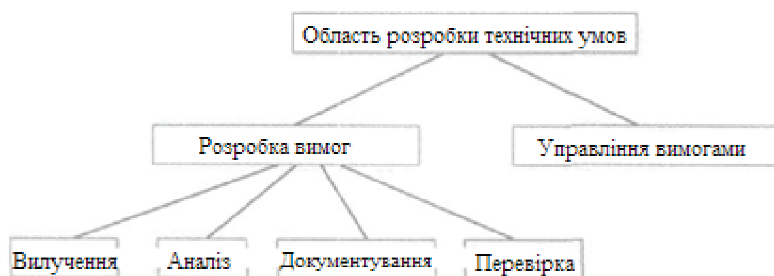

Рисунок 3.1 – Область розробки технічних умов

У підетапи розробки вимог входять всі дії, що включають збір, оцінку та документування вимог, для програмного забезпечення або продуктів, що містять програмне забезпечення, в тому числі:

– ідентифікація класів користувачів для даного продукту;

– з’ясування потреб тих, хто представляє кожен клас користувачів;

– визначення завдань і цілей користувачів, а також бізнес-цілей, з якими ці завдання пов’язані;

– аналіз інформації, отриманої від користувачів, щоб відокремити завдання від функціональних і не функціональних вимог, бізнес-правил, передбачуваних рішень і надходять ззовні даних;

– розподіл низькорівневих вимог, за компонентами ПЗ, розподілених в системній архітектурі;

– встановлення відносної важливості атрибутів якості;

– встановлення пріоритетів реалізації;

– документування зібраної інформації і побудова моделей;

– перегляд специфікації вимог, який дозволяє впевнитися в тому, що запити користувачів усіма розуміються однаково, і усунення виникаючих проблем до передачі документа розробникам.

Управління вимогами – визначається як «вироблення і підтримку взаємної згоди з замовниками з приводу вимог щодо розроблюваного ПЗ». Ця угода втілюється в специфікації (у письмовій формі) і в моделях. Розробники також повинні прийняти задокументовані вимоги і висловитися за створення цього продукту. До дій з управління вимогами відносяться:

– визначення основної версії вимог (моментальний зріз вимог для конкретної версії продукту);

– перегляд передбачуваних змін вимог і оцінка ймовірності впливу кожної зміни до його прийняття;

– включення схвалених змін вимог до проекту встановленими способами;

– узгодження плану проекту з вимогами;

– обговорення нових зобов’язань, заснованих на оціненому вплив зміни вимог;

– відстеження окремих вимог до їх дизайну, вихідного коду та варіантів тестування;

– відстеження статусу вимог і дій зі зміни протягом усього проекту.

3.2.4 Керування вимогами

Вимогами необхідно керувати. Керування вимогами являє собою частину загального керування проектом. Воно пов’язане з трьома основними питаннями:

1. Ідентифікація, класифікація, організація й документування вимог.

2. Зміна вимог (за допомогою процесів, що встановлюють способи висування, узгодження, перевірки вірогідності та документування неминучих змін до вимог).

3. Простежуваність вимог (за допомогою процесів, що підтримують відносини взаємозалежності між вимогами й іншими системними артефактами, а також, власно, між вимогами).

Вимоги описуються природною мовою, наприклад: «Система повинна запланувати наступний телефонний дзвінок клієнтові по запиту», «Система повинна автоматично набирати запланований телефонний номер» і т. д.

Типова система може складатися з сотень або тисяч формулювань вимог. Для належного керування такою величезною кількістю вимог їх необхідно пронумерувати за допомогою певної *схеми ідентифікації*. Схема може включати *класифікацію* вимог у вигляді груп, що легше піддаються керуванню.

Існує декілька методів ідентифікації й класифікації вимог:

– *унікальний ідентифікатор* – звичайно послідовний номер, привласнений вручну або згенерований з використанням бази даних *case*-засобу;

– *послідовний номер усередині ієрархії документа* – привласнюється з урахуванням положення вимог у межах документа опису вимог;

– *послідовний номер у межах категорії вимог* – привласнюється на додаток до мнемонічного імені, що позначає категорію вимог.

Вимоги можна впорядкувати у вигляді ієрархічно впорядкованої структури, наприклад відношення батько-нащадок. Відношення батько-нащадок подібно відношенню композиції Батьківська вимога складається з дочірніх вимог. Дочірня вимога – це фактично «під-вимога» батьківської вимоги.

Ієрархічні відносини дозволяють увести додатковий рівень класифікації вимог. Це може безпосередньо позначатися в ідентифікаційному номері (вимога, пронумерована як 4.9, може бути дев'ятим нащадком «батька» з ідентифікаційним номером, рівним 4).

Простежуваність вимог (*requirements traceability*) – це всього лише частина керування змінами. Блок вимог техно-

логії керування змінами підтримує відносини простежуваності, щоб фіксувати зміни, що виходять від або внесені у вимоги протягом ЖЦ розробки.

3.2.5 Бізнес-модель вимог

На етапі *встановлення вимог* здійснюється виявлення вимог і їх визначення, переважно у вигляді формулювань природною мовою. Формальне моделювання вимог з використанням мови UML проводиться пізніше на етапі аналізу або *специфікації вимог*. Проте під час установлення вимог постійно ведеться діяльність з узагальненого візуального подання зібраних вимог, що називається *бізнес-моделюванням вимог*.

Внаслідок того, що вимоги піддаються постійним змінам, напевно, найбільше занепокоєння при розробленні доставляє так зване *«розповзання рамок»* системи. Хоча деякі зміни вимог неминучі, необхідно суворо стежити за тим, щоб заявлені зміни не виходили за межі прийнятих рамок проекту.

Щоб відповісти на запитання про рамки системи, необхідно знати, в якому контексті функціонує наша система. Необхідно знати, які зовнішні сутності – інші системи, організації, люди, машини й тощо – розраховують на отримання послуг від нас або готові надати послуги нам.

Тому рамки системи можна визначити, позначивши зовнішні сутності та вхідні/вихідні потоки даних між зовнішніми сутностями та нашою системою. Система, що проектується, одержує вхідну інформацію та виконує необхідну обробку з метою вироблення вихідної інформації. Усяка вимога, що не може бути підтримана за рахунки внутрішньосистемних можливостей обробки, виходить за рамки системи.

Модель бізнес-прецедентів являє собою модель прецедентів на верхньому рівні абстракції. Модель бізнес-прецедентів визначає узагальнені бізнес-процеси. Бізнес-прецедент відповідає тому, що іноді називають можливостями системи. (Можливості системи визначаються в документі, що описує бачення системи (system vision). Якщо при розробці системи

наводиться документ опису бачення системи, він може використовуватись замість моделі бізнес-прецедентів).

Діаграма бізнес-прецедентів концентрується на архітектурі бізнес-процесів. Ця діаграма дає можливість глянути на передбачуване поводження системи так сказати «з висоти пташиного польоту». Неформальний опис кожного з бізнес-прецедентів повинний бути коротким, орієнтованим на ділову сторону системи, і концентруватися на основних потоках видів діяльності.

На етапі аналізу бізнес-прецеденти перетворюються в прецеденти. Саме на цьому етапі визначаються детальні прецеденти, і неформальний опис розширюється за рахунок включення в нього підпроцесів і альтернативних процесів, деяких копій екранів, що демонструють *GUI*-інтерфейс, а також взаємозв'язків між уведеними прецедентами.

Суб'єкти (actor) діаграми бізнес-прецедентів відрізняються від зовнішніх сутностей на діаграмі контексту. Суб'єкти активні. Вони керують процесом. Вони активізують прецеденти, відправляючи їм повідомлення про події. Прецеденти управляють подіями. Лінії, що зв'язують суб'єктів і прецеденти, – це не потоки даних. Ці лінії зв'язку являють собою потік подій, що виходять від суб'єктів і потік відгуків, що виходять від прецедентів.

Модель бізнес-класів – це модель класів. Як і у випадку з бізнес-прецедентами, різниця міститься в рівні абстрагування.

3.2.5 Документ опису вимог

Документ, що описує вимоги, є відчутним результатом етапу встановлення вимог.

Шаблони для документів опису вимог широко доступні. Згодом кожна організація розробляє свої власні стандарти, які відповідають прийнятій в організації практиці, корпоративній культурі й т. п.

Документ опису вимог повинен створити прецедент для системи.

Приклад змісту документа:

1. Попередні зауваження до проекту
   a. Мета й рамки проекта
   b. Діловий контекст
   c. Учасники проекта
   d. Ідеї відносно рішень
   e. Огляд документа
2. Системні сервіси
   a. Рамки системи
   b. Функціональні вимоги
   c. Вимоги до даних
3. Системні обмеження
   a. Вимоги до інтерфейсу
   b. Вимоги до продуктивності
   c. Вимоги до безпеки
   d. Експлуатаційні вимоги
   e. Політичні і юридичні вимоги
   f. Інші обмеження
4. Проектні питання
   a. Відкриті питання
   b. Попередній план-графік
   c. Попередній бюджет
5. Додатки
   a. Глосарій
   b. Ділові документи та форми
   c. Посилання

Основна частина документа опису вимог присвячена визначенню системних сервісів. Ця частина може займати до половини всього обсягу документа. Ця частина документа може містити узагальнені моделі – моделі бізнес-вимог.

Рамки системи можна моделювати за допомогою діаграми контексту.

Функціональні вимоги можна моделювати за допомогою діаграми бізнес-прецедентів. Однак діаграма охоплює перелік функціональних вимог тільки в найбільш загальному ви-

гляді. Всі вимоги необхідно позначити, класифікувати й визначити.

Вимоги до даних можна моделювати за допомогою діаграми бізнес-класів.

Системні сервіси визначають, що повинна робити система. Системні обмеження визначають, наскільки система обмежена при виконанні обслуговування. Системні обмеження зв'язані з такими видами вимог:

– вимоги до інтерфейсу, що визначають як система взаємодіє з користувачами;

– вимоги до продуктивності, що у вузькому сенсі задають швидкість відгуку системи, з якої повинні виконуватися різні завдання;

– вимоги до безпеки, які описують права доступу користувача до інформації, контрольовані системою.

– експлуатаційні вимоги, які визначають програмно-технічне середовище, якщо воно відоме на етапі проектування, у якому повинна функціонувати система.

– політичні й юридичні вимоги, які в основному мають на увазі.

3.3 Моделювання бізнес-процесів

Моделювання бізнес-процесів в останні роки стало актуальною тенденцією і використовується на практиці для вирішення широкого спектра завдань. Один з найбільш типових способів застосування подібних моделей – це вдосконалення самих модельованих процесів.

Діяльність:

– повторювана – процеси;
– не є повторюваною – проекти.

Як правило, процеси становлять значну частину діяльності організації.

Процес – це пов'язаний набір повторюваних дій, які перетворять вихідний матеріал і (або) інформацію в кінцевий продукт або послугу відповідно до попередньо встановлених правил, враховуючи, що процес має кінце-

вий результат, розгляд діяльності компанії як сукупності процесів дозволяє більш оперативно реагувати на зміну зовнішніх умов, уникати дублювання діяльності та витрат, що не приводить до бажаного результату, і правильно мотивувати співробітників для його досягнення.

Моделювання бізнес-процесу зазвичай означає їх графічно формалізований опис.

Побудова бізнес-моделі є одним з ключевих моментів специфікації вимог.

Бізнес-архітектура – це область, яка визначається вищими керівниками, відповідальними за основні функції організації і включає в себе твердження з приводу місій і цілей організації, критичні фактори успіху, бізнес стратегії, описи функцій а також структури і процеси, необхідні для реалізації функцій.

Ключем до побудови гарної бізнес-архітектури є визначення бізнес процесів, їх функцій і характеристик. Це стає основою для побудови архітектури ІТ-додатків, які забезпечують автоматизовану підтримку цих процесів.

В рамках моделі бізнес-архітектури виділяються наступні основні компоненти:

– бізнес процеси / цілі і стратегія побудови бізнесу;
– організаційна компонента / організаційне оточення;
– інформація / інформаційне оточення;
– додаток / забезпечує оточення.

Бізнес архітектура включає в себе наступні аспекти:

– Бізнес стратегія, функції та організаційна структура – збори цільових установок, планів і структур організацій.

– Архітектура бізнес-процесів, яка визначає основні функціональні області організації.

– Показники результативності – цей аспект полягає у визначенні ключових показників результативності (КПР) роботи організації, їх поточних і бажаних рівнів, модель КПР використовується як засіб моніторингу виконання бізнес-процесів.

Balanced scorecard – широку популярність ця методика отримала, яка представляє собою систему, засновану на при-

чинно-наслідкових зв'язках між стратегічними цілями, що відображають їх параметрами і факторами отримання планованих результатів. Вона розглядає 4 проекції: фінансову, взаємини зі споживачем, операційні ефективності, мети і завдання яких взаємопов'язані і відображені фінансовими і нефінансовими показниками.

Види Balanced Scorecard:

– Фінансові: Якими нас бачать акціонери?

– Внутрішні: На яких процесах ми повинні досягти успіху?

– Інновації: Чому ми повинні навчитися, щоб рости і процвітати?

– Клієнт: Як наші клієнти сприймають нас?

Структурно-організаційна компонента.

Організаційна компонента в моделі бізнес-архітектури відповідає на питання: хто за що відповідає в бізнес-процесах. Розподіл відповідальності за результати бізнес-процесу визначаються у вигляді завдання ролей (повноважень), інкапсуляція даних ролей в бізнес процеси і закріплення ролей між конкретними персоналіями. Відповідно, організаційна компонента повинна підтримувати опис існуючої в організації організаційно-штатної структури, а також відображати закріплене в посадових інструкціях розподіл функціональних обов'язків учасників бізнес процесу. Метою розробки моделі організаційної компоненти є забезпечення можливості отримання якісних і кількісних оцінок, ефективності використання кадрових ресурсів в реалізації бізнес процесів, і як наслідок – пошук варіантів оптимізації організаційної структури підприємства.

Структура інформаційної компоненти: в рамках моделі бізнес архітектури зміст і детальність відображення інформаційної компоненти визначається ступенем її впливу на підтримку бізнесу. Інформаційна компонента повинна бути корелирующим відображенням бізнес архітектури. В рамках моделі бізнес-архітектури інформаційна компонента включає в себе всі ті інформаційні об'єкти (потоки, документи, дані), які безпосередньо пов'язані з бізнес подіями. Метою розробки моделі інформації та моделі даних є створення гра-

фічних уявлень потреб організації і окремих бізнес процесів в інформації. Це стає основою для реорганізації бізнес процесів і конструювання нових прикладних систем, опису взаємодій та інформаційного обміну, який відбувається між організацією і клієнтом-партнером. Інформаційна компонента представляється в такому вигляді, щоб була забезпечена можливість розгляду моделі інформації на різних рівнях розгляду абстракції, виходячи з потреб бізнес-процесу.

Організація компоненти «Додатки».

Компонента додатка орієнтована на відображення того, які прикладні системи потрібні підприємству для виконання бізнес-процесів. Детальність опису прикладних систем повинна забезпечуватися на рівні, достатньому для розуміння складу автоматизуються функцій, які зберігаються (оброблюваних) операційних даних (документів)) що в кінцевому підсумку дає об'єктивне уявлення про рівень її значущості для організації в цілому. Опис компоненти програми повинно бути не тільки досить для розуміння в якій частині бізнес процесів забезпечується підтримка, але і з точки зору оцінки витрат і вигод щодо використання системи.

3.3.1 Моделювання

Об'єктом моделювання може виступати будь-яка сутність, підходи по моделювання універсальні, і можуть бути застосовні як до архітектури корпоративно-операційної системи, або компанії в цілому, так і при проектуванні окремих інформаційних систем.

Моделювання (по ISO-15704) – абстрактне уявлення реальності в будь-якій формі (наприклад, у фізичній, символічної, графічної або дескриптивної), призначене для подання певних аспектів цієї реальності і дозволяє відповідати на питання, що розглядаються.

Моделі можуть бути класифіковані за різними критеріями, наприклад:

1. Формальні (використовують загальноприйняті правила, нотації і засоби) і неформальні.

2. Кількісні (дозволяють виробляти чисельні оцінки і перевірки) і якісні (призначені для розуміння поведінки і структури системи).
3. Описові (призначені тільки для сприйняття людини) або виконувані (дозволяють досліджувати їх поведінку і використовувати отримані результати для висновків про вихідний об'єкті).

Загальні принципи моделювання:

1. Принцип здійсненності: створювана модель насамперед повинна забезпечувати досягнення поставлених цілей, таким чином перш ніж приступити до збору інформації про об'єкт потрібно чітко визначити межі області моделювання, цілі і кількісні показники їх досягнення.
2. Принцип інформаційної достатності: при повній відсутності інформації про досліджуваний об'єкт побудова його моделі неможливо. При наявності повної інформації моделювання не має сенсу. Існує певний критичний рівень апріорних відомостей про об'єкт, при досягненні якого має сенс переходити від етапу збору інформації до етапу власне побудови моделі. В даному випадку закладаються умови для виконання такого значимого вимоги, як адекватність моделі, а саме досягнення розумного балансу між детальністю і споживчими якостями моделі.
3. Принцип множинності моделі: створювана модель повинна відображати ті властивості реального об'єкта, які впливають на обрані показники ефективності. При використанні будь-якої конкретної моделі пізнаються тільки деякі області дійсності. Для більш повного дослідження реального об'єкта необхідний ряд моделей, що дозволяють з різних сторін і з різною деталізацією відображати розглянутий процес.
4. Принцип агрегування: в більшості випадків складну систему можна представити у вигляді сукупності агрегатів (підсистем), для адекватного опису яких виявляються придатними деякі стандартні схеми.

5. Принцип відділення: досліджувана область як правило має в своєму складі кілька ізольованих компонентів, внутрішня структура яких досить прозора, або не подає безпосереднього інтересу для мети проекту. В такому випадку її місце в моделі займає умовний порожній блок, для якого визначаються тільки значні вхідні і вихідні інформаційні потоки.

3.3.2 Об´єктний аналіз

Об'єктний аналіз – це метод дослідження не бізнес-процесів в цілому, а його неподільних найменших функціональних частин системи (на даному рівні розгляду) – структурних елементів (об'єктів), пов'язаних між собою деякими відносинами.

Процес:

– Це безліч внутрішніх кроків діяльності, що починаються з одного і більше входу, і закінчуються створенням продукції, необхідної клієнту.

– Це потік роботи, що проходить від одного фахівця до іншого або від одного відділу до іншого (в залежності від рівня розгляду).

– Це процедура або набір процедур, які спільно реалізують бізнес-завдання або політичну мету підприємства, як правило в рамках організаційної структури, яка описує функціональні ролі і відносини.

– Це взаємонезалежні компонент виробничої системи, що перетворює вхід в один або кілька виходів відповідно до попередньо встановлених правил.

– Це пов'язаний набір повторюваних дій (функцій), які перетворять вихідний матеріал і / або інформацію в кінцевий продукт (послугу) відповідно до визначених критеріїв.

Процесний підхід до моделювання дозволяє:

1. Перейти від «точкового» текстового опису діяльності до повного формалізованого графічного опису діяльності, інтегруючим стрижнем якого є модельне уявлення бізнес-процесу.

2. Виділити і використовувати процеси як об’єкти управління
3. Змінити орієнтацію вектора управління компанії від «вертикальної» («на начальника») до «горизонтальної» («на замовника»).

3.3.3 Класифікація бізнес-процесів

Типи бізнес-процесів:

– Основні (або ключові) процеси – стійкі процеси виробничо-господарської діяльності підприємства, орієнтовані на створення кінцевого продукту або послуги; в складі основних процесів може бути виділений контур керуючого впливу (власне дію і контроль за його виконанням).

– Процеси, що забезпечують нормальне виконання основних процедур і змінюються в залежності від зміни складу, технологій основних процесів.

– Процеси зовнішньої взаємодії – це процеси взаємодії з об’єктами, що не входять в узгоджене опис предметної області.

Компоненти процесу:

1) назва;
2) реалізована функція;
3) учасники;
4) відповідальна особа;
5) межі;
6) вхідні і вихідні потоки (вхідні потоки – це матеріали, послуги, та / або інформація, що перетворюються процесам для створення вихідними потоками. Вихідні потоки – це результат перетворення вхідних потоків);
7) необхідні ресурси – сприяючі чинники, які не перетворюються щоб стати вихідним потоком (персонал, обладнання, приміщення, інформація);
8) мета процесу;
9) метрики процесу;
10) можливі ризики.

Власник процесу, що несе повну відповідальність за процес, і наділена повноваженнями щодо цього процесу. Він не стосуєть-

ся функцій, які виконуються в рамках процесу окремими департаментами. Йому важлива успішна реалізація всього процесу, і перш за все його продуктивність, ефективність, адаптованість. Власник процесу забезпечує взаємодію з постачальниками вхідних потоків процесу і з споживачами його результатів.

Основні складові моделі бізнес-процесу – це функції, ресурси, документи і дані, учасники процесу, матеріали, продукти, послуги.

Для аналізу процесів рекомендується використовувати досвід консультантів, еталонні і референтні моделі, чек-листи, і інші статистичні методи, що застосовуються в сфері управління якістю.

Аспекти аналізу процесу:

– аналіз топології процесу;
– аналіз характеристик процесу;
– аналіз помилок процесу;
– аналіз динаміки виконання процесу;
– аналіз ризиків процесу;
– аналіз ресурсного оточення процесу;
– аналіз можливостей стандартизації процесу.

Аналіз топології процесу ставить собі за мету досягнути максимально зрозумілого перебігу процесу, що відображає при цьому або реальний стан речей, або оптимальний з урахуванням доступності ресурсів.

Етапи аналізу характеристик процесу:

1. Визначення основних характеристик (показників процесу).
2. Визначення метрик характеристик для їх оцінки.
3. Моніторинг метрик характеристик процесу.

Основними характеристиками процесу є наступні показники:

– результативність – характеризує відповідність результатів процесу потребам і очікуванням споживачів;
– визначеність – відображає ступінь, з якою реальний процес відповідає опису;
– керованість – характеризує ступінь, в якій здійснюється управління виконання процесів виробництва необхідних

продуктів або послуг, що відповідають певним цільовим показникам;

– ефективність – відображає, наскільки оптимально використовуються ресурси при досягненні необхідного результату процесу;

– повторення – характеризує здатність процесу створювати вихідні потоки з однаковими характеристиками при повторних його реалізаціях;

– гнучкість (адаптованість) – це здатність процесу пристосовуватися до зміни зовнішніх умов, перебудовуватися так, щоб це не призводило не результативна, не ефективність;

– вартість процесу – визначає сукупну вартість виконання функцій процесу і передачі результатів від однієї функції до іншої.

3.3.4 Етапи аналізу помилок процесу

Основні етапи:

– класифікація можливих помилок процесу;

– опис помилок процесу;

– виявлення помилок в процесі.

Можливі помилки, які можуть виникати при моделюванні бізнес-процесів:

– незавершеність – наявність прогалин в описі процесів;

– невідповідність – неадекватне використання інформаційних ресурсів в різних частинах процесу, що призводить до спотвореного сприйняття інформації або до неясності вказівок;

– ієрархічна несумісність – несумісність процесу з підпроцесами, його складовими;

– спадкова несумісність – наявність конфлікту між основними і подальшими процесами.

Динаміка процесів досліджується за допомогою динамічної (імітаційної) моделі.

Імітаційне моделювання – це методика, що дозволяє представляти в рамках динамічної комп'ютерної моделі протікання процесів, дії людей і застосування технологій, що використовуються в досліджуваних процесах.

3.3.5 Аналіз ризиків процесу

Операційний ризик – ризик прямих або непрямих збитків, який виникає в результаті невірного виконання бізнес-процесів, неефективності процедур внутрішнього контролю, технологічних збоїв, несанкціонованих дій персоналу або зовнішнього впливу.

Операційний ризик критичний для тих процесів, які характеризуються:

- значимістю для діяльності організації в цілому;
- великим числом транзакцій в одиницю часу;
- складністю системи технічної підтримки.

Етапи аналізу ризиків процесу:

1) структуризація ризиків;
2) опис ризиків та процесів, їх запобігання;
3) визначення ризиків в бізнес-процесах.

Аналіз ресурсного оточення процесу.

Основу процесу складають виконувані функції, і для виконання кожної з них потрібно ресурси:

- людські – учасники процесу;
- виробничі;
- матеріальні;
- інформаційні;
- інтелектуальні – знання і повноваження учасників.

Аналіз можливостей стандартизації процесу (створення еталонних референтних моделей).

Еталони можуть бути базовими критеріями для інжинірингу бізнес-процесів. Складання власного бізнес процесу з аналогічним процесом, узятим за зразок, дозволяє отримати цільові чи орієнтовні показники. Така процедура називається еталонним порівнянням. Розбіжність між характеристиками еталонного процесу і власними показниками може підказати, як краще організувати у себе бізнес-проект.

3.3.6 Складові моделі об´єкта

Основні складові:

- використовувані методики;

– нотація;
– лінгвістичне забезпечення.
– Види нотації бізнес-процесів:
– IDEF0;
– BPMN 2.0 – Business Process Modeling Notation;
– UML.

Нотація BPMN – була розроблена в 2001 – 2004 рр. групою BPMI.org, для стандартизованого опису бізнес процесів, зрозумілу як менеджерам і бізнес-аналітикам, так і розробникам ПЗ з можливістю подальшого збереження цього опису BPML. На відміну від UML, нотація BPMN включає лише ті елементи і поняття, які необхідні для моделювання бізнес-процесів. Її важливою особливістю є можливість встановити однозначну відповідність між однозначним описом елементів графічної нотації і описом мови на базі XML. У BPMN є тільки один тип діаграм – це діаграми бізнес-процесів (BPD), за допомогою яких описують послідовність виконання операцій в бізнес процесі.

Для побудови діаграми використовуються чотири види об’єктів:

1) потоку;
2) зв’язку;
3) розділові доріжки;
4) артефакти.

Дії зображуються прямокутниками з закругленими кутами. Вони підрозділяються на завдання (елементарні дії, які не підлягають декомпозиції) і підпроцеси (складові дії, які самі можуть бути представлені у вигляді бізнес-процесів). Підпроцеси можуть бути зображені на діаграмі в згорнутому або розгорнутому вигляді. Завдання, згорнуті і розгорнуті підпроцеси можуть бути обладнані маркерами, що вказують деякі характеристики їх виконання.

Маркери BPMN:

– Маркер циклу.
– Маркер багатопримірниковості.
– Маркер компенсації.

Події BPMN служать для позначення різних подій, які можна почати, перервати і закінчити хід процесу. Проміжні

і більшість початкових подій можуть бути забезпечені тригерами, які відображають суть події. Використання подій на діаграмах не є обов'язковим.

Шлюзи служать для управління розподілом і з'єднанням декількох ліній ходу процесу. Вони бувають єдиного, множинного, і складного вибору, а також паралельного виконання. Шлюзи єдиного вибору поділяються на засновані на даних (рішення про подальший перебіг процесу) і засновані на подіях; рішення приймається виходячи з того, що відбувається в цій точці події.

Шлюзи:

- оператор що виключає АБО, керований даними;
- оператор що виключає АБО, керований подіями;
- оператор що включає АБО;
- оператор І.

3.3.7 Складний оператор

У BPMN визначено три типи зв'язків:

1. Зв'язки потоку, що відображають послідовність виконання дій і з'єднує між собою об'єкти потоку (для них може бути задана умова переходу).
2. Зв'язки повідомлень, що відображають потік повідомлень між учасниками бізнес-процесу.
3. Асоціації, призначені для прив'язування до об'єкта потоку додаткової інформації у вигляді тексту або інших об'єктів.

До розділових доріжок відносяться пули і доріжки.

У вигляді пулів представляється учасник бізнес-процесу, наприклад, компанія, постачальник, клієнт і т.д. Пул може служити для поділу складових бізнес-процес дій між кількома учасниками, але може і не мати внутрішніх елементів, а представляти учасника процесу як чорний ящик. Якщо необхідно впорядкувати бізнес процес всередині пулу, його поділяють на доріжки, принцип розділення яких залишається на розгляд аналітика. Якщо бізнес-процес зображений всередині пулу, він не може виходити за його межі, тобто зв'язки

потоку можуть перетинати кордони доріжок всередині пулу, але не межі пулу. Взаємодія поміщеного всередину пулу бізнес-процесу з зовнішнім світом моделюється за допомогою зв’язків повідомлень. Зв’язки повідомлень можуть починатися і закінчувати як на об’єктах потоку всередині пулу, так і на кордоні пулу, однак вони не повинні з’єднувати об’єкти всередині одного пулу.

3.4 Практичні завдання

Завдання 1. Розгляньте концепцію створення програмного забезпечення для інтернет-магазину іграшок. Дайте відповіді на такі запитання:

Для кого призначений цей додаток? Хто входить до кола зацікавлених осіб? Дайте оцінку потенційних покупців в Україні та за кордоном.

Укажіть три властивості, які повинна мати система, та три властивості, яких не повинно бути.

Вкажіть дві системи, з якими буде працювати ваша система.

Вкажіть три найбільш важливі ризики.

Завдання 2. Підготуйте реферат з методів, що використовуються при підготовці інтерв’ю та анкетування.

Завдання 3. Підготуйте порівняльну характеристику *RAD* та *JAD* технологій.

Завдання 4. Для проекту системи «Інтернет-магазин подарунків» сформулюйте типові види ризиків.

Завдання 5. Виконайте завдання 1 для системи віртуального казино.

Завдання 6. Виконайте завдання 1 для системи дистанційного навчання.

Завдання 7. Для системи «*Інтернет-магазин подарунків*» сформулюйте 20 функціональних вимог, 10 вимог до якості продукту, 10 вимог до інтерфейсу, 5 вимог до апаратного забезпечення.

Завдання 8. Для системи «*Диспетчер таксі*» сформулюйте 10 функціональних вимог з боку диспетчера, 10 функціональних вимог з точки зору менеджера автопарку, 10 нефункціональних вимог.

Завдання 9. Розробіть прототипи інтерфейсу для системи «*Диспетчер таксі*».

Завдання 10. Складіть анкету для виявлення вимог для системи «*Інтернет магазин подарунків*».

Завдання 11. Підготуйте огляд методів що використовуються при керування вимогами.

Завдання 12. Розробіть документ Vision для системи «Інтернет-магазин подарунків».

Завдання 13. Розробіть документ *Vision* для підсистеми графічних елементів САПР радіоелектронної техніки.

3.5 Контрольні запитання

1. Які існують способи пошуку концепції нової системи?
2. На які питання потрібна давати відповідь концепція системи?
3. Хто виконує виявлення вимог?
4. Які традиційні методи виконуються для виявлення вимог?
5. У чому головні вади та недоліки використання прототипування?
6. З якою метою використовується еволюційний прототип?
7. Які підходи об'єднує в собі *RAD*?
8. Як створюються ієрархії вимог?
9. Як виконується ідентифікація та класифікація вимог?
10. Які типові ризики властиві вимогам?
11. З якими питаннями пов'язано керування вимогами?
12. Чи існує єдиний стандарт документа опису вимог?

3.6 Література до розділу

1. Брагина, Т.И. Нечеткий анализ проектного риска [Текст] / Т.И. Брагина, Г.В. Табунщик // Системи обробки інформації. Вип.3 (93). – 2011. – С. 15-21.
2. Вигерс, К. И. Разработка требований к программному обеспечению / К. И. Вигерс, Д. Битти. – СПБ: БХВ-Петербург, 2016. – 436 с.
3. Кон, М. Непрерывное развертывание ПО: автоматизация процессов сборки, тестирования и внедрения новых версий программ / Майк Кон. – М.:Издательский дом «Вильямс», 2012. – 256 с.
4. Мартин, Р. Чистый код. Создание, анализ и рефакторинг / Роберт Мартин. – СПБ.:Издательский дом «Питер», 2011. – 464 с.
5. Пат. 81169 Україна, МПК (2011) МПК2012 G06Q 10/06, G06Q 10/10, G06Q 90/00. Спосіб керування ризиками проектів [Текст] / Брагіна Тетяна Ігорівна; Табунщик Галина Володимирівна; заявник і патентовласник Запорізький національний технічний університет. – № u201214521; заявл. 18.12.2012; опубл. 25.06.2013, бюл. № 12.
6. Табунщик Г.В. Проектування, моделювання та аналіз інформаційних систем / Кудерметов Р.К., Притула А.В., Табунщик Г.В., Запоріжжя: ЗНТУ. -296 с.
7. Мацяшек Л. А. Анализ и проектирование информационных систем с помощью UML 2.0. ; пер. с англ. [Текст] / Мацяшек Л. А. – М.:Издательский дом «Вильямс», 2008. – 816 с.
8. Хамбл, Д. Непрерывное развертывание ПО: автоматизация процессов сборки, тестирования и внедрения новых версий программ / Джез Хамбл. – М.:Издательский дом «Вильямс», 2011. – 436 с.

4 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

На етапі аналізу розробники займаються конструюванням моделей та намагаються досягти глибокого розуміння вимог.

Аналітична модель – це точне, чітке уявлення завдання, що дозволяє відповідати на запитання та конструювати рішення. Вона описує три аспекти об'єктів: їх статичну структуру (модель класів), взаємодії між ними (модель взаємодії) та життєві цикли об'єктів (модель станів).

Метою аналізу предметної області є розробка точної, чіткої, доступної для розуміння та конкретної моделі реального світу.

4.1 Модель класів предметної області

Модель предметної області відображає статичну структуру системи в реальному світі та ділить її на окремі елементи, зручні для оперування. Модель предметної області описує реальні класи та відносини між ними.

Конструювання моделі класів предметної області виконується у наведеній нижче послідовності:

– виділити класи;

– підготувати словник даних;

– виділити асоціації;

– виділити атрибути об'єктів і зв'язків;

– організувати та спростити класи за допомогою спадкування;

– перевірити наявність маршрутів для найбільш ймовірних запитів;

– перейти до наступної ітерації та вточнити модель;

– переглянути рівень абстрагування;

– згрупувати класи в пакети.

<u>Виявлення класів</u>

Існує безліч підходів до виділення класів. Найбільш поширені:

–підхід на основі використання іменних груп;

–підхід на основі використання загальних шаблонів для класів;

–підхід на основі використання прецедентів;

–підхід CRC (Class-Responsibility-Collaboration).

Підхід на основі використання іменних груп (тобто іменників) припускає, що аналітик читає формулювання *документа опису вимог* у пошуках іменних груп. Кожний іменник розглядається як потенційний клас. Потім список усіх класів поділяється на наступні три групи:

– релевантні, або відповідні класи;

– нечіткі, або сумнівні класи;

– нерелевантні, або невідповідні класи.

До *нерелевантних* нележать класи, що виходять за рамки проблемної області. Для них не вдається дати формулювання їх призначення.

До *релевантних* класів відносяться класи, що належать до проблемної області. Іменники, що відображають імена цих класів, часто зустрічаються в документі опису вимог.

До *нечітких* належать класи, що не можна впевнено і беззастережно визнати відповідними. Їх аналізують більш глибоко, апотім відносять до однієї з попередніх груп.

Підхід на основі використання загальних шаблонів для класів дозволяє вивести потенційні класи на основі теорії родової класифікації об’єктів.

Барамі пропонує такий перелік груп:

– понятійний (концептуальний) клас. Він являє собою ідею, яку поділяє або з якою згодна значна спільність людей.

Наприклад, клас «*Резервування*»;

– клас подій. Подія не вимагає часу стосовно тимчасової шкали, що розглядається. Наприклад, клас «*Прибуття*»;

– організаційний клас. Організація – це будь-який вид цілеспрямованого об’єднання або сукупності сутностей. Наприклад, клас «*Туристичне бюро*»;

– клас людей. Під людьми тут розуміється роль, яку людина грає в тій чи іншій системі. Наприклад, клас «*Пасажир*»;

– клас місцерозташувань. Наприклад, клас «*Офіс*».

Дж. Рамбау, А. Джекобсон та Г. Буч пропонують наступну схему класифікації:

– фізичний клас (наприклад, клас «*Літак*»);

– бізнес-клас (наприклад, клас «*Резервація*»);

– логічний клас (наприклад, клас «*Розклад*»);

– прикладний клас (наприклад, клас «*Операція резервування*»);

– комп'ютерний клас (наприклад, клас «*Індекс*»);

– поведінковий клас (наприклад, клас «*Скасування резервування*»).

Даний підхід служить скоріше як корисне керівництво, але не визначає систематичного процесу, за допомогою якого можна було б виділити надійну та повну множину класів.

Підхід на основі використання прецедентів. Цьому підходу надається особливе значення в мові *UML*. Можна навіть сказати, що цей підхід рекомендується використовувати в рамках методології *RUP* (*Rational Unified Process*). Графічна модель прецедентів супроводжується неформальними описами, а також діаграммами послідовностей і кооперації для окремих прецедентів. Ці додаткові описи та кроки визначення діаграм (і об'єктів) потрібно виконати для кожного прецеденту. На основі цієї інформації можна прийти до узагальнень, необхідних для виявлення потенційних класів.

Підхід, що керується прецедентами, має особливості, притаманні підходу знизу-вгору. Після того, як прецеденти стають відомі, а *уявлення* про систему *з точки зору взаємодії,* щонайменше, частково визначено за допомогою діаграм послідовностей, об'єкти, що використовуються в цих діаграмах, приводять до виявлення класів.

Насправді цей підхід у чомусь схожий на підхід, який використовує іменні групи. Їх об'єднує те, що прецеденти специфікують вимоги. Обидва підходи спрямовані на вивчення формулювань, викладених у документі опису вимог, щоб виявити в підсумку потенційні класи. Те, що ці формулювання викладаються в оповідній формі або подані графічно, має другорядне значення. У будь-якому випадку на цьому етапі

ЖЦ розробки ПЗ більшу частину прецедентів можна описати лише в текстовій формі без діаграм взаємодії.

Підхід, заснований на прецедентах, має ті ж недоліки, що йпідхід, який використовує іменні групи. Будучи по суті підходом знизу-вгору в сенсі точності, він спирається на повноту та коректність моделей прецедентів. У результаті, він навіть може призвести до небажаного розбалансування ітеративного інарощуваного процесу розроблення ПЗ, при якому моделі прецедентів повинні бути завершені ще до побудови моделей класів. Загалом, які б не були цілі та засоби, це призводить до *функціонального підходу* (*function-driven approach*), прихильники об'єктно-орієнтованого підходу вважають за краще називати його *проблемно-орієнтованим (problem-driven)*.

Підхід CRC. Підхід *CRC* (*Class – Responsibility – Collaborators* – клас – відповідальність – «співробітники») являє собою щось більше, ніж метод виявлення класів, – це привабливий спосіб інтерпретації та вивчення об'єктів (а також і навчання об'єктному підходу). Найбільшу популярність підхід *CRC* отримав завдяки роботам Ребеки Вірфс-Брок (*Rebecca Wirfs-Brock*) та її колег Б. Вілкерсон (*B. Wilkerson*) і Л. Вінер (*L. Wiener*).

Підхід *CRC* включає в себе сеанси «мозкового штурму», проведення яких полегшується за рахунок використання спеціально підготовлених карток. Картки складаються з трьох відділень: *ім'я класу* записується у верхньому відділенні, *обов'язки класу* перераховані в лівому відділенні, а *співробітники* перераховані в правому відділенні. Обов'язки – це послуги (операції), які клас готовий виконати в інтересах інших класів. Для виконання багатьох обов'язків необхідна участь (обслуговування) з боку інших класів. Такі класи перераховуються як «співробітники».

Процес *CRC* – це живий процес, під час якого розробники «грають в карти», вони заповнюють картки іменами класів і призначають їм «обов'язки» та «співробітників» у ході виконання сценарію оброблення інформації (наприклад, сценарію прецеденту). У тих випадках, коли виникає потреба в якійсь послузі, аіснуючі класи не покривають її, створюється

новий клас, якому призначаються відповідні «обов'язки» та «співробітники». Якщо клас стає «надто зайнятим», він розділяється на кілька менших класів.

Підхід *CRC* відрізняється від інших підходів тим, що при його використанні виділення класів є результатом аналізу повідомлень, переданих між об'єктами для виконання завдань оброблення інформації. Акцент робиться на уніфікованому методі розподілу «інтелекту» в системі, і деякі класи можуть бути швидше отримані, виходячи з подібної технічної потреби, ніж виявлені у якості «бізнес-об'єктів» як таких. У цьому сенсі метод *CRC* може бути більш прийнятним для перевірки правильності вибору класів вже виявлених з допомогою інших методів. Підхід *CRC* також корисний при встановленні властивостей класів (що логічно випливають із «обов'язків» і типів «співробітників» класу).

Підготовка словника даних

Слова допускають дуже багато інтерпретацій, тому для всіх елементів моделі необхідно підготувати словник даних. Для кожного класу слід придумати опис розміром в один абзац. Треба описати область застосування класу в рамках даної задачі, вказати всі припущення та обмеження, що стосуються його використання. Словник даних повинен містити опис асоціацій атрибутів, операцій і значень перерахованих типів.

Наприклад, словник даних для системи підтримки банкоматів може містити наступні терміни:

Рахунок – окремий рахунок в банку, з яким виконуються операції. Рахунки можуть бути різних типів. Клієнт може мати декілька рахунків.

Банкомат – термінал, що дозволяє клієнту виконувати операції, використовуючи для ідентифікації кредитні картки. Банкомат взаємодіє з клієнтом, отримуючи від нього дані, відправляючи інформацію про операцію на центральний комп'ютер для її перевірки та обробки, також видає клієнту готівку.

Банк – фінансова установа, що зберігає рахунки клієнтів та видає кредитні картки для доступу до рахунків за допомогою банкоматів.

Клієнт – власник одного або декількох рахунків у банку. Клієнт – це не обов’язково одна особа, це може бути декілька осіб або організація. Одну і ту ж людину, що має рахунок у декількох банках, слід розглядати як декілька клієнтів.

Виділення асоціацій

Структурне відношення між двома та більше класами є асоціацією. Асоціації часто відповідають дієсловам стану або дієслівним групам. До них належать характеристики фізичного розміщення, спрямовані дії, передання інформації та виконання будь-яких умов.

При виділенні асоціації між класами найбільш стандартними є такі:

– клас *A* є фізичною частиною класу *B* («*Крило*» – «*Літак*»);

– клас *A* є логічною частиною класу *B* («*Відрізок дороги*» – «*Маршрут польоту*»);

– клас *A* фізично міститься в класі *B* («*Пасажир*» – «*Літак*»);

– клас *A* логічно міститься в класі *B* («*Політ*» – «*Графік польоту*»);

– клас *A* є описом класу В («*Опис польоту*» – «*Політ*»);

– клас *A* є елементом транзакції класу *B* («*Послуга*» – «*Журнал технічного обслуговування*»);

– клас *A* відомий (записаний, включений) у клас *B* («*Замовлення квитка*» – «*Декларація*»);

– клас А є організаційною одиницею класу *B* («*Служба підтримки*»– «*Літак*»);

– клас *A* використовує або керує класом *B* («*Пілот*» – «*Літак*»);

– клас *A* пов’язаний з транзакцією класу *B* («*Пасажир*» – «*Квиток*»).

Потім необхідно видалити зайві асоціації, для чого можна користуватися такими рекомендаціями:

1. Асоціації між класами, що були видалені на попередніх етапах. Якщо один із класів, зв’язаних асоціацією, був видалений, асоціацію теж потрібно видалити або переформулювати в термінах інших класів.

2. Несуттєві, або асоціації, що відносяться до реалізації. Викиньте всі асоціації, що лежать за межами області завдання або ті, що описують конструкції, які відносяться до реалізації.

3. Дії. Асоціація повинна описувати структурну властивість області додатка, а не короткочасну подію. У деяких випадках вимога виражається у вигляді дії, проте має на увазі деяке структурне відношення. Таке твердження потрібно переформулювати.

Приклад з банкоматом. Асоціація *«Банкомат приймає банківську карту»* описує частину циклу взаємодії банкомата з клієнтом, а не постійне відношення банкомата та карти. З тієї ж причини можна виключити асоціацію *«Банкомат взаємодіє з користувачем»*.

4. Тернарні асоціації. Більшість *n*-арних асоціацій можна виразити через бінарні, додавши відповідні кваліфікатори. Якщо термін у тернарній асоціації є описовим і не має власної індивідуальності, він є атрибутом бінарної асоціації. Наприклад, асоціацію «*Компанія виплачує співробітнику зарплату*» можна переформулювати у вигляді бінарної асоціації «*Компанія наймає співробітника*», причому кожен зв'язок «*Компанія-Співробітник*» буде характеризуватися своїм значенням *salary* (зарплата).

Іноді в додатку дійсно потрібна тернарна асоціація. Наприклад, структуру *«Викладач читає курс лекцій в аудиторії»* не можна розбити на бінарні асоціації без втрати інформації.

5. Похідні асоціації. Відкиньте асоціації, що можуть бути виражені через інші асоціації.

Наприклад, *GrandparentOf (Дідусь)* можна виразити через пару асоціацій *ParentOf (Родитель)*. Викидайте й ті асоціації, що виражаються як обмеження на атрибути Наприклад, *youngerThan (молодший)* виражає умову, що стосується дат народження двох людей, а не якусь додаткову інформацію.

Класи, атрибути та асоціації моделі класів повинні відображати максимально незалежну інформацію. Наявність безлічі

маршрутів між класами часто вказує на похідні асоціації, що можуть бути виражені через деякі примітивні асоціації.

Не всі асоціації, що утворюють множинні маршрути між класами, є надлишковими. Іноді існування асоціації можна вивести з декількох примітивних асоціацій, а от її кратність таким шляхом визначити не можна. У подібній ситуації асоціацію слід зберегти, при умові, що обмеження на кратність важливо для моделі.

Наприклад, на рис.4.1 компанія наймає безліч співробітників і володіє безліччю комп'ютерів. Кожному співробітнику надається нуль і більше комп'ютерів для персонального використання. Деякі комп'ютери призначені для загального користування і не приписані до жодного співробітника. Кратність асоціації «*Приписаний до*» не може бути виведена з кратності асоціацій «*Наймає*» та «*Володіє*».

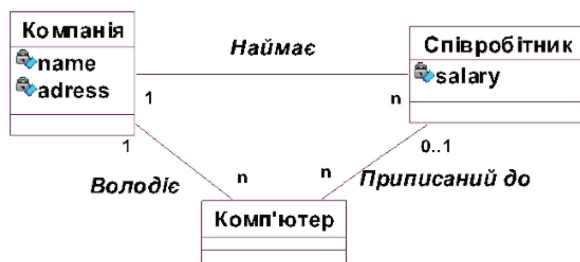

Рисунок 4.1 – Приклад виділення асоціацій

Хоча похідні асоціації не вносять додаткової інформації до моделі, вони можуть бути корисні в реальному світі та при проектуванні Наприклад, відношення родинності типу «дядя», «теща» та «кузен» мають власні назви, тому що вони описують типові відношення, що вважаються досить важливими в нашому суспільстві. Якщо відносини такого роду особливо важливі для моделі, їх можна залишити на діаграмі класів, вказавши перед ім'ям символ навскісної риски (/), що означає залежний статус та дозволяє відрізняти їх від фундаментальних асоціацій.

Далі слід проаналізувати семантику асоціацій:

– назва повинна говорити не про те, як або чому виникла певна ситуація, а про те, в чому ця ситуація полягає. Назви важливі для розуміння моделі в цілому, а тому вибирати їх слід дуже обережно. Наприклад, «*Банківський комп'ютер обслуговує рахунки*» – це твердження, що описує дію. Назву асоціації правильніше буде переформулювати як «*Банк керує рахунком*»;

– назви полюсів асоціацій потрібно вказувати скрізь, де вони мають сенс. Наприклад, в асоціації «*Працює на*» клас «*Компанія»* відносно класу «*Людина*» є роботодавцем, а «*Людина*» відносно класу «*Компанія*» являє собою співробітника. Асоціація між двома примірниками одного і того ж класу вимагає наявності імен полюсів, за допомогою яких можна було б відрізняти ці екземпляри. Наприклад, в асоціації «*Людина керує Людиною*» полюси будуть називатися «*начальник*» та «*підлеглий*»;

– використовуйте кваліфіковані асоціації. Зазвичай назва ідентифікує об'єкт у рамках певного контексту. Більшість назв не є унікальними в масштабах всієї системи (глобально). Контекстразомзім'ямдозволяютьунікальноідентифікуватиоб'єкт. Кваліфікатор дозволяє відрізняти один від одного об'єкти, що перебувають на полюсі асоціації з кратністю «багато»;

– необхідно завжди вказувати кратність асоціації, але не намагайтеся визначити її точно на першому етапі моделювання. Кратність часто змінюється в процесі аналізу. Перевіряйте асоціації з кратністю «один». Наприклад, асоціація «*Один Менеджер керує безліччю Працівників*» не дозволить створити матричну систему управління або описати співробітника, відповідального перед кількома начальниками. Подумайте про необхідність введення кваліфікаторів для асоціацій з кратністю «багато», а також про упорядкування об'єктів;

– додайте всі пропущені асоціації, що вам удасться виявити;

– агрегація важлива для деяких видів додатків, зокрема для опису деталей механізмів і специфікацій матеріалів. Для інших додатків важливість агрегації не настільки велика, і не завжди буває зрозуміло, чи слід її використовувати замість звичайної асоціації. Не витрачайте занадто багато часу на ви-

значення типу асоціації. Виберіть те, що відразу здасться вам більш правильним, і рухайтеся далі.

Наприклад будемо вважати «*Банк*» частиною «*Консорціуму*» іпозначимо відношення між ними як агрегацію.

Приклад з банкоматом. На рис. 4.2 наведена діаграма класів з нанесеними на неї асоціаціями. На ній вказані імена тільки для найбільш важливих асоціацій. На діаграмі зазначені значення кратності. Деякі рішення були довільними. Не варто турбуватися про це: існує безліч коректних моделей завдання.

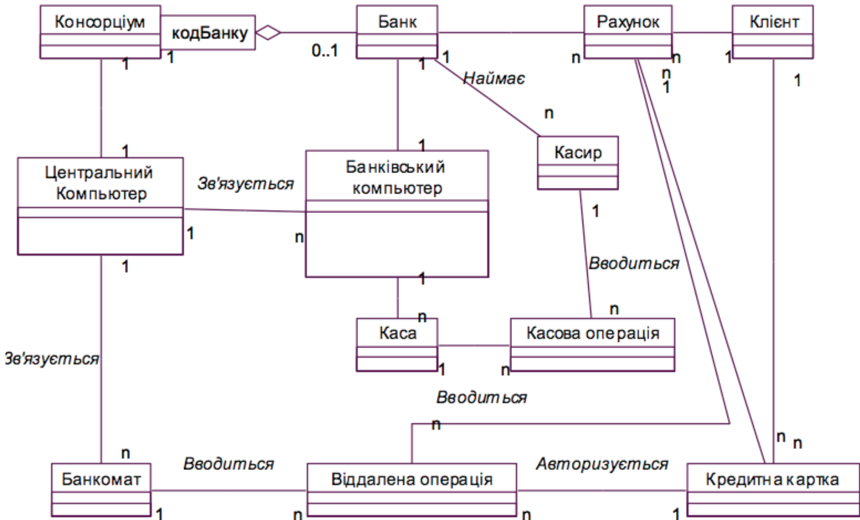

Рисунок 4.2 – Вихідна діаграма класів для банкомата

Виділення атрибутів

Атрибути – це властивості об’єктів, такі як вага, швидкість або колір. Значення атрибутів не повинні бути об’єктами; щоб показати відношення між об’єктами, слід використовувати асоціації.

Атрибути зазвичай наявні в описі завдання у вигляді іменників, що беруть участь в присвійних зворотах, на зразок «колір машини» або «розташування курсора». Прикметники

часто відповідають конкретним значенням атрибутів-перерахувань (наприклад, «червоний», «включено», «застарілий»). На відміну від класів і асоціацій атрибути навряд чи будуть повністю перераховані в описі завдання. Щоб виділити їх, усі доведеться спиратися на своє знання області завдання та реального світу. Атрибути також присутні в артефактах споріднених систем.

При виділенні атрибутів треба розглядати тільки ті з них, що мають безпосереднє відношення до додатка. Спочатку займіться найбільш важливими атрибутами, дрібні деталі можна буде додати в модель пізніше. У процесі аналізу не витрачайте час на деталі реалізації. Обов'язково треба давати всім атрибутам значущі імена.

Зазвичай похідні атрибути включати в модель на цьому етапі не слід. Наприклад, вік можна визначити за датою народження іпоточною датою (остання є властивістю оточення).

Треба також шукати атрибути і для асоціацій. Ці атрибути євластивостями зв'язків між об'єктами, а не властивостями індивідуальних об'єктів. Наприклад, асоціація «багато-до-багатьох» між «*ОдержувачАкцій*» і «*Компанія*» має атрибут «*кількість Акцій*».

Треба виключити непотрібні та некоректні атрибути, коли вони:

– об'єкти. Якщо важливою рисою елемента є незалежне існування, то цей елемент – об'єкт, а не атрибут. Наприклад, «*Начальник*» – це клас, а «*зарплата*» – атрибут. Різниця часто залежить від додатка. Елемент, що володіє власними рисами врамках даного додатка, має моделюватися як клас;

– кваліфікатори. Якщо значення атрибуту залежить від конкретного контексту, його можна спробувати переформулювати у вигляді кваліфікаторів. Наприклад, «*номер Співробітника*»– це не унікальна характеристика людини, зайнятої на двох роботах, це, скоріше, кваліфікатор асоціації «*Компанія наймає співробітника*»;

– імена. Імена краще моделювати як кваліфікатори, а не як атрибути. Дайте відповіді на наступні запитання: чи служить

ім'я для вибору унікального об'єкта з множини? Чи може об'єкт, що належить множині, мати більше одного імені? Якщо відповідь позитивна, ім'я служить кваліфікатором асоціації. Якщо ім'я здається унікальним, можливо, ви забули додати в модель клас, для якого воно служить кваліфікатором.

Наприклад, «*назва Відділу*» може бути унікальною в рамках компанії, але коли-небудь вам доведеться працювати з декількома компаніями. Краще відразу використовувати відповідну кваліфіковану асоціацію. Ім'я є атрибутом у тому випадку, якщо його використання не залежить від контексту, особливо якщо воно не унікальне в деякій множині. Імена людей, на відміну від назв компаній, можуть повторюватися, а тому є атрибутами:

– ідентифікатори. Об'єктно-орієнтовані мови використовують поняття ідентифікатора об'єкта для позначення однозначного посилання на об'єкт. Не слід включати в модель атрибут, єдиним призначенням якого є ідентифікація, тому що ідентифікатори присутні в моделях класів неявним чином. Перераховуйте тільки ті атрибути, що присутні в області додатка. Наприклад, кодРахунку – це справжній атрибут, тому що банк призначає кожному рахунку свій код, а клієнт бачить цей код. Навпаки, внутрішній «*ідентифікатор Транзакції*» не є атрибутом, хоча на етапі реалізації його використання може бути вигідним;

– атрибути асоціацій. Якщо існування значення вимагає існування зв'язку, відповідна властивість є атрибутом асоціації, ане одного зі зв'язаних нею класів. Атрибути зазвичай досить ясно виділяються з асоціацій «багато-до-багатьох»: їх не можна прикріпити до жодного з класів через їх кратності. Наприклад, атрибут «*дата Вступу*» належить асоціації між «*Людиною*» і«*Клубом*», тому що людина може належати до кількох клубів, аклуб може мати кількох членів. Асоціації «один-до-багатьох» деталізувати складніше, тому що тут можна прикріпити атрибут до одного з класів без втрати інформації. Чиніть опір цьому бажанню, тому що якщо кратність зміниться, модель стане некоректною. Ті самі проблеми виникають і з асоціаціями типу «один-до-одного».

– внутрішні значення. Якщо атрибут описує внутрішній стан об'єкта, невидимий зовні, його слід виключити з аналітичної моделі.

– зайві деталі. Виключіть незначні атрибути, які не впливають на більшість операцій.

– нетипові атрибути. Атрибут, що повністю відрізняється від усіх інших і не зв'язаний з ними, може вказувати на те, що клас, до якого він належить, слід розділити на два різні класи. Клас повинен бути простим і цільним. Змішування різних класів – одна з основних причин появи ненадійних моделей. Нечітко визначені класи часто бувають результатом занадто раннього прийняття рішень, що стосуються реалізації.

– логічні атрибути. Розгляньте всі логічні атрибути ще раз. Часто логічний атрибут можна розширити та переформулювати увигляді перерахування.

Реструктурування за допомогою спадкування

Наступний крок: організація класів за допомогою спадкування шляхом виявлення їх загальної структури. Спадкування може бути використано двома способами: як узагальнення однакових аспектів існуючих класів у суперкласах (знизу вгору) і як конкретизація існуючих класів безліччю підкласів (зверху вниз).

Узагальнення знизу вверх. Спадкування можна простежувати знизу вгору шляхом пошуку класів з однаковими атрибутами, асоціаціями та операціями. Для кожного узагальнення слід визначити суперклас, в якому міститимуться загальні риси. Для цього вам, можливо, доведеться перевизначити деякі атрибути або класи. Деякі узагальнення копіюються з реального світу. Скрізь, де це можливо, слід використовувати існуючі поняття. Відсутні класи можна виписувати з міркувань симетрії.

Наприклад, класи «*Віддалена операція*» та «*Касова операція*» подібні одне одному (за винятком ініціалізації) і можуть бути узагальнені класом «*Операція*».

Конкретизація зверху вниз. Конкретизація звичайно слідує зопису області додатка. Шукайте іменні групи, що складаються

зрізних прикметників з ім’ям класу: фіксоване меню, меню, що розкривається та меню, що висувається. Уникайте надмірного уточнення. Якщо запропонована конкретизація несумісна зі існуючим класом, можливо, цей клас просто неправильно сформульований.

Узагальнення та перерахування. Перераховані підкласи області додатка найчастіше потрапляють під визначення конкретизації. Зазвичай буває досить відзначити існування безлічі перерахованих окремих випадків без явної їх вказівки. Наприклад, «*Рахунок*» (у прикладі з банкоматом) може бути рахунком до запитання або ощадним рахунком. У деяких банківських застосуваннях такий розподіл може бути дуже важливим, проте на поведінку банкомата він не впливає, тому «*тип рахунка*» можна зробити просто атрибутом класу «*Рахунок*».

Множинне спадкування. Його краще використовувати лише тоді, коли це дійсно необхідно, оскільки воно підвищує концептуальну і технічну складність моделі.

Подібні асоціації. Якщо назва асоціації з’являється в моделі кілька разів, причому несе вона при цьому однаковий сенс, спробуйте узагальнити асоційовані класи. Інколи в таких класів може не бути нічого спільного, за винятком асоціації, але досить часто ви виявлятимете загальні риси, пропущені на попередніх етапах.

Наприклад, «*Операція*» вводиться як за допомогою класу «*Каса*», так і в класі «*Банкомат*». Клас «*Пристрій Вводу*» узагальнює класи «*Каса*» та «*Банкомат*».

Коректування рівня спадкування. Атрибути й асоціації мають бути присвоєнні конкретним класам з ієрархії. Кожен з них має бути присвоєний найбільш загальному класу, до якого він застосовний. Для цього вам можуть знадобитися деякі коректування. З симетрії може виходити наявність додаткових атрибутів, які дозволять чіткіше відрізняти підкласи один від одного.

На рис. 4.3 показана модель класів банкомата після додавання спадкування.

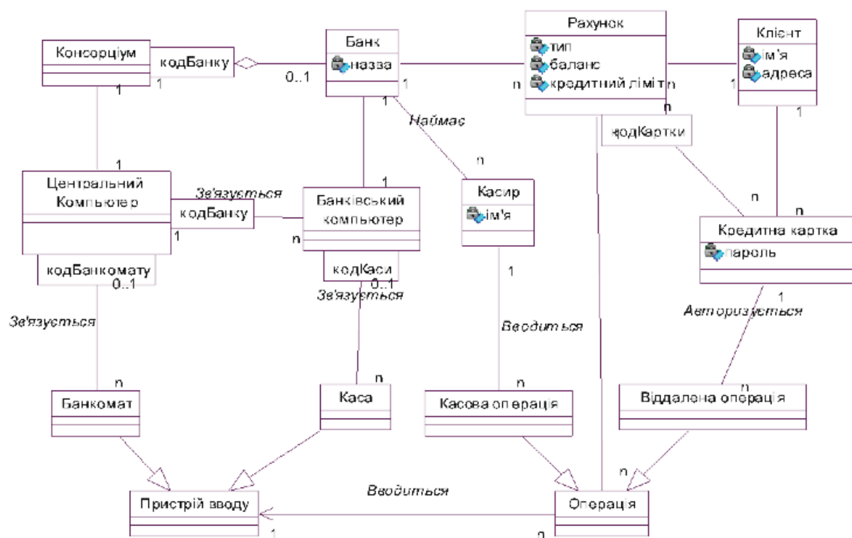

Рисунок 4.3 – Перетворена модель класів банкомата

Перевірка маршрутів

При перевірці маршрутів у моделі необхідно отримати відповіді на такі запитання: Якщо в певному місці передбачається унікальне значення, чи існує маршрут, що дає унікальний результат? Чи передбачений метод вибору унікальних значень об'єктів, охарактеризованих кратністю «багато»? Чи є важливі питання, на які ви не можете відповісти?

Питання без відповіді вказують на інформацію, якої бракує. Якщо щось просте з реального світу в моделі виявилося складним, це означає, що пропущена важлива інформація (проте обов'язково перевірте, чи не властива та ж складність реальному світу).

У моделі можуть бути присутні класи, не зв'язані з іншими. Зазвичай це відбувається тоді, коли відношення між такими класами і останньою моделлю носить розмитий характер. Проте завжди перевіряйте такі класи, тому що ви могли просто пропустити існуючу асоціацію.

Ітераційне розроблення моделі класів

Модель класів рідко буває правильною після першого проходу. Весь процес розроблення ПЗ будується на ітераціях. Різні частини моделі звичайно знаходяться на різних стадіях розробки. Якщо був знайдений недолік, необхідно повернутись на попередню стадію та усунути його. Деякі уточнення можуть бути зроблені тільки після розроблення моделі станів та взаємодії.

Групування класів у пакети

Останній етап моделювання – групування класів у пакети. ***Пакет*** – це група елементів (класів, асоціацій, узагальнень, вкладених пакетів), що характеризуються загальною темою. Пакети роблять модель більш зручною з точки зору конструювання, друку та огляду. Класи одного пакету зв'язані друг з другом міцніше, ніж класи в різних пакетах.

Для того щоб розподілити класи між пакетами потрібно знайти точку з'єднання – класи, що єднають частини моделей, які не мають інших зв'язків. Треба обирати пакети так, щоб зменшувати кількість перетинань на діаграмах класів.

4.2 Модель станів предметної області

Деякі об'єкти предметної області за час свого існування змінюють декілька якісно різних станів. У цих станах вони можуть мати різні обмеження та значення атрибутів, різні асоціації або кратності, виконувати різні операції або мати різну поведінку і т. д. Для таких класів корисно побудувати діаграму станів. Діаграма станів описує різні стани, в яких може знаходитися об'єкт, властивості об'єкта та діючі на них обмеження, атакож події, що викликають перехід об'єкта з одного стану в інший.

Більшість класів предметної області не вимагають використання діаграм станів. Для їх опису досить списку операцій. Модель станів може допомогти в розумінні поведінки тих класів, що можуть перебувати у суттєво різних станах.

Спочатку потрібно виявити класи, які можуть перебувати врізних станах, і записати стан для кожного класу. Потім необхідно визначити події, що викликають перехід кожного об'єкта зодного стану в інший. Знаючи стан і події, ви можете побудувати діаграму станів для кожного з об'єктів. Нарешті, ви повинні перевірити отримані діаграми на повноту і коректність.

Модель станів предметної області конструюється в декілька етапів:

– виявлення класів, що мають різні стани;
– виділення станів;
– виділення подій;
– побудова діаграм станів;
– перевірка діаграм станів.

Виявлення класів з різними станами

На цьому етапі потрібно вивчити перелік класів предметної області, що чітко характеризуються певним життєвим циклом. Треба відшукайте класи, що розвиваються або мають циклічну поведінку, та ідентифікувати значимі стани в життєвому циклі кожного з об'єктів. Наприклад, «*Наукова статтям*» для певного журналу послідовно переходить зі стану *написання* в стан *рецензування*, а потім або в стан *прийнята,* або *відкинута*. Стани статті можуть змінюватися циклічно, наприклад, якщо рецензенти вимагатимуть внесення змін або доповнень, але основна послідовність станів може бути охарактеризована як розвиток.

Деякі стани можуть бути присутніми не в кожному циклі; можуть існувати стани і поза циклом. Бувають класи і з хаотичним життєвим циклом, але більшість класів можуть бути віднесені або до тих, що розвиваються, або до циклічних.

Наприклад, «*Рахунок*» – це важливе поняття з області бізнесу. Поведінка банкомата залежить від стану рахунку. Життєвий цикл рахунку – суміш послідовної та циклічної поведінки. Інші класи предметної області банкомата не мають значимих моделей стану предметної області.

Виділення станів

Наступний крок – треба перерахувати стани для кожного зкласів. Потрібно охарактеризувати об'єкти кожного класу: вказати значення атрибутів, що може мати об'єкт; асоціації, вяких він може брати участь; значення кратності вузлів цих асоціацій; визначити атрибути та асоціації, що мають сенс тільки в певних станах. Надати кожному стану осмислену назву. Назва не повинна описувати, яким чином стан було отримано, вона має позначати сам стан. Стани повинні виділятися на підставі якісних відмінностей поведінки, атрибутів або асоціацій.

Не обов'язково визначати всі стани до виділення подій. Розглянувши події та переходи між станами, можна буде знайти відсутні стани.

Наприклад, клас «*Рахунок*» може перебувати в станах *активний* (звичайний режим доступу), *закритий* (клієнт закрив свій рахунок, але він ще не був виключений з записів банку), *з перевищеним кредитним лімітом* (клієнт перевищив кредитний ліміт за своїм рахунком) і *призупинений* (доступ до рахунка був заблокований з яких-небудь причин).

Виділення подій

Отримавши попередній список станів, треба зайнятися пошуком подій, що викликають переходи між цими станами. Для цього потрібно продумати які зовнішні впливи викликають зміни станів. У багатьох ситуаціях подію можна розглядати як завершення поточної діяльності. Наприклад, якщо стаття перебуває в стані *Рецензування*, перехід з цього стану здійснюється після завершення роботи рецензента. Рішення може бути позитивним (перехід у стан *прийнята*) чи негативним (перехід устан *відкинута*). Завершення діяльності може викликати альтернативні переходи, які можуть додаватися в процесі вдосконалення моделі. Наприклад, стаття може перейти в стан *Прийнята* зобов'язковими змінами.

Інші події можна побачити, подумавши про те, яким чином об'єкт може потрапити в певний стан.

Всередині стану можуть відбуватися і події, що не викликають переходів. Для моделі станів предметної області важливі тільки ті події, що викликають переходи. Інформацію, що міститься в події, слід подавати у формі списку його параметрів.

Наприклад перерахуємо найважливіші події для класу «*Рахунок*»: *закриття рахунка, перевищення кредитного ліміту, повторне неправильне введення PIN-коду, можлива підробка* та *адміністративні дії*.

Побудова діаграм станів

Наступний крок – розподіл подій за станами, до яких вони належать, додавання переходів, що показують зміни станів, викликані здійсненням події в той час, коли об'єкт знаходиться впевному стані. Якщо подія завершує стан, з цього стану зазвичай буває тільки один перехід в інший стан. Якщо подія ініціює цільовий стан, то потрібно розглянути, в яких станах ця подія може відбуватися, і додати на діаграму переходи з цих станів до цільового стану. Якщо подія має різну дію на різні стани, потрібно додати переходи для кожного з цих станів.

Якщо ж об'єкти класів виконують діяльність при переходах, то треба додати цю діяльність на діаграму.

Наприклад, для класу «*Рахунок*» була створена діаграма станів, наведена на рисунку 5.4.

Перевірка діаграм станів

Останній крок – перевірка кожної моделі станів. Чи всі стани досяжні? Особливу увагу потрібно приділити маршрутам по діаграмі. Якщо діаграма описує клас з поведінкою, що розвивається, чи є на діаграмі маршрут, що з'єднує початковий стан зкінцевим? Чи присутні на ній очікувані відхилення від основної послідовності? Якщо діаграма описує клас з циклічною поведінкою, чи є на ній основна петля циклу? Чи є тупикові стани, що завершують цикл?

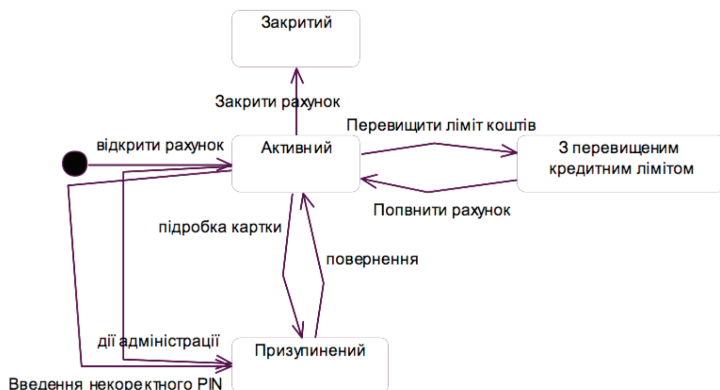

Рисунок 4.4 – Модель станів класу «Рахунок»

Іноді відсутність маршрутів вказує на відсутність потрібних станів. Готова модель станів повинна точно описувати життєвий цикл класу.

4.4 Практичні завдання

Завдання 1. Розробіть словник предметної області. Створить таблицю, до якій необхідно занести всі іменники. Виконайте її класифікацію. Підготуйте документ Glossary.

Для наступних систем:

1. Система «*Бібліотека*». До системи ставляться такі вимоги: це незалежна бібліотека, бібліотека складається здекількох відділів абонементів та читальних залів, бібліотека працює лише з книжками, читач може отримати книжку на абонементі на декілька днів або читати її в читальному залі. Старі книжки можливо замовляти в архіві, з якого вона поступає продовж одного дня, та бібліотекар відсилає читачу відповідне sms повідомлення. Є окрема категорія коштовних книжок, за використання яких потрібно платити. Оплату можливо виконувати як готівкою так і з використанням карток. Якщо користувач не повернув книжку вчасно за її використання нараховується пеня. Реєстрація та щорічна перереєстрація у бібліотеці також

платна. Системою користуються тільки співробітники бібліотеки.

2. Система «*Інтернет-магазин*», для якої необхідно виконання таких вимог: це магазин, що торгує одним видом товарів (наприклад книги, диски, продукти та інші), покупець може ознайомитись з каталогом продукції, замовити товар, замовити товар також якщо він відсутній, оплата виконується через картки VISA, Maestro або через банківський переказ, після поставки оформлюється відповідний акт, для оформлення продажу покупець повинен бути зареєстрований. Можливо внести передоплату якщо товар у магазині відсутній.

3. Система «*Дистанційне навчання*», для якої необхідне виконання таких вимог: студент повинний обрати спеціальність, для кожної спеціальності є перелік обов'язкових дисциплін та кредити, що залишились, можна обрати від 1 до 5 дисциплін на кожен рік навчання, головне щоб були використані всі кредити, навчання продовжується 4 роки, навчання можна переривати, але повертатися можна лише на курс, перед яким немає заборгованостей, заняття складаються з лекцій, практичних робіт та електронного тестування. Дисципліна вважається завершеною, якщо виконанні всі види робіт, за один семестр екзамен з дисципліни можливо складати не більше 3 разів за семестр. Диплом видається після успішного закінчення всіх років навчання. Сплата за навчання здійснюється через картки *VISA*, *Maestro* або через банківський переказ. Розглянути тільки частину системи, де користувач тільки студент.

Завдання 2. Розробіть діаграму класів для заданої предметної області. Діаграма повинна містити відношення агрегації (або композиції) та узагальнення. Розбийте діаграму класів на пакети.

Завдання 3. Розробіть діаграми станів для класів:

– Системи «Бібліотека» – «Читач», «Книжка», «Читацький абонемент».

– Системи «Інтернет магазин» – «Рахунок», «Товар», «Покупець».

– Системи «Дистанційне навчання» – «Студент», «Дисципліна», «Робота».

Завдання 4. Для діаграми класів 4.3 випишіть всі можливі запроси. Покажіть як її можна використовувати для кожного запросу.

Завдання 5. Для системи Інтернет магазин наведений перелік можливих асоціацій та узагальнень. Підготуйте перелік асоціацій, що необхідо виділити або переіменувати. Обгрунтуйте своє рішення. Потенційні асоціації:

– користувач замовляє товар;
– користувач сплачує послуги;
– адміністратор редагує інформацію про товар;
– адміністратор оновлює інформацію про товар;
– користувач вносить передоплату;
– користувач оформлює бланк замовлення;
– користувач заповнює бланк замовлення товару;
– система відстежує кількість одиниць кожного товару, і якщо товар закінчується відсилає відповідне повідомлення адміністратору.

4.5 Контрольні запитання

1. Що містить аналітична модель?
2. Що відображає модель предметної області?
3. Які підходи використовуються для виявлення класів?
4. Який перелік для класів запропонував Барамі?
5. Чим відрізняється підхід на основі іменних груп від підходу на основі використання прецедентів?
6. Які методи містить підхід CRC?
7. За якими принципами слід знищувати зайві асоціації?
8. Що містить словник даних?
9. Назвіть найбільш стандартні асоціації.

10. За якими правилами слід додавати ти виключати атрибути?
11. За якими двома способами може бути визначено спадкування?
12. Для чого використовують перевірку маршрутів?
13. Як будується модель станів предметної області?

4.6 Література до розділу

1. Mishra J. Software Engineering. [Текст] / J. Mishra, A. Mohanty. – Dorling Kindersley (India), 2012. – 373 p.
2. 20. Schmidt, D.C. Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects / D.C.Schmidt,M. Stal,Hans.–USA:JohnWilye&Sons,2013.–450p.
3. Griffiths, D. Organizational Learning and Knowledge: Concepts, Methodologies, Tools and Applications – Concepts, Methodologies, Tools and Applications, Volumes 1-4 / D. Griffiths, S. Koukpaki. – USA.: Management Association, Information Resources, 2011. – 3164 p.
4. Буч Г. Язык UML. Руководство пользователя [Текст] / Буч Г. – М.: ДМК Пресс; СПб.: Питер, 2004. – 432 с.
5. Эванс Э. Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. [Текст] / Эванс Э. – М.:Вильямс, 2010. – 448 с.
6. Шитікова О.В. Формалізація процесу випробувань газотурбінних установок наземного використання / Шитікова О.В., Табунщик Г.В. // Тиждень науки : тези доповідей науково-практичної конференції, м. Запоріжжя, 13–19 квітня 2012 р./ редкол.: Ю. М. Внуков(відпов. ред.) та ін. – Запоріжжя: ЗНТУ, 2012. -С. .332 – 333.
7. Шитикова Е.В. Информационная модель процесса испытаний газотурбинных установок наземного применения [Текст] / Е.В. Шитикова, Г.В. Табунщик // Радіоелектроніка, інформатика, управління. – 2013. – № 1(28). – С. 101-108. http://ric.zntu.edu.ua/issue/view/777

5 МОДЕЛЮВАННЯ ПОВЕДІНКИ СИСТЕМИ

Поведінка системи – те як вона виглядає для зовнішнього користувача – зображується у вигляді ***прецедентів***. Моделі прецедентів можна розробляти на різних рівнях абстракції. Їх можна застосувати до системи в цілому для того, щоб специфікувати основні функціональні блоки програмного додатка, що розробляється. Їх також можна використати для фіксації поводження пакетів *UML*, частин пакетів або навіть *класу* усередині пакету.

На етапі *аналізу* прецеденти вбирають у себе системні вимоги, концентруючись на тому, *що* робить або повинна робити система. На етапі *проектування* подання проектних рішень у вигляді прецедентів можна використати для специфікації поводження системи в тому вигляді, як воно повинне бути реалізоване.

Поводження системи, закріплене за допомогою прецедентів, вимагає здійснити відповідні обчислення й забезпечити взаємодію об'єктів для виконання цих прецедентів. *Обчислення* можна змоделювати за допомогою діаграм видів діяльності. *Взаємодію* об'єктів можна задати за допомогою діаграм послідовностей або діаграм кооперації.

Моделювання поведінки дозволяє подивитися на систему з *погляду її функціонування*. Тут основне завдання полягає в тому, щоб визначити прецеденти для області додатків і встановити, які класи беруть участь у виконанні цих прецедентів. При цьому необхідно ідентифікувати операції класів і повідомлення, передані між об'єктами. Хоча взаємодія об'єктів ініціює зміни стану об'єктів, моделювання поведінки дає функціональний погляд на миттєвий стан системи.

Моделі прецедентів повинні розроблятися ітеративно та паралельно з моделями класів. У ході створення моделей поведінки з'являються ще два рівні класів:

– класи, що обслуговують події, які ініціюються користувачами, та являють собою бізнес-процеси (*керуючі класи*);

– класи, що являють собою *GUI*-інтерфейси (*прикордонні класи*).

5.1 Моделювання взаємодії

Модель взаємодії для додатка будується в декілька етапів:

– виконати моделювання прецедентів;

– побудувати діаграми діяльності для складних прецедентів;

– виконати моделювання взаємодії.

Моделювання прецедентів

Моделювання прецедентів тісно пов’язане з установленням вимог. Вимоги, викладені в текстовому вигляді в документі опису вимог, необхідно довести до прецедентів, зафіксованих у документі специфікації вимог. Якщо подальший процес розроблення керується прецедентами, то процес називається проблемно-орієнтованим.

Подібно до моделювання класів моделювання прецедентів істотно ітеративний та нарощуваний процес. Первісну діаграму прецедентів можна визначити на основі вимог верхнього рівня. Це може бути модель бізнес-прецедентів. Для подальшого уточнення прецедентів варто керуватися більш деталізованими вимогами. Якщо протягом ЖЦ розробки вимоги користувачів піддаються змінам, ці зміни варто спершу занести до документа опису вимог, а вже потім – до моделі прецедентів. Потім зміни в прецедентах доводять до інших моделей.

Ідентифікація дійових осіб. По-перше, необхідно ідентифікувати *зовнішні об’єкти*, що безпосередньо взаємодіють із системою. Це і будуть дійові особи. До їх числа належать люди, зовнішні пристрої та інші програмні системи. Найважливішою властивістю дійових осіб є те, що вони не контролюються програмно, а їх поведінка повинна вважатися непередбачуваною. Попри те, що може існувати якась очікувана послідовність дій дійової особи, система повинна бути досить стійкою, щоб витримувати порушення цієї послідовності.

У процесі пошуку дійових осіб цікавлять не індивідуальні сутності, а з архетипічною поведінкою. Кожна дійова особа

– це ідеалізований користувач, який використовує будь-яку підмножину функціональності системи. Необхідно вивчити кожний зовнішній об’єкт і з’ясувати, чи не характеризується він істотно різними видами поведінки. Дійова особа – це один варіант поведінки відносно до системи, а один зовнішній об’єкт може відповідати кільком дійовим особам. З іншого боку, різні типи зовнішніх об’єктів можуть описуватися однією дійовою особою.

Наприклад, для системи обслуговування клієнтів банку одна людина може одночасно бути і касиром, і клієнтом одного і того ж банку. Це цікавий збіг, який, проте, частіше за все не є важливим. Відносно банку людина завжди виступає або в одній, або в іншій ролі. Для додатка для банкомата дійовими особами будуть клієнт, банк і консорціум.

Ідентифікація прецедентів. При виявленні прецедентів аналітик повинен переконатися в тому, що він твердо дотримується сутності концепції прецедентів. Прецеденти являють собою такі компоненти загальної моделі системи:

– завершений фрагмент функціональних можливостей (включаючи основний потік логіки керування, його будь-які варіації (підпотоки) та виняткові умови (альтернативні потоки));

– фрагмент із зовні спостережуваних функцій (відмінних від внутрішніх функцій);

– ортогональний фрагмент функціональних можливостей (прецеденти можуть при виконанні спільно використати об’єкти, але виконання кожного прецеденту незалежно від інших прецедентів);

– фрагмент функціональних можливостей, що ініціюється суб’єктом (будучи ініційований, прецедент може взаємодіяти з іншими суб’єктами). При цьому можливо, що суб’єкт виявиться тільки на приймальному кінці прецеденту (може бути опосередковано), ініційованого іншим суб’єктом;

– фрагмент функціональних можливостей, що надає суб’єктові відчутного корисного результату (і цей корисний результат досягається в межах одного прецеденту).

Виявлення прецедентів базується на аналізі наступних джерел інформації:

– вимоги, що визначені в документі опису вимог;

– суб’єктів і їхніх цілей стосовно до системи.

У ході аналізу прецеденти звертаються до особистісних потреб суб’єктів. Будь-яка поведінка системи має належати певному прецеденту. Іноді можуть виникнути проблеми з віднесенням якої-небудь поведінки, близької до межі, до того чи іншого прецеденту. При розбитті завжди слід пам’ятати про те, що завжди існують прикордонні варіанти. У цьому випадку рішення можна прийняти довільно.

Кожний прецедент має відображати якийсь сервіс (послугу), що надається системою, тобто щось цінне для діючої особи. Всі прецеденти слід розглядати на одному рівні деталізації. Наприклад, якщо один з варіантів використання для банку називається *«Запросити позику»*, то інший варіант не повинен називатися *«Зняти готівку з депозитного рахунка через банкомат»*. Друга назва занадто деталізована порівняно з першим. Краще назвати цей варіант просто «*Зняти гроші*».

Отже, для системи роботи з банкоматом можемо виділити наступні прецеденти:

– *«Ініціалізація сеансу»* – банкомат визначає особу користувача і пропонує йому список дій;

– *«Запитування рахунка»* – система надає загальні відомості про рахунок, такі як поточний баланс, дату останньої операції, дату відправлення останнього звіту поштою;

– *«Оброблення операції»* – система банкомата виконує дії, що впливають на баланс рахунка, такі як внесок, зняття та переведення коштів. Банкомат гарантує, що всі завершення операцій в кінцевому підсумку записуються в базу даних банку;

– *«Передання даних»* – банкомат використовує можливості консорціуму для взаємодії з комп’ютерами відповідного банку.

Ідентифікація початкових і кінцевих подій. Прецеденти розбивають функціональність системи на дискретні частини та показують діючі особи, які використовують кожну з цих частин. Проте поведінка системи демонструється в них недостатньо чіт-

ко. Для розуміння поведінки необхідно знати послідовності виконання, відповідні кожному з прецедентів. Починати їх аналіз слід з пошуку подій, що ініціюють кожний з прецедентів. Треба визначити, яка особа ініціює варіант використання, і подію, яку вона для цього передає системі. У багатьох випадках початковою подією є запит певної послуги, що надається системою. В інших випадках початковою подією є подія (пригода), що запускає ланцюжок дій. Дайте цій події осмислену назву, але поки не намагайтеся визначити повний список її параметрів.

Крім того, слід визначити кінцеву подію або групу подій, а також загальні рамки подій, що повинні бути включені в кожний з прецедентів. Наприклад, прецедент «Запит позики» може тривати до тих пір, поки прохання не буде подано, поки позика не буде видана чи відхилена або поки позика не буде погашена та закрита. Будь-який з цих варіантів цілком прийнятний. Розробник моделі повинен визначити межі прецедента, встановивши його кінцеву подію.

Підготовка типових сценаріїв. Для кожного прецеденту потрібно підготувати один або кілька типових сценаріїв, щоб відчути очікувану поведінку системи. Ці сценарії описують основні взаємодії, формати зовнішніх відображуваних даних, а також обмін інформацією. Сценарієм називається послідовність подій на множині взаємодіючих об'єктів. Аналіз варто здійснювати в термінах прикладів взаємодії, а не розписувати найбільш загальні варіанти.

Для більшості завдань логічна коректність залежить від взаємної послідовності взаємодій, а не від конкретних часових проміжків між ними.

Іноді опис завдання повністю задає послідовність взаємодії, але в більшості випадків її доведеться вигадувати (або, принаймні, конкретизувати). Наприклад, у постановці задачі про банкомат ідеться про необхідність отримання даних про транзакції від користувача, але не вказується, які конкретно параметри потрібно в нього запитати і в якій послідовності.

Для більшості додатків порядок введення вихідних даних не має особливої важливості та може бути відкладений до етапу проектування.

Підготуйте сценарії для типових ситуацій – взаємодій без незвичайних вхідних параметрів і помилкових ситуацій. Подією є обмін інформацією між об'єктом системи та зовнішнім агентом. Параметром події є передана інформація. Наприклад, параметром події «введений пароль» є сам введений пароль. Події без параметрів теж мають значення і досить поширені. Саме здійснення події є інформацією. Для кожної події необхідно вказати особу, що викликала її (систему, користувача або іншого зовнішнього агента) і параметри події.

Наприклад, для прецеденту «*Ініціалізація сеансу роботи з банкоматом*» можна описати такий сценарій:

Банкомат запрошує користувача вставити кредитну картку.

Користувач вставляє кредитну картку.

Банкомат приймає картку і зчитує її серійний номер.

Банкомат запитує пароль. Користувач вводить «1234».

Банкомат перевіряє пароль, зв'язуючись із консорціумом і банком.

Банкомат виводить меню дій над рахунками та команд.

Для прецеденту «Оброблення операції» основний сценарій буде таким:

Банкомат виводить меню рахунків та команд.

Користувач обирає зняття грошей з рахунка.

Банкомат запитує суму, що знімається.

Користувач вводить суму.

Банкомат перевіряє суму на перевищення ліміту видачі готівки.

Банкомат зв'язується з консорціумом та банком для перевірки, чи достатньо коштів на рахунку.

Банкомат видає готівку та просить користувача її забрати.

Користувач бере готівку.

Банкомат виводить меню дій над рахунками і команд.

Нетипові сценарії та виняткові ситуації. Після розробки типових сценаріїв необхідно розглянути особливі ситуації, такі як відсутність введених значень, введення мінімального та максимального значень, введення однакових значень. Потім потрібно розглянути помилкові ситуації, включаючи введення неправильних значень і відсутність відгуку. Для багатьох інтерактивних додатків оброблення помилок є найбільш складною

частиною процесу розроблення. Необхідно надавати користувачу можливість відмінити операцію або відкотитися до чітко визначеної початкової точки кожного етапу. Нарешті, потрібно розглянути додаткові взаємодії, що можуть перетинатися з базовими (такі як звернення до довідкової системи або запит відомостей про стан).

Наприклад, для прецеденту «*Ініціалізація сеансу роботи з банкоматом*» можна виділити такі нетипові сценарії та виняткові ситуації:

Банкомат не може прочитати картку.

Термін дії картки закінчився.

Тайм-аут при очікуванні банкоматом відповіді клієнта.

Лінії зв'язку не працюють.

Для прецеденту «*Оброблення операцій*» :

Введена сума перевищує ліміт видачі готівки

Введена сума перевищує суму коштів на рахунку.

Додаткові сценарії описують роботу адміністративних частин системи банкомата, наприклад авторизацію нових карток.

Структурування дійових осіб і прецедентів. На даному етапі потрібно впорядкувати прецеденти за допомогою відносин включення (*<<include>>*), розширення (*<<extend>>*) та узагальнення. Це особливо корисно для великих і складних систем. Як і з моделями класів і станів, структурування краще відкласти до тих пір, поки не будуть сформульовані всі базові прецеденти. Якщо виконати структурування занадто рано, існує небезпека спотворення додатка через фіксування підсвідомих міркувань у структурі прецедентів.

Узагальнення можна застосовувати і до дійових осіб. Наприклад, адміністратора можна вважати оператором, що має додаткові привілеї.

Наприклад, на рис. 5.1 наведена діаграма прецедентів, що узагальнено зображає роботу системи банківського обслуговування через банкомати.

Перевірка щодо моделі класів предметної області. На цьому етапі модель предметної області та модель додатку повинні бути вже добре узгоджені одна з одною. Дійові особи,

прецеденти та сценарії засновані на класах і концепціях моделі предметної області. Згадайте, що одним з етапів побудови цієї моделі є перевірка маршрутів. На практиці ця перевірка – перший крок до виділення прецедентів.

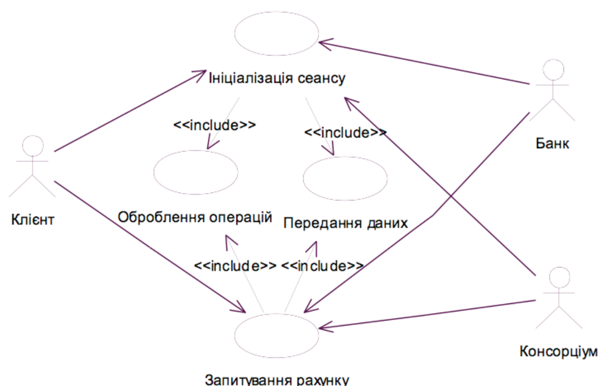

Рисунок 5.1 – Діаграма прецедентів

Зіставте модель додатка та модель предметної області, щоб переконатися в їх узгодженості. Вивчіть сценарії і перевірте, що в моделі предметної області є всі необхідні дані. Переконайтеся, що ця модель містить всі параметри подій.

<u>Моделювання зовнішніх подій</u>

Проаналізуйте всі розроблені сценарії та виділіть всі зовнішні події: введення даних, прийняття рішень, переривання та взаємодії з іншими користувачами та зовнішніми пристроями. Подія може викликати дії цільового об’єкта. Етапи внутрішніх обчислень не є подіями, за винятком розрахунків, у ході яких здійснюється взаємодія із зовнішнім світом. За допомогою сценаріїв ви можете відшукати типові події, але не забудьте і про нетипові події та помилкові ситуації.

Передання інформації об’єкту є подією. Наприклад, «*введений пароль*» – це повідомлення, передане від зовнішнього агента «*Користувач*» об’єкту програми «*Банкомат*». Деякі потоки інформації наявні в моделі неявним чином. Багато подій характеризуються певними параметрами.

Згрупуйте під однаковою назвою події, що надають однаковий вплив на потік управління, навіть якщо значення їх параметрів відрізняються. Наприклад, «*введений пароль*» – це подія, параметром якого є значення пароля. Вибір значення пароля не впливає на потік управління, тому події з різними паролями є екземплярами одного і того ж типу подій. Аналогічним чином, «*видача готівки*» також є подією незалежно від суми (параметра). Екземпляри подій, значення яких впливають на потік управління, повинні бути віднесені до різних типів подій.

Треба стежити за ситуаціями, коли розбіжності кількісних значень досить істотні для того, щоб події можна було вважати різними. Наприклад, натискання будь-якої цифрової клавіші на клавіатурі можна вважати подією, що не залежить від конкретної цифри, тоді як натискання клавіші «введення» можна розглядати окремо, тому що програма буде обробляти його не так, як натискання на цифрові клавіші. Розрізнення подій залежить від додатка.

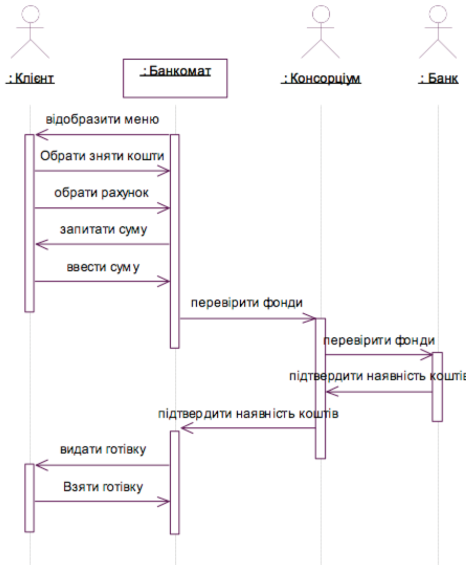

Рисунок 5.2 – Діаграма послідовності для прецеденту «Оброблення операції» для основного сценарію

Підготуйте діаграму послідовності для кожного сценарію. Діаграма послідовності показує учасників взаємодії та послідовність повідомлень, якими вони обмінюються. Кожному учаснику виділяється свій стовпець. Діаграма показує відправника та одержувача кожного повідомлення. Якщо в сценарії бере участь кілька об'єктів одного і того ж класу, їм слід присвоїти різні номери. Вивчивши один стовпчик таблиці, ви можете визначити події, що безпосередньо впливають на конкретний об'єкт. Після цього ви можете згрупувати події, що відправляються і приймаються кожним класом.

Для кожного альтернативного потоку повинна існувати своя діаграма взаємодії.

Приклад з банкоматом. На рис. 5.2 показана діаграма послідовності для головного сценарію «Оброблення операції», на рис. 5.3 – діаграма для альтернативного потоку.

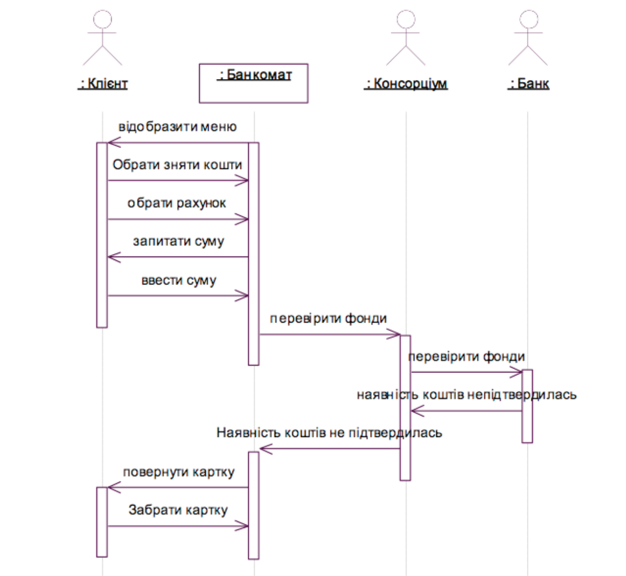

Рисунок 5.3 – Діаграма послідовності для альтернативного потоку прецеденту «Оброблення операції»

Діаграми послідовності описують діалог і взаємодію дійо-

вих осіб, але на них не можна відобразити наявні альтернативи й прийняті рішення. Для цього використовують окрему діаграму для основного потоку взаємодії та окремі діаграми для кожної помилкової ситуації і кожної точки прийняття рішення. Діаграми діяльності дозволяють об'єднати всю цю поведінку завдяки документуванню розгалужень і злиттів (поєднань) потоку управління.

Підготовка діаграм діяльності для складних прецедентів

Подібно традиційним потоковим діаграмам і структурним схемам, що одержали поширення в рамках структурних методів для розроблення процедурно-орієнтованих програм, діаграми видів діяльності являють собою потік логіки керування в об'єктно-орієнтованих програмах. Розходження ж полягає в можливості подання за допомогою діаграм видів діяльності керування паралельними потоками поряд з послідовним керуванням.

Моделі видів діяльності широко використаються в проектуванні.

Однак вони можуть бути особливо корисні для визначення потоків видів діяльності в процесі виконання прецедентів. Оскільки моделі видів діяльності не відображають об'єктів, що здійснюють діяльність, граф видів діяльності можна побудувати навіть у тому випадку, коли модель класів відсутня або розробляється. Зрештою кожний вид діяльності визначається однією або декількома операціями в одному або декількох класах, що кооперуються. Детальне опрацювання такої кооперації може бути здійснене з використанням діаграм кооперації.

Кожний прецедент можна моделювати за допомогою одного або декількох графів видів діяльності. Подія, джерелом якої служить суб'єкт – ініціатор прецеденту, ця та ж сама подія, що запускає виконання графа видів діяльності. Процес виконання послідовно переходить від одного стану виду діяльності до іншого. Стан виду діяльності вважається завершеним, коли завершується його обчислення. Зовнішні переривання, що ініціюються подіями, які можуть викликати завершення

стану виду діяльності, допускаються тільки у виняткових випадках. Якщо очікується, що подібні події можуть відбуватися часто, то треба замість цього скористатися діаграмою станів.

Види діяльності найкраще виявляти на основі аналізу пропозицій неформальної специфікації прецедентів. Кожна фраза, що містить дієслово, може розглядатися як потенційний вид діяльності. Опис альтернативних потоків уводить у граф видів діяльності розгалуження й поділ потоків. Вони приводять до виняткових станів діяльності. Можливі також паралельні потоки керування.

Після виявлення станів видів діяльності специфікація видів діяльності виглядає як досить простий процес з'єднання цих станів лініями переходів. Паралельні потоки ініціюються (розділяються) і зливаються, альтернативні потоки розгалужуються та поєднуються, що відображається у вигляді ромбів розгалуження.

Зовнішні події на графі видів діяльності звичайно відсутні. Однак існує графічний метод включення зовнішніх подій у граф. Аналогічно, існують графічні позначення для станів потоків об'єктів для подання об'єктів, які є вхідними або вихідними для годиться діяльності.

На рисунку 5.4 наведена діаграма діяльності для перевірки картки.

Рисунок 5.4 – Діаграма діяльності для перевірки картки

5.2 Практичні завдання

Завдання 1. Сформулюйте властивості та визначте прецеденти для наступних систем:

1.1 Система «*Електронна бібліотека*». Для системи повинні виконуватись такі вимоги: це незалежна бібліотека, бібліотека працює лиш з електронними журналами, користувачі бібліотеки – модератори та читачі. Читачі повинні сплатити реєстрацію за рік – за використання чи певного журналу чи всієї бібліотеки. Користувач може отримати електронну копію або проглядати електрону версію on-line статті абу журналу якщо в нього є передплата. Або придбати додатково якщо раніше передплати не було. Модератор оновлює електронні версії журналів, обробляє повідомлення від читачів, виконує розсилку коли додається новий журнал, перевіряє інформацію від читачів про реєстрацію та платежі.

1.2 Система «*Склад*». Для системи повинні виконуватись такі вимоги: товар, що зберігається має назву, розрізняється за видом, класом та типом. У кожного товару є унікальний код, вартість за яку був придбаний, остання дата оновлення, кількість одиниць, місце зберігання. Товар прибуває до складу та відбуває зі складу. Кожний вид товару може обслуговувати тільки визначений комірник, у якого є унікальний код у системі. Менеджер може переглянути кількість одиниць товару, відповідального комірника, замовити товар у постачальника якщо товар відсутній, відмовити у постачанні товару, якщо він відсутній. При отриманні товару оформлюється відповідна накладна, що містить кількість одиниць товару, постачальник, вартість одиниці, комірник що отримав, експедитор, що привіз. Виконується оплата товару через банківський переказ. Якщо фактична кількість та кількість у накладній відрізняється, то робиться додатковий заказ. Інформація про товар оновлюється. Накладні зберігаються у загальному реєстрі. Відправлення товару зі складу відбувається відповідно до бланку замовлення. До клієнта можуть бути відправлені декілька видів товарів, що відображається у замовленні. Якщо на складі відсутні необхідна кількість товару то оформлюєть-

ся додаткове замовлення. Відправлення виконується тільки після попередньої оплати товару через банківський переказ. Інформація про товар оновлюється. Замовлення зберігаються у загальному реєстрі.

Завдання 2. Для вищезгаданих систем створіть прецеденти та відповідні документи (*Use Case Document*) з використанням *Rational RequisitePro*, намалюйте діаграми прецедентів.

Завдання 3. Для кожного прецедента з попереднього завдання розробіть альтернативні потоки для обробки помилок.

Завдання 4. Розгляньте як зміниться діаграма класів та прецедентів якщо для системи *«Електронна бібліотека»* додати такі вимоги:

– система може працювати з електронними книжками;

– книжки можна читати тільки в електронному вигляді з сайту бібліотеки.

Завдання 5. Створіть діаграми взаємодії для кожного основного сценарію та альтернативних потоків.

Завдання 6. Створіть діаграми діяльності для вишевизначених прецедентів.

5.3 Контрольні запитання

1. Яке призначення прецедентів?
2. Які класи з'являються як результат розробки моделі взаємодії?
3. Які етапи потрібно виконати для будування моделі взаємодії?
4. Які компоненти загальної моделі системи являють собою прецеденти?
5. Що відображає прецедент?
6. Які типи сценаріїв використовуються при описі прецедентів?
7. Чи можливо на діаграмах взаємодії одночасно зобразити основний та альтернативний сценарій?
8. Як виконується моделювання зовнішніх подій?

9. Коли рекомендується використовувати діаграми схем діяльності?

5.4 Література до розділу

1. Шитикова Е. В. Модельно-ориентированные методы автоматизации процесса испытаний сложных технических систем / Шитикова Е. В. Табунщик Г. В. // Электротехнические и компьютерные системы № 22 (98), 2016, - С. 338 – 346 http://dx.doi.org/10.15276/eltecs.22.98.2016.61
2. Tabunshchyk G. Flexible Technologies for Smart Campus/ D. Van Merode, G. Tabunshchyk, K. Patrakhalko, Y. Goncharov // Proceedings of XIII International Conference on Remote Engineering and Virtual Instrumentation (REV2016) (24-26 February, 2016, Madrid, Spain) UNED: pp. 58-62.
3. А.С. № 66615 Система керування контентом для віддалених експериментів з дослідження надійності вбудованих систем /Табунщик Г.В., Охмак В.О., опубл. 13.07.2016
4. Tabunshchyk G. Multipurpose Educational System based on Raspberry Pi/ G. Tabunshchyk, D. Van Merode , O.Petrova, V. Okhmak // Proceedings of the International Symposium on Embedded Systems and Trends in Teaching Engineering, Nitra, Slovakia, 12-15 September, 2016 – pp. 202-206
5. Tabunshchyk G. SMART-CAMPUS INFRASTRUCTURE DEVELOPMENT BASED ON BLE 4.0/ G. Tabunshchyk, PhD., D. Van Merode, Y. Goncharov, K. Patrakhalko // Journal Electrotechnic and Computer Systems No. 18 (94), 2015 17 – 20
6. Arras, P. Iterative Pattern for the Embedding of Remote Laboratories in the Educational Process / P. Arras, K. Henke, G.Tabunshchyk, D. V.Merode // 12th International Conference on Remote Engineering and Virtual Instrumentation (REV 2015) 25-28 February 2015, Bangkok, Thailand, PP. 52-55 (10.1109/REV.2015.7087262)

6 МОДЕЛЬ ПРОЕКТУВАННЯ

6.1 Модель класів додатку

Класи додатку описують сам додаток, а не об'єкти реального світу, з якими він працює. Більшість класів додатку пов'язані з комп'ютерами та визначають сприйняття додатку користувачами. Модель класів додатку будується в кілька етапів:

– вказати інтерфейси користувача;

– визначити прикордонні класи та керуючі об'єкти;

– перевірити модель класів на відповідність моделі взаємодії.

Визначення інтерфейсів користувача

Більшість взаємодій можна розділити на дві частини: логіка додатку та інтерфейс користувача. *Інтерфейсом користувача* (*user interface*) називається об'єкт або група об'єктів, що надають користувачеві системи єдину точку доступу до об'єктів предметної області, командам і параметрам додатку. У процесі аналізу увага приділяється насамперед потокам інформації та керування, а не формату уяви. Одна й та сама програмна логіка може приймати вхідні дані з командного рядка, зчитувати з файлу, інтерпретувати як натискання кнопки миші, натискання на сенсорні панелі та кнопки, так приймати і віддалені сигнали, що надходять, за умови, що зовнішній інтерфейс акуратно відділений від внутрішньої частини додатку.

По-перше потрібно спробувати визначити команди, які користувач повинен мати можливість виконувати. ***Командою*** (*command*) називається великомасштабний запит певної послуги системи. Наприклад, командами можуть бути «забронювати квитки» та «знайти потрібний рядок в базі даних». Формат введення інформації та запуску на виконання змінити відносно нескладно, тому, в першу чергу, слід розробити самі команди.

Проте цілком припустимо попередньо приблизно визначити інтерфейси, щоб з їх допомогою візуалізувати роботу додатку та перевірити, чи не було пропущено щось важливе. Можна навіть зробити імітацію інтерфейсу, щоб користувачі могли спробувати попрацювати з ним і оцінити його. Логіку додатку можна імітувати фіктивними процедурами. Відділення логіки від інтерфейсу користувача надає можливість оцінити відчуття від роботи з цим інтерфейсом, поки додаток все ще перебуває в стадії розроблення.

Визначення прикордонних класів та керуючих об'єктів

Підхід BCE (Boundary-Control-Entity). Даний підхід до об'єктно-орієнтованого моделювання, заснований на три факторному уявленні класів. У мові UML у класах визначені три стереотипи: *boundary* (межа), *control* (керування) та *entity* (сутність).

Примежеві класи (boundary class) описують об'єкти, що є інтерфейсом між суб'єктом та системою. Вони виділяють частину стану системи та надають її користувачу у формі візуального відображення. Примежеві об'єкти зазвичай зберігаються після однократного виконання програми.

Керуючі класи (control class) описують об'єкти, що перехоплюють вхідні події, ініційовані користувачем, та контролюють виконання бізнес-процесу. Керуючий клас містить дії та види діяльності прецедентів.

Класи сутності (entity class) описують об'єкти, що містять семантику сутностей, що належать предметній області. Вони співвідносяться зі структурами даних бази даних. Об'єкти сутності завжди зберігаються після виконання програми та беруть участь у багатьох прецедентах.

У правильно спроектованій ієрархії пакетів суб'єкт може взаємодіяти тільки з примежевими об'єктами, об'єкти сутності можуть взаємодіяти тільки з керуючими об'єктами, керуючі об'єкти взаємодіють з об'єктами будь-якого типу (рис. 6.1).

Примежеві класи відповідають класам *GUI*-інтерфейсу. Керуючі об'єкти маніпулюють взаємодією між подіями, що ініціюють користувачі, та впливають на об'єкти сутності.

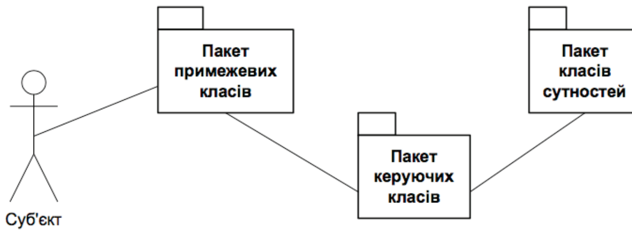

Рисунок 6.1 – З'єднання об'єктів в ієрархії *BCE*

Основна перевага *BCE* – групування класів у вигляді ієрархічних рівнів. Це сприяє кращому розумінню моделей та зменшує її складність. Також дозволяє знизити ризик виникнення класів, що беруть на себе забагато функцій, класів, що контролюють роботу всієї системи.

Перевірка щодо моделі взаємодії

Побудувавши модель класів програми, потрібно повернутися до моделі прецедентів та подумати про те, як вони можуть здійснюватися цим додатком. Наприклад, якщо користувач передає додатку команди, її параметри повинні передаватися будь-яким об'єктом інтерфейсу користувача. Запит на саму команду повинний виходити з будь-якого керуючого об'єкта. За наявності коректних моделей класів предметної області та додатку можна імітувати варіанти використання за допомогою класів. Тут потрібно мислити в термінах простежування маршрутів. Ручне моделювання допомагає переконатися в тому, що всі складові знаходяться на своїх місцях.

6.2 Модель станів програмних додатків

Модель станів додатку описує стани класів додатку і, таким чином, доповнює модель станів предметної області. Класи додатку з більшою вірогідністю продемонструють зміну свого стану, тобто спостерігати їх поведінку важливіше ніж поведінку класів предметної області.

Спочатку потрібно виділити класи додатку, що володіють кількома станами, і за допомогою моделі взаємодії визначити

події, які впливають на ці стани. Потім потрібно впорядкувати допустимі послідовності подій для кожного з класів на діаграмі станів. Після цього потрібно перевірити різні діаграми станів і переконатися, що однакові події на них дійсно відповідають одна одній. Нарешті, діаграми станів потрібно перевірити за моделями класів і взаємодії.

Побудова моделі станів додатку здійснюється у кілька етапів:

– виділити класи програми, що володіють станами;
– знайти події;
– побудувати діаграми станів;
– перевірити щодо інших діаграм станів;
– перевірити щодо моделі класів;
– перевірити щодо моделі взаємодії.

Виділення динамічних класів додатку

Модель класів програми складається з «комп'ютерних» класів, видимих користувачам і важливих для роботи додатку. Необхідно розглянути кожен з цих класів і визначити, чи володіє він кількома різними станами. Найчастіше станами володіють класи інтерфейсу користувача і класи керуючих об'єктів. Прикордонні класи, навпаки, зазвичай є статичними та здійснюють імпорт та експорт даних.

Пошук подій

Раніше ви підготували набір сценаріїв для опису моделі взаємодії додатку. Тепер ви повинні вивчити ці сценарії і виділити з них події. Хоча ці сценарії не обов'язково покривають всі можливі ситуації, вони не дадуть вам пропустити найбільш типові взаємодії і в них підкреслюються найбільш важливі події.

Зверніть увагу на різницю між послідовністю етапів побудови моделей станів для предметної області та для додатку. У першому випадку ми починали з пошуку станів, після чого виділяли події. Справа в тому, що модель предметної області концентрується на даних. Дані групуються в стани, що піддаються до впливу подій. У моделі додатку треба починати з пошуку подій, а потім виділити стани. Для моделі програми

важлива перш за все поведінка, і тому варіанти використання розкриваються сценаріями, які містять події.

Побудова діаграм станів

Наступний етап полягає в побудові діаграми станів для кожного класу додатку, що володіє значно залежною від часу поведінкою. Для цього необхідно обрати один з цих класів і взяти його діаграму послідовності. Далі треба розставити події, в яких бере участь клас, у вигляді графа, над дугами якого напишіть назви подій. Інтервал між будь-якими двома подіями буде станом. Кожному стану присвоюється конкретне ім'я, якщо воно виходить осмисленим, а якщо ні – краще залишити стан безіменним. Потім пов'язані між собою діаграми послідовності поєднуються у діаграму станів. Ця діаграма буде послідовністю станів і подій. Кожен сценарій чи діаграма послідовності визначають один маршрут на діаграмі станів.

Після цього треба виділити цикли на діаграмі. Якщо послідовність подій може повторюватися нескінченно, вона утворює цикл. Початковий та кінцевий стани з циклу збігаються. Якщо об'єкт «запам'ятовує», що він вже зробив повний прохід по циклу, ці два стани вже не є ідентичними, а тому подання у вигляді простого циклу буде некоректно. Принаймні один стан циклу повинен мати кілька вихідних переходів, або цей цикл ніколи не завершиться.

Після виділення циклів інші діаграми послідовності об'єднують з отриманою діаграмою станів. На кожній діаграмі послідовності знаходять точку, в якій вона відхиляється від інших діаграм. Ця точка відповідає наявному стану на діаграмі. Приєднайте нову послідовність подій до існуючого стану у вигляді альтернативного маршруту. Вивчаючи діаграми послідовності, можна виявити додаткові можливі події, які можуть здійснюватися в кожному стані. Вони також додаються на діаграму.

Найскладніше – визначити, в якій точці альтернативний маршрут возз'єднується зі вже вказаним на діаграмі станів. Два маршрути з'єднуються в одному стані, якщо об'єкт «забуває», за яким з них він прийшов у цей стан. У багатьох ви-

падках із знань про додаток, з очевидністю випливає, що два стани ідентичні один одному. Наприклад, кинути дві монети по п'ять центів в торговий автомат – все одно, що кинути одну монету в десять центів.

Необхідно бути акуратними з маршрутами, що здаються ідентично, але за деяких обставин можуть відрізнятися. Наприклад, деякі системи повторюють послідовність введення даних, якщо користувач припускається помилки, але через кілька невдалих спроб вони припиняють введення. Повторюються одні й ті ж події, за винятком того, що система пам'ятає про минулі помилки. Цю різницю можна відобразити за допомогою параметра (наприклад, *number of failures* – число помилок), в якому буде зберігатися відповідна інформація. Принаймні один перехід повинен залежати від цього параметра.

Розсудливе використання параметрів і умовних переходів може значно спростити діаграми станів за рахунок змішування інформації про стани з даними. Діаграми станів, що дуже сильно залежать від даних, можуть збивати з пантелику та суперечити здоровому глузду. Інша альтернатива полягає в розподілу діаграми станів на дві паралельні діаграми, використовуючи одну з них для основної послідовності, а іншу – для керуючої інформації. Наприклад, піддіаграмма, що описує ситуацію, в якій користувачеві дається можливість зробити одну помилку, може мати стани *No error* і *One error (Немає помилок і Одна помилка)*.

Після розгляду всіх типових подій на діаграму додають нетипові сценарії та виняткові ситуації. Треба розглянути події, що відбуваються в невдалі моменти: наприклад, запит на скасування транзакції, що надходить після початку її обробки. Якщо користувач (або інший зовнішній агент) не відповідає на запит системи у належний термін, необхідно породити подію тайм-ауту. Обробка помилок користувачів часто вимагає більше роздумів і кодування, ніж типові послідовності. Обробка помилок часто ускладнює структуру програми, яка інакше була б ясною та компактною, але без неї програми створювати не можна.

Діаграма станів може вважатися закінченою, якщо вона покриває всі сценарії та враховує всі події, що можуть вплинути на стани. Ви можете використовувати діаграму станів для розгляду нових сценаріїв, досліджуючи вплив подій, обробка яких ще не передбачена, на різні стани. Запитання типу «що якщо?» являють собою гарний метод перевірки повноти моделі та її здатності коректно обробляти помилки.

Для опису складних взаємодій з незалежними вхідними даними можна використовувати вкладені діаграми станів. В іншому випадку достатньо плоскої структури діаграми. Описану вище процедуру побудови діаграм стану необхідно повторити для кожного класу, поведінка якого залежить від часу.

Перевірка щодо інших діаграмам станів

Перевіряйте діаграму станів кожного класу на повноту й узгодженість. Кожна подія повинна мати відправника та одержувача. У деяких випадках вони обидва можуть бути одним і тим самим об'єктом. Стани без попередніх або наступних станів повинні викликати підозру. Переконайтеся, що вони відповідають початковим і кінцевим точкам послідовностей взаємодії. Простежте результати вхідної події від об'єкта до об'єкта через всю систему та переконайтеся, що вони відповідають сценаріям. Об'єктам властивий паралелізм. Остерігайтеся помилок синхронізації, пов'язаних з тим, що будь-яка подія відбувається в невдалий момент. Переконайтеся, що однакові події на різних діаграмах відповідають одна одній.

Перевірка щодо моделі класів

Треба переконатися, що діаграми станів узгоджені з моделями класів предметної області та додатку.

Перевірка щодо моделі взаємодії

Коли модель станів буде готова, потрібно повернутися та перевірить її за сценаріями моделі взаємодії. Треба виконати імітацію кожної послідовності поведінки вручну та переконатися, що діаграма станів дає коректну поведінку. У разі виявлення невідповідностей потрібно змінити або діаграму,

або сценарій. Іноді діаграма станів дозволяє виявити помилки в сценаріях, тому не слід заздалегідь припускати, що вони вірні.

Далі візьміть модель станів і прослідкуйте дозволені маршрути. Вони описують всі можливі сценарії. Запитайте себе, чи є сенс в цих сценаріях. Якщо його немає, змініть діаграму станів. Часто ця процедура дозволяє виявити корисну поведінку, яку було пропущено на попередніх етапах. Виявлення несподіваної інформації, що випливає з проекту, а також важливих (і іноді таких, що здаються очевидними) властивостей системи вказує на високу якість цього проекту.

Додавання операцій

Стиль об'єктно-орієнтованого аналізу надає набагато менше значення визначенню операцій, ніж інші методології розроблення. Список потенційно корисних операцій здається нескінченним, і важко вибрати момент, щоб зупинитися та припинити додавати їх. Операції випливають з певних джерел, і саме на цьому етапі прийшов час додати їх у модель. Операції повинні:

– виконувати читання та запис значень атрибутів і зв'язків асоціацій, що передбачаються моделлю класів;

– виконувати діяльність визначену в прецедентах;

– виконувати «операції за списком» – операції не залежать від програми. Ці операції дозволяють розширити визначення класу в порівнянні з потребами поточного завдання.

Спрощення операцій

Вивчіть модель класів на предмет наявності подібних операцій і різних варіацій однієї операції. Потрібно розширити визначення операцій таким чином, щоб вони містили ці варіації і особливі випадки. Скрізь, де це можна, потрібно використовувати спадкування для скорочення кількості різних операцій. Можна вводити нові суперкласи для спрощення операцій, за умови, що їх введення не виявляється «насильницьким» і неприродним. Кожну операцію потрібно розміщувати на визначеному рівні ієрархії класів. У результаті

такого уточнення моделі кількість операцій скорочується і вони стають більш потужними, але складати їх специфікацію простіше, ніж для початкових операцій, тому що вони більш універсальні та загальні.

6.3 Шаблони розподілення обов΄язків

Шаблон це апробоване рішення загального завдання.

Принципи об'єктно-орієнтованого проектування відображені у шаблонах розподілення обов'язків *GRASP* (*General Responsibility Assignment Software Patterns*).

В *UML* обов'язки визначаються як контракт або обов'язки класифікатора. Вони описують поведінку об'єкта, та використовуються під час об'єктно-орієнтованого проектування. Можна виділити два типа обов'язків – знання та дії.

Далі розглянемо п'ять шаблонів *GRASP*: *Information Expert* (інформаційний експерт), *Creator* (твореці), *High Cohesion* (високе з'єднання), *Low Coupling* (низьке зачеплення) та *Controller* (контролер). Приклади використання даних шаблонів були взяті з [68].

6.3.1 Шаблон Information Expert

Проблема. Як розподілити обов'язки між об'єктами при об'єктно-орієнтованому проектуванні?

У моделі системи можуть бути визначені десятки або сотні програмних класів, а в додатку може знадобитися виконання сотень чи тисяч обов'язків. Під час об'єктно-орієнтованого проектування при формулюванні принципів взаємодії об'єктів необхідно розподілити обов'язки між класами.

При правильному виконанні цього завдання система стає набагато простішою для розуміння, підтримки та розширення. Крім того, з'являється можливість повторного використання вже розроблених компонентів у наступних програмах.

Вирішення проблеми

Для вирішення проблеми необхідно виконати такі кроки:

1. Чітко сформулювати обов'язки, які необхідно розподілити.

2. Якщо в моделі проектування є необхідні класи, то взяти їх за основу, якщо ні, то за основу взяти модель предметної області та створити відповідні програмні класи.

3. Обрати клас, у якого є інформація, що необхідна для виконання обов'язку.

Приклад

У системі обслуговування продаж певному класу необхідно знати загальну суму продажу.

Проблема: який клас повинен відповідати за знання загальної суми продажу?

Згідно з шаблоном, потрібно визначити, об'єкти яких класів містять інформацію, необхідну для обчислення загальної суми.

Припустимо, ми знаходимося на самому початку етапу проектування, коли модель проектування розроблена в мінімальному обсязі. Отже, кандидатуру на роль інформаційного експерта слід шукати в моделі предметної області, фрагмент якої відображений на рис. 6.2.

Імовірно, на цю роль підійде концептуальний клас «Продаж». Тоді в модель проектування потрібно додати відповідний програмний клас під ім'ям «Sale» і присвоїти йому обов'язок обчислення загальної вартості, що реалізується за допомогою виклику методу *getTotal*. При такому підході скорочується розрив між організацією програмних об'єктів і відповідних їм понять з предметної області.

Яка інформація потрібна для обчислення загальної суми? Необхідно з'ясувати вартість усіх проданих товарів «Елемент продажу» і підсумувати ці проміжні суми. Такою інформацією володіє лише екземпляр об'єкта «Продаж». Отже, з точки зору шаблону *Information Expert* об'єкт «:Sale» підходить для виконання цього обов'язку, тобто є інформаційним експертом.

Як уже згадувалося, подібні питання розподілу обов'язків часто виникають при створенні діаграм взаємодій. Уявіть, що ви приступили до роботи, почавши створення діаграм для розподілу обов'язків між об'єктами. Прийняті рішення ілю-

струються на фрагменті діаграми взаємодій, зображеній на рис. 6.3.

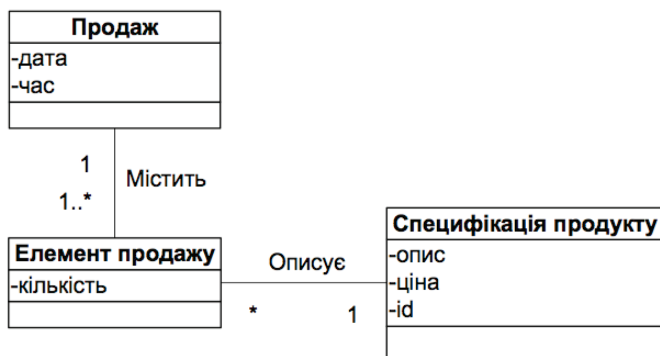

Рисунок 6.2 – Фрагмент предметної області для системи продаж

Рисунок 6.3 – Результат моделювання обов'язків класу «Sale»

Проте на даному етапі виконана не вся робота. Яка інформація потрібна для обчислення проміжної суми елементів продажу? Для цього необхідно знати значення атрибуту *кількість* класу «Елемента продажу» та атрибуту *вартість* класу «Специфікація продукту». Отже, відповідно шаблону *Information Expert*, проміжну суму повинен обчислювати об'єкт «:SaleItem». Іншими словами, цей об'єкт є інформаційним експертом.

У термінах діаграм взаємодій це означає, що об'єкт «:Sale» повинен передати повідомлення *getSubtotal* кожному об'єкту «:SaleItem», а потім підсумувати отримані результати. Цей процес проілюстровано на рис. 6.4.

Для виконання обов'язку, пов'язаного зі знанням і наданням проміжної суми, об'єкту «:SaleItem» повинна бути відома вартість товару.

У даному випадку в якості інформаційного експерта буде виступати об'єкт «:ProductSpecification».

Результати проектування зображені на рис. 6.5 та рис. 6.6.

Рисунок 6.4 – Призначення обов'язку обчислення проміжної суми

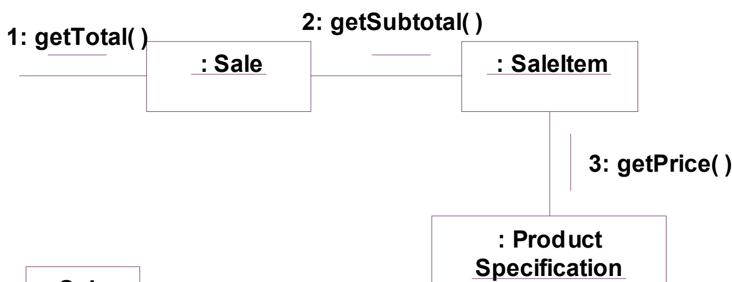

Рисунок 6.5 – Діаграма взаємодії

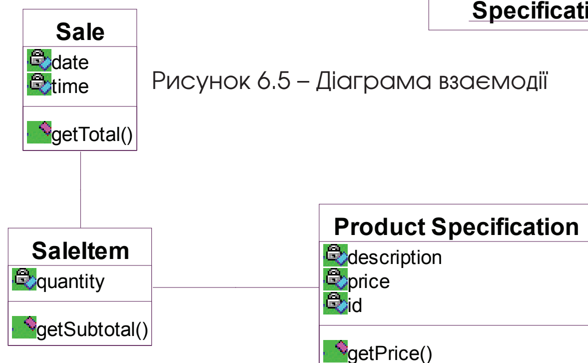

Рисунок 6.6 – Фрагмент моделі проектування

На завершення можна сказати наступне. Для виконання обов'язку «знати і надавати загальну суму продажу» трьом об'єктам класів були таким чином присвоєно три обов'язки:

– клас «Sale» – обов'язок «*Знання загальної суми продажу*»;

– клас «Saleltem» – обов'язок «*Знання проміжної суми для даного товару*»;

– клас «Product Specification» – обов'язок «*Знання ціни товару*».

Розгляд та розподіл обов'язків виконувалися в процесі створення діаграми взаємодій. Потім отримані результати можуть бути реалізовані в розділі методів діаграми класів (рис. 6.5–6.6).

Рекомендації. При розподілі обов'язків шаблон *Information Expert* використовується набагато частіше від будь-якого іншого шаблону. У ньому визначені основні принципи, які вже давно використовуються в об'єктно-орієнтованому проектуванні. Шаблон *Information Expert* не містить неясних або заплутаних ідей та відображає звичайний інтуїтивно зрозумілий підхід. Він полягає в тому, що об'єкти здійснюють дії, пов'язані з наявною у них інформацією.

Треба звернути увагу, що для виконання обов'язку найчастіше потрібна інформація, розподілена між різними класами або об'єктами. Це передбачає, що повинно існувати багато «часткових» експертів, що взаємодіють при виконанні спільного завдання. Наприклад, для обчислення загальної суми продажу в кінцевому рахунку необхідна взаємодія трьох класів. Якщо інформація розподілена між різними об'єктами, то при виконанні спільного завдання вони повинні взаємодіяти за допомогою повідомлень.

У деяких ситуаціях застосування шаблону *Information Expert* не бажане, наприклад у зв'язку з проблемами зі зв'язуванням і зачепленням.

Перевагою шаблону *Information Expert є* підтримка інкапсуляції – для виконання необхідних завдань об'єкти використовують власні дані. Таку можливість забезпечує також шаблон *Low Coupling*, застосування якого призводить до створення більш надійних і легко підтримуваних систем.

(*Low Coupling* є шаблоном *GRASP*, який буде розглянутий трохи нижче.)

Споріднені типові рішення – *Low Coupling та High Cohesion*.

6.3.2 Шаблон Creator

Проблема. Який клас повинен відповідати за створення нового екземпляра певного класу?

Створення об'єктів в об'єктно-орієнтованій системі є одним із найбільш стандартних видів діяльності. Отже, при призначенні обов'язків, пов'язаних зі створенням об'єктів, корисно керуватися певним основним принципом. Правильно розподіливши обов'язки при проектуванні, можна створити слабко пов'язані незалежні компоненти з можливістю їх подальшого використання, спростити їх, а також забезпечити інкапсуляцію даних та їх повторне використання.

Вирішення проблеми

Призначити класу B обов'язок створювати екземпляри класу A якщо виконується одна з наступних умов:

– клас **B** агрегує (*aggregate*) об'єкти **A;**

– клас **B** містить (*contains*) об'єкти **A**;

– клас **B** записує (*records*) екземпляри об'єктів **A**;

– клас **B** активно використовує (*closely uses*) об'єкти **A**;

– клас **B** володіє даними ініціалізації (*has the initializing data*), які будуть передаватися об'єктам **A** при їх створенні (тобто при створенні об'єктів **A** класс **B** є експертом);

– клас **B** – творець (*creator*) об'єктів **A**.

Якщо виконується декілька з цих умов, то краще використовувати клас **B**, який агрегує або містить клас **A**.

Розгляд та розподіл обов'язків виконується в процесі створення діаграм взаємодій. Потім отримані результати можуть бути реалізовані в конкретних методах, розміщених у розділі методів діаграми класів.

Приклад

Хто в системі продаж повинен відповідати за створення нового екземпляра об'єкта «:SaleItem»? Відпо-

відності до шаблону *Creator*, необхідно знайти клас, що агрегує (містить і т. д.) екземпляри об’єктів «:SaleItem». За основу візьмемо модель предметної області, що зображена на рис. 7.1.

Оскільки об’єкт «:Sale» містить об’єкти «:SaleItem», то відповідно до шаблону *Creator* він виступає гарним кандидатом для виконання обов’язку по створенню екземпляра об’єкта «:SaleItem».

Одже це призводить до необхідності розроблення взаємодії об’єктів (рис. 6.7).

При такому розподілі обов’язків потрібно, щоб в об’єкті «:Sale» було визначено метод *makeLineItem*.

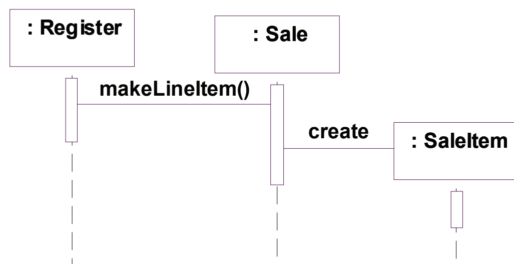

Рисунок 6.7 – Створення екземпляра об´єкта «:SaleItem»

Рекомендації

Шаблон *Creator* визначає спосіб розподілу обов’язків, пов’язаний з розподілом об’єктів. В об’єктно-орієнтованих системах це завдання є одним з найбільш поширених. Основним призначенням шаблону *Creator* є виявлення об’єкта-творця, що при виникненні будь-якої події має бути пов’язаний з усіма створеними ним об’єктами. При такому підході забезпечується низький ступінь зв’язаності. Ціле агрегує свої частини, контейнер зберігає свій вміст, реєстр веде облік. Всі ці взаємозв’язки є дуже поширеними способами взаємодії класів на діаграмах класів. У шаблоні *Creator* визначається, що зовнішній контейнер або клас-реєстр – це хороші кандидати на виконання обов’язків, пов’язаних зі створенням сутностей, які вони будуть містити або реєструвати.

У деяких випадках у ролі *творця* обирається клас, що містить дані ініціалізації, які передаються об'єкту під час його створення. Насправді це приклад використання шаблону *Information Expert*. У процесі створення ініціалізуючі дані передаються за допомогою методу ініціалізації певному виду, такого як конструктор мови Java з параметрами.

Коли створення екземпляра виконується при реалізації деякої умови на основі будь-яких зовнішніх властивостей, то краще використовувати шаблон *Factory* і делегувати обов'язок створення екземплярів допоміжному класу.

До переваг слід віднести, що застосування шаблону *Creator* не підвищує ступеня пов'язаності, оскільки створений (created) клас, як правило, виявляється видимим для класу-творця за допомогою наявних асоціацій.

Споріднені типові рішення – *Low Coupling, Factory*.

6.3.3 Шаблон ***Low Coupling***

Проблеми: Як забезпечити залежність між класами при незначному впливі змін та підвищити можливість повторного використання?

Ступінь зв'язаності (*coupling*) – це міра, яка визначає наскільки жорстко один елемент зв'язаний з іншими елементами, або якою кількістю даних про інші елементи він володіє. Елемент з низьким ступенем зв'язаності (або слабким зв'язуванням) залежить від невеликої кількості інших елементів. Вираз «дуже багато» залежить від контексту, проте необхідно провести його оцінку.

Клас з високим ступенем зв'язаності (або жорстко зв'язаний) залежить від безлічі інших класів. Однак наявність таких класів небажана, оскільки вона призводить до виникнення наступних проблем:

– зміни у зв'язаних класах призводять до локальних змін у даному класі;

– ускладнюють розуміння кожного класу окремо;

– ускладнюється повторне використання, оскільки для цього потрібен додатковий аналіз класів, з якими пов'язаний даний клас.

Вирішення проблеми

Розподілити обов'язки таким чином, щоб ступінь пов'язаності залишався низьким.

Приклад

Розглянемо інший фрагмент діаграми предметної області для системи продаж (рис. 6.8).

Створимо відповідні програмні класи. Припустимо, що необхідно створити екземпляр класу «:Payment» і пов'язати його з об'єктом «:Sale». Який клас повинен відповідати за виконання цієї операції? Оскільки в реальній предметній області реєстрація об'єкта Платіж виконується об'єктом Реєстр, відповідно до шаблону Creator, об'єкт «:Register» є гарним кандидатом для створення об'єкта «:Payment». Потім екземпляр об'єкта «:Register» повинен передати повідомлення *addPayment* об'єкту «:Sale», вказавши в ролі параметра новий об'єкт «:Payment». Наведені міркування відображені на фрагменті діаграми взаємодій, представленої на рис. 6.9.

Такий розподіл обов'язків припускає, що клас «:Register» має знання про дані класу «:Payment» (тобто зв'язується з ним).

Зверніть увагу на позначення, прийняті в мові UML. Екземпляру об'єкта «:Payment»присвоєно явне ім'я p, щоб його можна було надалі використовувати.

Альтернативний спосіб створення об'єкта «:Payment» та його зв'язування з об'єктом «:Sale» показаний на рис. 6.10.

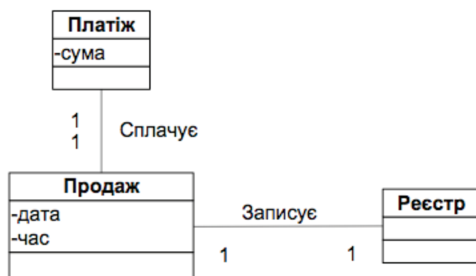

Рисунок 6.8 – Фрагмент діаграми предметної області для системи продаж

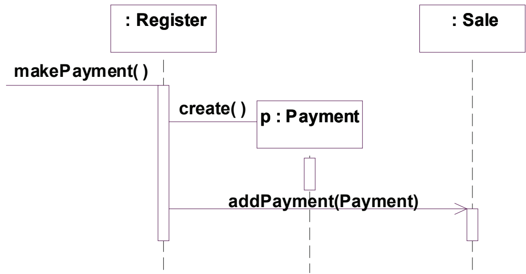

Рисунок 6.9 – Створення нового екземпляру «:Payment» об´єктом «:Register»

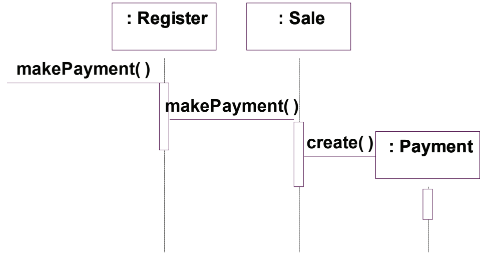

Рисунок 6.10 – Створення нового екземпляру «:Payment» об´єктом «:Sale»

Який з методів проектування, заснований на розподілі обов'язків, забезпечує більш низький ступінь зв'язування? В обох випадках передбачається, що в кінцевому підсумку об'єкту «:Sale» має бути відомо про існування об'єкта «:Payment». При використанні першого способу, коли об'єкт «:Payment» створюється за допомогою об'єкта *Register*, між цими двома об'єктами додається новий зв'язок, тоді як другий спосіб ступінь зв'язування об'єктів не підсилює. З точки зору числа зв'язків між об'єктами, кращим є другий спосіб, оскільки в цьому випадку забезпечується низький ступінь зв'язування. Наведена ілюстрація є прикладом того, як при використанні двох різних шаблонів – *Low Coupling* і *Creator* – можна прийти до двох різних рішень.

На практиці рівень зв'язування не розглядається окремо від інших принципів, сформульованих в шаблонах *Information Expert* і *High Cohesion*.

Рекомендації

У шаблоні *Low Coupling* описується принцип, про який не можна забувати протягом усіх стадій роботи над проектом.

В об'єктно-орієнтованих мовах програмування, таких як *C++, Java і C #*, є такі стандартні способи зв'язування об'єктів **TypeX** і **TypeY**:

– об'єкт **TypeX** містить атрибут (змінну-член), який посилається на екземпляр об'єкта **TypeY** або сам об'єкт **TypeY**;

– об'єкт **TypeX** викликає служби об'єкта **TypeY**;

– об'єкт **TypeX** містить метод, який будь-яким чином посилається на екземпляр об'єкта **TypeY** або сам об'єкт **TypeX** (зазвичай мається на увазі використання **TypeY** як тип параметра локальної змінної або значення, що повертається);

– об'єкт **TypeX** є прямим або непрямим підкласом об'єкта **TypeY**. І Об'єкт **TypeY** є інтерфейсом, а **TypeX** реалізує цей інтерфейс.

Шаблон *Low Coupling* має на увазі такий розподіл обов'язків, який не тягне за собою надмірного підвищення ступеня зв'язування, що призводить до негативних результатів.

Підклас жорстко пов'язаний зі своїм суперкласом. Тому приймаючи рішення про спадкування властивостей об'єктів, слід враховувати, що відношення успадкування підвищує ступінь зв'язаності класів.

Не існує абсолютних заходів для визначення дуже високого ступеня зв'язування. Важливо лише розуміти ступінь зв'язаності об'єктів на поточний момент і не упустити той момент, коли подальше підвищення ступеня зв'язаності може призвести до виникнення проблем. У цілому, слід керуватися таким принципом: класи, які є досить загальними за своєю природою і з високою ймовірністю будуть повторно використовуватися в подальшому, повинні мати мінімальну ступінь зв'язаності з іншими класами.

Крайнім випадком при реалізації шаблону *Low Coupling* є повна відсутність зв'язування між класами. Така ситуація теж не бажана, оскільки базовою ідеєю об'єктного підходу є система пов'язаних об'єктів, що «спілкуються» між собою за допомогою передання повідомлень. При занадто частому використанні принципу слабкого зв'язування система складатиметься з декількох ізольованих складних активних об'єктів, що самостійно виконують усі операції, і безлічі пасивних об'єктів, основна функція яких зводиться до зберігання даних. Тому при створенні об'єктно-орієнтованої системи має бути наявний деякий оптимальний ступінь зв'язування між об'єктами, що дозволяє виконувати основні функції за допомогою взаємодії цих об'єктів.

Високий ступінь зв'язування сам по собі не є проблемою. Проблемою є жорстке зв'язування з нестійкими в деякому відношенні елементами.

Важливо розуміти, що розробник може забезпечувати гнучкість програми, реалізовувати принцип інкапсуляції і дотримуватися принципу слабкого зв'язування в багатьох аспектах системи. Однак без переконливої мотивації не слід будь-якою ціною боротися за зменшення ступеня зв'язування об'єктів.

Споріднено типове рішення – *Protected Variations*.

6.3.4 Шаблон *High cohesion*

Проблема. Як забезпечити можливість керування складністю?

У термінах об'єктно-орієнтованого проектування зачеплення (*cohesion*) – це міра зв'язаності та сфокусованості обов'язків класу. Вважається, що елемент має високий ступінь зачеплення, якщо його обов'язки тісно пов'язані між собою і він не виконує непомірних обсягів роботи. У ролі таких елементів можуть виступати класи, підсистеми і т. д.

Клас з низьким ступенем зачеплення виконує багато різнорідних функцій або не пов'язаних між собою обов'язків. Такі класи створювати не бажано, оскільки вони призводять до

виникнення таких проблем, як труднощі розуміння; складнощі при повторному використанні; складнощі підтримки; ненадійність, постійна схильність до змін.

Класи зі слабким зачепленням, як правило, є дуже «абстрактними» або виконують обов'язки, що можна легко розподілити між іншими об'єктами.

Вирішення проблеми

Розподілити обов'язки так, щоб забезпечити високий ступінь зачеплення.

Приклад

Для аналізу шаблону *High Cohesion* можна використовувати той же приклад, що і для *Low Coupling*.

У першому варіанті, для створення екземпляра об'єкта «:Payment» використовується об'єкт «:Register». Тоді екземпляр об'єкта «:Register» відправляє повідомлення *addPayment* об'єкту «:Sale», передаючи як параметр новий екземпляр об'єкта «:Payment» (рис. 6.8).

При такому розподілі обов'язків платежі виконує об'єкт «:Register», тобто об'єкт «:Register»частково несе відповідальність за виконання системної операції *makePayment*. У цьому відокремленому прикладі це прийнятно. Однак, якщо і надалі покладати на клас «:Register» обов'язки з виконання все нових і нових функцій, пов'язаних з іншими системними операціями, то цей клас буде занадто перевантажений і матиме низький ступінь зачеплення.

Припустимо, додаток повинний виконувати п'ятдесят системних операцій і всі вони покладені на клас Register. Якщо цей об'єкт буде виконувати всі операції, то він стане надмірно «роздутим» і не буде мати властивість зачеплення. І справа не в тому, що одне завдання створення екземпляра об'єкта «:Payment» сама по собі знизила ступінь зачеплення об'єкта «:Register»; воно є частиною загальної картини розподілу обов'язків.

У другому варіанті (рис. 6.10) функція створення екземпляра платежу делегована об'єкту «:Sale». Завдяки цьому підтримується більш високий ступінь зачеплення об'єкта

«:Register». Оскільки такий варіант розподілу обов'язків забезпечує низький рівень зв'язування та більш високий ступінь зачеплення, він є кращим.

На практиці рівень зачеплення не розглядають ізольовано від інших обов'язків і принципів, що забезпечуються шаблонами *Information Expert* і *Low Coupling*.

Рекомендації

Як і про принцип слабкого зв'язування, про високий ступінь зачеплення слід пам'ятати під час всього процесу проектування. Цей шаблон необхідно застосовувати при оцінюванні ефективності кожного проектного рішення.

При визначенні рівня зачепленості можна використовувати наступні рекомендації.

Дуже слабке зачеплення вважається, коли тільки один клас відповідає за виконання безлічі операцій у найрізноманітніших функціональних областях.

Слабке зачеплення – це коли клас несе повну відповідальність за виконання складного завдання із однієї функціональної області.

Сильне зачеплення – коли клас має середню кількість обов'язків з однієї функціональної області та для виконання своїх завдань взаємодіє з іншими класами.

Середнє зачеплення – коли клас має нескладні обов'язки в декількох різних областях, логічно пов'язаних з концепцією цього класу, але не пов'язаних між собою.

Як правило, клас з високим ступенем зачеплення містить порівняно невелику кількість методів, що функціонально тісно пов'язані між собою, та не виконує дуже багато функцій. Він взаємодіє з іншими об'єктами для виконання більш складних завдань.

Класи з високим ступенем зачеплення є дуже бажаними, оскільки вони дуже прості в розумінні, підтримці та повторному використанні. Високий ступінь однотипної функціональності в поєднанні з невеликим числом операцій спрощують підтримку і модифікацію класу, а також можливість його повторного використання.

Зв'язування і *зачеплення* – це старі принципи проектування програмних продуктів. Об'єктне проектування не йде врозріз із застарілими підходами. Ще однією особливістю, тісно пов'язаною зі зв'язуванням і зачепленням, є модульність. *Модульність* – це властивість системи, розбитої на безліч модулів з високим ступенем зачеплення і слабким зв'язуванням.

Модульне проектування забезпечується за рахунок створення методів і класів з високим зачепленням. На рівні базових об'єктів модульність досягається за рахунок проектування кожного методу з явно виділеною конкретною метою і групування набору взаємопов'язаних методів у рамках одного класу.

Некоректне зв'язування породжує неправильне зачеплення і навпаки.

Існує кілька випадків, коли низьке зачеплення виявляється виправданим.

Одна з таких ситуацій виникає в тому випадку, коли обов'язки або код групуються в одному класі або компоненті для спрощення його підтримки однією людиною. Проте в даному випадку необхідно пам'ятати про те, що таке угрупування може призвести і до ускладнення підтримки.

Інший приклад слабкого зачеплення має відношення до розподілених серверних об'єктів. Оскільки швидкодія системи визначається продуктивністю віддалених об'єктів та їх взаємодією, іноді бажано створити декілька більш значних серверних об'єктів зі слабким зачепленням, що надають інтерфейс багатьох операцій.

До переваг використання цього шаблону слід віднести підвищення ясності та простоти проектних рішень; спрощення підтримки та доопрацювання; забезпечення слабкого зв'язування; поліпшення можливості повторного використання, оскільки клас з високим ступенем зачеплення виконує конкретне завдання.

6.3.5 Шаблон Controller

Проблеми: Який клас повинен відповідати за оброблення вхідних системних подій?

Системна подія *(system event)* – це подія високого рівня, що генерується зовнішнім виконавцем (подія з зовнішнім входом*)*. Системні події пов'язані з ***системними операціями*** *(system operation*), тобто операціями, що виконуються системою у відповідь на події.

Наприклад, коли касир у системі продаж тисне на кнопку «*Сплатити*», він генерує системну подію, яка свідчить про завершення торговельної операції. Аналогічно, коли користувач текстового процесора обирає команду «Орфографія», він генерує системну подію «виконати перевірку орфографії».

Контролер (controller) – це об'єкт, що не належить до інтерфейсу користувача та відповідає за оброблення системних подій. Контролер визначає методи для виконання системних операцій.

Вирішення проблеми

Делегування обов'язків з оброблення системних повідомлень класу, що задовольняє одній із наступних умов:

– клас являє собою всю систему в цілому, пристрій або підсистему (зовнішній контролер);

– клас являє собою сценарій певного прецеденту, в рамках якого виконується оброблення всіх системних подій, і зазвичай називається <Прецедент> Handler, <Прецедент> Coordinator або <Прецедент> Session (контролер прецеденту або контролер сеансу).

Для всіх системних подій у рамках одного сценарію прецеденту використовується один і той самий клас-контролер.

Сеанс – це екземпляр взаємодії з виконавцем. Сеанси можуть мати довільну довжину, але найчастіше організовані в рамках прецеденту (сеанси прецеденту).

Наслідок. Зауважимо, що до цього переліку не включаються класи, що реалізують вікно, аплет, додаток, вид і документ. Такі класи не виконують завдання, пов'язані з системними подіями. Вони зазвичай отримують повідомлення та делегують їх контролерам.

Приклад

Інформаційна система продажу повинна обробляти такі системні операції – закінчити продаж (*endSale*),

додати елемент продажу (*enterItem*), створити новий продаж (*makeNewSale*), виконати платіж (*makePayment*) та інші. Розглянемо приклад визначення класу, що повинний виступати як контролер для системних подій типу enterItem або endSale.

Відповідно до шаблону *Controller* це може бути:

– клас, що являє собою систему взагалі (наприклад Register, POSSysytem);

– клас, штучний обробник усіх системних подій (наприклад ProcessSaleHandler, ProcessSession) (рис. 6.11).

Вибір найбільш придатного визначається зачепленням та зв'язаністю.

Рекомендації

Більшість систем отримує зовнішні події. Зазвичай вони пов'язані з графічним інтерфейсом користувача. Крім того, системі можуть передаватися зовнішні повідомлення, наприклад, при обробленні телекомунікаційних сигналів або сигналів від датчиків у системах керування.

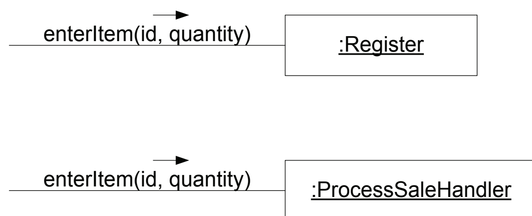

Рисунок 6.11 – Варіанти контролера

У всіх випадках при використанні об'єктно-орієнтованого підходу для оброблення зовнішніх подій застосовуються контролери. Шаблон *Controller* забезпечує найбільш типові проектні рішення для цього випадку. *Контролер* – це своєрідний вид інтерфейсу між рівнями предметної області і графічного подання.

Щоб забезпечити можливість підтримки інформації про стан прецеденту, для оброблення всіх системних подій у рамках одного прецеденту повинен використовуватися один і той же клас контролера. Така інформація може знадобити-

ся, наприклад, для ідентифікації моменту порушення послідовності системних подій (наприклад, виконання операції *makePayment* перед виконанням операції *endSale*). Для різних прецедентів можна використовувати різні контролери.

Типовою помилкою при створенні контролерів є покладання на нього надто великої кількості обов'язків.

Зазвичай контролер повинен лише делегувати функції іншим об'єктам і координувати їх діяльність, а не виконувати ці дії самостійно.

Першим типом контролерів є ***зовнішній контролер*** (*facade controller*), що являє собою всю «систему», пристрій або підсистему. Основна ідея зводиться до вибору певного класу, ім'я якого охоплює всі прошарки програми. Цей клас забезпечує головну точку виклику всіх служб з інтерфейсу користувача та звернення до інших прошарків. Наприклад, клас що являє собою всю систему – «Register».

Зовнішні контролери зручно використовувати в тому випадку, коли існує лише кілька системних подій або системні повідомлення неможливо перенаправити іншим контролерам, на зразок системи обробки повідомлень.

Якщо застосовується ***контролер прецеденту*** (*use-case controller*), то для кожного прецеденту повинен існувати окремий контролер. Зауважимо, що це не об'єкт з предметної області, а штучна конструкція, що підтримує життєдіяльність системи. Наприклад, якщо в системі використовуються прецеденти «*Оформлення продажу*» та «*Повернення товару*», то в ній може бути реалізований клас «ProcessSaleHandler» («Оброблювач продажу»).

У тому випадку, коли застосування зовнішнього контролера призводить до слабкого ступеня зачеплення або високого ступеня зв'язування, слід використовувати контролери прецедентів. Контролер прецеденту вводиться в тому випадку, якщо існуючий контролер занадто «роздувається» при покладанні на нього додаткових обов'язків. Контролери прецедентів слід застосовувати при наявності великої кількості системних подій, розподілених між різними процесами.

Такі контролери дозволяють розділяти обов'язки їх оброблення між декількома класами і відслідковувати стан поточного сценарію.

Важливим наслідком застосування шаблону Controller є винесення обов'язків виконання системних подій за межі рівня представлення та зовнішнього інтерфейсу об'єктів (наприклад, об'єктів вікон або аплетів). Іншими словами, системні операції, що відображають процеси в предметній області, повинні оброблятися на рівні логіки додатка або реалізації об'єктів, а не на рівні інтерфейсу. Це питання більш докладно розглядається нижче.

Об'єкт-контролер, як правило, відноситься до клієнтської частини програми і функціонує в рамках того ж процесу, що і інтерфейс користувача. Тому шаблон *Controller* безпосередньо непридатний, якщо в ролі клієнта виступає *Web*-броузер. Для такої архітектури існують різні шаблони оброблення системних подій, що грунтуються на використанні серверних технічних каркасів, зокрема на базі сервлетів *Java*. Найчастіше основна ідея зводиться до використання контролерів прецедентів у серверній частині програми, коли оброблення кожного прецеденту здійснює або окремий сервлет, або компонент-обробник сеансу *EJB (Enterprise JavaBeans*). Серверний об'єкт-обробник сеансу обслуговує один сеанс взаємодії із зовнішнім виконавцем.

Якщо інтерфейс із користувачем забезпечується не через *Web*-броузер (наприклад, через графічний інтерфейс *Windows* або *Swing*), але додаток викликає віддалені служби, то такий додаток частіше за все грунтується на шаблоні Controller. З інтерфейсу користувача направлено запит локальному класу-контролеру, що належить до клієнтської частини програми, а цей клас може перенаправляти весь запит або його частину віддаленим службам. При такому рішенні знижується ступінь зв'язування між інтерфейсом користувача і віддаленими службами.

Таким чином, клас-контролер отримує запити від об'єктів рівня інтерфейсу користувача та координує їх виконання, звичайно делегуючи обов'язки іншим об'єктам.

Погано спроектований клас контролера має низький ступінь зачеплення: він виконує дуже багато обов'язків і є несфокусованим. Такий контролер називається *роздутим* (*bloated controller*). Ознаки роздутого контролера такі:

– у системі є єдиний клас контролера, який отримує всі системні повідомлення, яких надходить дуже багато, що часто виникає при використанні зовнішнього контролера;

– контролер сам виконує всі завдання, не делегуючи обов'язки інших класів. Зазвичай це призводить до порушення основних принципів шаблонів Information Expert і High Cohesion;

– контролер має багато атрибутів і містить значний обсяг інформації про систему або предметної області, яку необхідно розподілити між іншими об'єктами, або дублює інформацію, що зберігається в інших об'єктах.

Існує кілька рецептів для усунення проблеми роздутого контролера:

1. Додайте декілька контролерів; не потрібно обмежуватися лише одним. Крім зовнішніх контролерів, використовуйте контролери прецедентів.

2. Спроектуйте контролер таким чином, щоб він делегував обов'язки з виконання системних операцій іншим об'єктам.

Нагадаємо, що важливим наслідком застосування шаблону Controller є такий факт: об'єкти інтерфейсу (наприклад, вікна) не обробляють системні події.

Споріднені типові рішення – Command, Facade, Layers, Pure Fabrication.

6.4 Практичні завдання

Завдання 1. Для системи *«Склад»* з розділу 6 визначте 4 діючих особи, що взаємодіють з системою. Визначте межі системи. Визначте потік подій між ними та системою.

Завдання 2. Для системи *«Склад»* з розділу 6 створіть модель класів програмного додатку. Визначте класи для об-

робки подій. Створіть діаграму пакетів відповідно до підходу *BCE*.

Завдання 3. Підготуйте реферати за наступними тематиками:

– архітектурні (*architectural*) шаблони (*systems design*);

– шаблони проектування (*GoF*) ;

– шаблони аналізу – analysis patterns (*recurring & reusable analysis models*);

– шаблони архітектури корпоративних програмних додатків (*patterns of Enterprise Application Architecture*);

– шаблони організації (*structure of organizations/projects*);

– шаблони процесів (*software process design*).

Завдання 4. Розглянемо частину моделі предметної області для системи *«Редакційний відділ»* (рис. 6.12)

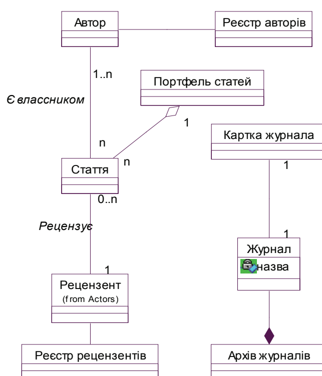

Рисунок 6.12 – Модель предметної області

Використовуючи шаблони *GRASP* розробіть модель проектування. Визначте який клас повинний відповідати за створення об'єктів «:<u>Стаття</u>», «:<u>Журнал</u>», «:<u>Рецензент</u>». Забезпечте високий рівень зачеплення.

Завдання 5. Розробіть прототип інтерфейсу для додавання інформації про статтю, авторів та рецензентів (див. завдання 2). Розробіть 2 варіанти контролерів – у першому випадку зовнішній контролер, у другому випадку – контролер прецеденту.

6.5 Контрольні запитання

1. В чому різниця моделі предметної області та моделі проектування?
2. Що описують класи додатків?
3. Який підхід використовується для визначення прикордонних класів та керуючих об'єктів?
4. У чому особливість шаблонів розподілення обов'язків?
5. Які кроки необхідно виконати, при визначенні ін формаційного експерта?
6. Які шаблони використовуються разом з шаблоном *Informational Expert*?
7. Які умови мають використовуватись для призначення обов'язку створювати певний екземпляр класу?
8. Який шаблон можна використовувати замість *Creator*, коли створення екземпляру виконується за умови виконання зовнішніх подій?
9. Що означає ступінь зв'язаності? Які стандартні способи зв'язування об'єктів в об'єктно-орієнтованих мовах ви знаєте?
10. Які рівні зачеплення існують?
11. Як співвідносяться ступень зв'язаності та рівень зачеплення?
12. З якою метою використовується шаблон Controller?
13. Назвіть ознаки роздутого контролера. Що можна зробити для усунення роздутого контролера?
14. Разом з якими шаблонами використовується шаблон Low Coupling?
15. Коли використовуються шаблони GRASP?

6.6 Література до розділу

1. Bragina T. Testing methods library for applications with web-based interfaces / T. Bragina, G. Tabunshchyk, D. Moroka // Central European Researchers Journal. – 2015. Vol. 2. -pp.90-94
2. OOP Tutorial Week 4: Class, Responsibilities, Collaborators. :[Електрон. ресурс]. – Режим доступу: http://www.inf.ed.ac.uk/teaching/courses/inf1/op/Tutorials/2008/crc.html.
3. Mishra J. Software Engineering. [Текст] / J. Mishra, A. Mohanty. – Dorling Kindersley (India), 2012. – 373 p.
4. Schmidt, D.C. Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects / D. C. Schmidt,M. Stal,Hans. – USA: John Wilye & Sons, 2013. – 450 p.
5. Ларман К. Применение UML и шаблонов проектирования. 2-е изд. Пер. с англ. -М.:Издательский дом «Вильямс», 2004. -625 с.: ил.

7 ПРОЕКТУВАННЯ СИСТЕМИ

7.1 Основи проектування систем

У процесі аналізу ми зосереджуємося на тому, що повинно бути зроблено, і не замислюємося про те, як це слід робити. В процесі проектування розробники приймають рішення про те, яким чином вони будуть вирішувати завдання, спочатку на високому рівні, а потім в деталях.

Проектування системи – це перший етап проектування, в процесі якого вибирається базовий підхід до вирішення завдання. Розробники формулюють загальну структуру та стиль рішення.

Архітектура системи визначає її розбиття на підсистеми. Крім того, архітектура створює контекст, у якому приймаються подальші рішення про детальну будову системи. Створення архітектури вимагає послідовного виконання таких дій:

– оцінити продуктивність системи;
– скласти план повторного використання;
– розбити систему на підсистеми;
– виявити властиву завданню паралельність;
– розподілити підсистеми відповідно до апаратного забезпечення;
– установити пріоритети при компромісах;
– спланувати сховища даних;
– розподілити глобальні ресурси;
– вибрати стратегію управління програмним забезпеченням;
– обробити пограничні умови;
– установити пріоритети при компромісах;
– вибрати стиль архітектури.

<u>Оцінка продуктивності</u>

На початковому етапі планування нової системи необхідно провести наближену оцінку її продуктивності. Мета полягає не в тому, щоб отримати високу точність оцінки, а в тому, щоб

визначити, чи реально побудувати потрібну систему. Розрахунок повинен бути простим і ґрунтуватися на здоровому глузді.

Планування повторного використання

Повторне використання часто називається основною перевагою об’єктно-орієнтованої технології, але воно не забезпечується одним лише фактом застосування цієї технології. Повторне використання характеризується двома істотно різними аспектами: можна використовувати існуючі речі, а можна створювати нові з розрахунком на повторне використання. Набагато простіше використовувати існуюче, ніж проектувати нове, орієнтуючись на невідомі перспективи повторного використання. Звичайно, щоб ми зараз могли повторно використовувати якісь сутності, хтось у минулому повинен був їх створити. Суть у тому, що більшість розробників використовують існуюче, і лише невелика їх частина створює нове. Не думайте, що вам доведеться починати роботу з об’єктно-орієнтованими технологіями зі створення нових сутностей. Для цього потрібен великий досвід.

Повторно використаними можуть бути *моделі*, *бібліотеки*, *каркаси* та *шаблони*. Найбільш практичним можна назвати повторне використання моделей. Логіка моделі часто буває застосована до безлічі завдань.

Бібліотека (library) – це сукупність класів, які можуть виявитися корисними в безлічі контекстів. Ця сукупність повинна бути добре влаштована, щоб користувачі могли шукати потрібні їм класи. Щоб побудувати таку бібліотеку, потрібно багато зусиль, і часто буває важко вирішити, де має бути розміщений який-небудь об’єкт. У вирішенні цього завдання може допомогти мережевий пошук, але він не замінить хорошої організації бібліотеки. Крім того, класи повинні мати точний та докладний опис, щоб користувачі могли самі вирішити, підходять вони їм чи ні. Відзначаються наступні якості, якими повинні володіти «хороші» бібліотеки класів:

– цільність – бібліотека класів повинна охоплювати невелику кількість чітко визначених тем;

– повнота – бібліотека класів повинна повністю описувати поведінку за обраними темами;

– узгодженість – поліморфні операції повинні характеризуватися однаковими іменами та сигнатурами в різних класах;

– ефективність – бібліотека повинна надавати альтернативні реалізації алгоритмів (таких як алгоритми сортування), що дозволяють вибирати між швидкістю й обсягом пам’яті;

– розширюваність – користувач повинен мати можливість визначати підкласи класів бібліотеки;

– універсальність – скрізь, де це можливо, повинні використовуватися параметризовані визначення класів.

Каркас (framework) – це скелет програми, деталізація якого дозволяє створити повноцінний додаток. Деталізація найчастіше зводиться до конкретизації абстрактних класів поведінкою, специфічною для даного додатку. З каркасом може поставлятися бібліотека класів, завдяки наявності якої програміст може конкретизувати абстрактні класи, вибираючи відповідні підкласи з бібліотеки, а не програмуючи їх поведінку з нуля. Каркас складається не тільки з класів, він включає парадигму потоку управління та загальні інваріанти. Зазвичай каркаси створюються для цілої категорії додатків, а бібліотеки класів для них ще більш спеціалізовані і не підходять для вирішення спільних завдань.

Шаблон (pattern) – це апробоване вирішення загального завдання. Різні шаблони призначені для різних стадій циклу розробки програмного забезпечення. Існують шаблони для аналізу, архітектури, проектування та реалізації. Краще використовувати існуючі шаблони, ніж заново винаходити рішення з нуля. Шаблони зазвичай супроводжуються рекомендаціями щодо їх використання, а також характеристиками для порівняння.

Використання шаблонів дає безліч переваг. Одне з них полягає в тому, що шаблон розроблявся іншими людьми та успішно застосовувався ними на практиці. Отже, він з більшою ймовірністю виявиться коректним і стійким, ніж власне рішення, що не пройшло тестування. Крім того, працюючи з

шаблонами, ви вчите язик, знайомий багатьом іншим розробникам. Величезна кількість книг описує шаблони, тонкощі та нюанси роботи з ними. Можна розглядати шаблони як розширення мови моделювання. Не обов'язково мислити в термінах примітивів, можна оперувати їх комбінаціями. Шаблони – це фрагменти моделі, що виражають знання експертів.

Шаблон відрізняється від каркасу. Перший зазвичай складається з невеликої кількості класів, зв'язаних відносинами. Навпаки, каркас зазвичай містить на порядок більше класів і охоплює цілу підсистему або навіть додаток.

Шаблони поділяються на:

– шаблони розподілення обов'язків (*software design*);
– архітектурні (architectural) шаблони (*systems design*);
– шаблони проектування (*GoF*);
– ідіоми – idioms (*low level*);
– шаблони аналізу – analysis patterns (*recurring & reusable analysis models*);
– шаблони архітектури корпоративних програмних додатків (*patterns of Enterprise Application Architecture*);
– шаблони організації (*structure of organizations/projects*);
– шаблони процесів (*software process design*).

Розбиття системи на підсистеми

Для всіх додатків, за винятком найпростіших, перший етап проектування системи зводиться до розподілу цієї системи на частини. Кожна велика частина системи називається підсистемою. Кожна підсистема володіє чимось таким, що відрізняє її від інших підсистем, наприклад єдиною функціональністю, фізичним розміщенням або виконанням на однотипному обладнанні. Наприклад, комп'ютер космічного корабля може складатися з підсистем життєзабезпечення, навігації, управління двигунами та проведення космічних експериментів.

Підсистема (*subsystem*) – це не об'єкт і не функція, а сукупність класів, асоціацій, операцій, подій та обмежень, зв'язаних між собою і які мають добре визначений та не надто обширний інтерфейс з іншими системами. Підсистема зазвичай визначається сервісами, які вона надає.

Сервіс (*service*) – це група споріднених функцій, що мають загальне призначення, наприклад оброблення вводу-виводу, побудова малюнків і виконання арифметичних операцій. Підсистема виражає певний цілісний погляд на одну частину завдання. Наприклад, файлова система є підсистемою операційної системи. Вона складається з набору зв'язаних абстракцій, які практично не залежать від абстракцій в інших підсистемах (таких як управління пам'яттю і процесами).

Кожна підсистема зв'язана з іншими частинами системи добре визначеним інтерфейсом. *Інтерфейс* визначає форму всіх взаємодій та потоки інформації через границі підсистеми, але він не обмежує внутрішню реалізацію підсистеми. Завдяки цьому кожна підсистема може розроблятися незалежно від інших.

Підсистеми слід визначати таким чином, щоб взаємодії в системі в цілому носили переважно внутрішній характер, тобто здійснювалися всередині підсистем. Це скоротить залежності між різними підсистемами. Кількість підсистем не повинна бути великою: в ролі розумного обмеження можна взяти число 20. Кожна підсистема, у свою чергу, може бути розбита на підсистеми більш низького рівня.

Відносини між двома підсистемами можуть бути організовані за принципом клієнт-сервер або ж ці підсистеми можуть бути одноранговими. У першому випадку (відношення клієнт-сервер) клієнт викликає сервер, який виконує деякий сервіс і повертає результат. Клієнт повинен знати інтерфейс сервера, але серверу не обов'язково знати інтерфейси клієнта, оскільки ініціатором взаємодії завжди є клієнт.

В однорангових відносинах кожна підсистема може звернутися до будь-якої іншої. Взаємодія підсистем не обов'язково супроводжується негайною відповіддю. Однорангові взаємодії складніші, оскільки підсистеми повинні знати інтерфейси всіх інших підсистем. У них можуть виникати цикли, важкі для розуміння і проектування. Намагайтеся розбивати систему на клієнтські та серверні частини завжди, коли це можливо, тому що односторонню взаємодію набагато простіше проектувати, розуміти та змінювати, ніж двосторонню.

Розбиття системи на підсистеми може бути організовано у вигляді послідовності *горизонтальних шарів* або *вертикальних відділів*.

Багатошарова система (*layered system*) – це впорядкована безліч віртуальних шарів (ярусів), кожен з яких будується в термінах шарів, що знаходяться нижче нього, і утворює базис для реалізації вищих шарів. Об'єкти кожного шару можуть бути незалежними, але часто між об'єктами різних шарів спостерігається деяка відповідність. Знання в даному випадку є односторонніми: підсистема знає про нижні шари, але не про верхні. Між верхніми і нижніми шарами є відношення клієнт-сервер.

Наприклад, в інтерактивній графічній системі вікна будуються за допомогою операцій з екраном, що складаються з операцій з окремими пікселами, які зводяться до операцій введення-виведення з пристроями. Кожний шар може мати свій набір класів і операцій. Кожний шар реалізується в термінах класів та операцій нижчих шарів.

Багатошарова архітектура буває двох видів: відкрита і закрита. У закритій архітектурі кожний шар будується тільки в термінах того рівня, який знаходиться безпосередньо під ним. Це скорочує залежності між шарами та полегшує внесення змін, оскільки інтерфейс шару впливає тільки на наступний за ним шар. У відкритій архітектурі шар може використовувати функції будь-якого нижчого рівня. Це скорочує потребу в перевизначенні операцій на кожному рівні, завдяки чому код може вийти більш ефективним і компактним. Однак відкрита архітектура не сприяє прихованню інформації. Зміни в підсистемі можуть вплинути на будь-яку вищу за підпорядкуванням підсистему, тому така архітектура виходить менш стійкою, ніж закрита. Обидва типи корисні. Проектувальнику доводиться зважувати переваги і недоліки кожної з них.

Зазвичай у постановці завдання визначаються тільки верхній і нижній рівні. Верхній рівень – це потребуєма система, а нижній – це доступні ресурси (обладнання, операційна система, існуючі бібліотеки). Якщо різниця між ними дуже велика (а так буває досить часто), вам доводиться вводити додаткові

рівні для скорочення концептуальних розривів між сусідніми рівнями.

Багатошарову систему можна перенести на інше обладнання, переписавши тільки один з її рівнів. Потрібно намагатися вводити принаймні один рівень абстрагування між додатком і будь-якими сервісами, що надаються операційною системою чи обладнанням. Додайте рівень класів інтерфейсу, що надають логічні сервіси, і відобразіть їх на конкретні системно-залежні сервіси.

Розділи (partitions) ділять систему по вертикалі на кілька незалежних або слабко зв'язаних між собою підсистем, кожна з яких надає сервіси одного типу. Наприклад, до складу комп'ютерної операційної системи входять файлова система, диспетчер процесів, система управління віртуальною пам'яттю та система керування пристроями. Підсистеми можуть знати про існування інших підсистем, але це знання не є глибоким і не створює серйозних залежностей на рівні проектування.

Різниця між шарами та розділами полягає в тому, що шари відрізняються один від одного рівнем абстракції. Розділи ж просто ділять систему на частини, що знаходяться на одному і тому ж рівні абстракції. Ще одна відмінність полягає в тому, що шари обов'язково залежать один від одного (зазвичай за типом клієнт-сервер). Розділи утворюють однорангову систему і є відносно незалежними або взаємозалежними.

Ви можете розбити систему на підсистеми, поєднуючи шари та розділи. Рівні можуть розбиватися на розділи, а розділи – на рівні.

Після визначення підсистем вищого рівня необхідно показати інформаційні потоки. Іноді всі підсистеми взаємодіють між собою, але частіше за все структура потоків виявляється більш простою. Наприклад, обчислення часто організуються у вигляді конвеєра (прикладом такої структури є компілятор). Інші системи мають структуру зірки, в якій головна підсистема керує всіма взаємодіями інших підсистем. Скрізь, де це можливо, слід прагнути спрощувати топологію і скорочувати кількість взаємодій між підсистемами.

Виділення паралелізму

В аналітичній моделі, у реальному світі та в апаратному забезпеченні всі об'єкти діють паралельно один з одним. Однак у програмній реалізації системи не всі об'єкти виявляються паралельними, тому що один процесор може і не забезпечувати одночасне виконання багатьох об'єктів. На практиці ви можете реалізувати кілька об'єктів на одному процесорі тільки в тому випадку, якщо ці об'єкти не можуть бути активними одночасно. Одним із важливих завдань етапу проектування системи є виділення об'єктів, що повинні бути активними одночасно, а також об'єктів, активність яких є взаємовиключною. Об'єкти другої групи можна об'єднати в один потік управління (завдання).

Основним засобом для пошуку паралелізмів є модель станів. Два об'єкти є невід'ємно паралельними, якщо вони можуть одночасно отримувати інформацію про події, не взаємодіючи один з одним. Якщо події не синхронізовані одна з одною, ви не можете об'єднати ці об'єкти в один потік управління. Наприклад, підсистеми управління двигуном і крилами літака повинні працювати паралельно (хоч і не повністю незалежно одне від одного). Звичайно, найкраще працювати з незалежними підсистемами, тому що їх можна розподілити по різних апаратних пристроях без будь-яких витрат на взаємодію.

Не обов'язково реалізовувати невід'ємно паралельні підсистеми у вигляді різних апаратних пристроїв. Найчастіше об'єкти, що мають бути реалізовані у вигляді окремих апаратних пристроїв, перераховуються в описі завдання.

Всі об'єкти з концептуальної точки зору є паралельними, однак на практиці об'єкти часто залежать один від одного. Вивчаючи діаграми станів окремих об'єктів і взаємодію між ними, що здійснюється за допомогою подій, можна об'єднувати групи об'єктів в один потік управління. ***Потік керування*** (*thread of control*) – це маршрут, що проходить через кілька діаграм станів, на якому лише один об'єкт може бути активним у конкретний момент часу. Потік залишається всередині однієї діаграми станів до тих пір, поки об'єкт не

пошле подію іншому об'єкту та не перейде в стан очікування іншої події. Тоді потік управління переходить до одержувача події і залишається у нього до тих пір, поки не повернеться до вихідного об'єкту разом з новою подією. Потік може розділитися в тому випадку, якщо об'єкт відправить подію та продовжить своє виконання.

Тільки один об'єкт може бути активним в кожному потоці управління в конкретний момент часу. Потоки управління в комп'ютерних системах реалізуються у вигляді задач.

Розподіл підсистем

Паралельні підсистеми повинні бути розподілені по апаратних пристроях: універсальним процесорам або спеціалізованим функціональним блокам. Для цього потрібно:

– оцінити потреби кожної підсистеми в обчислювальних ресурсах і обсяг ресурсів, необхідний для задоволення цих потреб;

– вибрати апаратну чи програмну реалізацію підсистеми;

– розподілити програмні підсистеми по процесорах з урахуванням вимог, що пред'являються до продуктивності, так щоб мінімізувати взаємодію між процесорами;

– визначити зв'язність фізичних модулів, що реалізують підсистеми.

Рішення використовувати кілька процесорів чи інших апаратних модулів приймається на підставі відомостей про потреби в апаратних ресурсах, що перевищують можливості одного процесора. Кількість процесорів залежить від обсягу обчислень і швидкості комп'ютера.

Проектувальник системи повинен оцінити необхідну потужність процесора, визначивши стаціонарну нагрузку (добуток кількості транзакцій у секунду на час оброблення транзакції). Наближене значення часто виявляється неточним. Буває корисно провести кілька експериментів. Оціночне значення потрібно дещо завищити, щоб врахувати перехідні ефекти, зв'язані з випадковими флуктуаціями навантаження та синхронізованими сплесками активності. Надлишкова ємність залежить від прийнятної частоти відмов, що викликані

нестачею ресурсів. Ви повинні враховувати не тільки стаціонарну (середню) навантаження, але й пікову.

Об'єктно-орієнтований підхід зручний для аналізу апаратних пристроїв. Кожен пристрій – це об'єкт, що існує паралельно з іншими об'єктами (пристроями або програмами). Ви повинні розподілити підсистеми по апаратних пристроях і програмних комплексах. Існує дві причини, з яких певна підсистема може бути реалізована у вигляді апаратного пристрою.

Перша – вартість. Існуюче обладнання повинно надавати необхідну функціональність. Інша – продуктивність. Система вимагає більш високої продуктивності, ніж може надати універсальний процесор, і при цьому існує більш ефективне обладнання.

Складність проектування системи зв'язана з тим, що ви повинні виконати всі зовнішні вимоги до обладнання та програмного забезпечення. Об'єктно-орієнтоване проектування – це не чарівна паличка, що може вирішити всі проблеми такого роду, проте воно допомагає моделювати зовнішні пакети. Потрібно враховувати питання сумісності, вартості і продуктивності. Крім того, потрібно зробити систему гнучкою в розрахунку на зміни в майбутньому (як проектні зміни, так і щодо вдосконалення готового продукту). Гнучкість вимагає певних витрат, і саме архітектор повинен вирішувати, скільки він готовий на неї витратити.

Проект системи повинен описувати розподіл програмних підсистем по процесорах. Існує кілька причин, через які цей розподіл має бути формалізовано:

1. Логістика. Деякі завдання повинні виконуватися в певних місцях, тому що вони зв'язані з керуванням, обладнанням або вимагають паралельного виконання. Наприклад, на робочій станції інженера повинна бути встановлена своя власна операційна система, щоб оператор міг працювати з нею в разі перебоїв у мережі.

2. Обмеження на взаємодію. Час відгуку та потік даних перевищують доступну полосу пропускання між задачею та

апаратним пристроєм. Наприклад, високошвидкісні графічні пристрої вимагають установлення власних контролерів, тому що вони породжують великі обсяги даних.

3. Обмеження на обчислювальні ресурси. Якщо потреби в обчислювальних ресурсах занадто великі, щоб їх міг задовольнити один процесор, можна встановити кілька процесорів. Витрати на взаємодію можна скоротити, призначивши задачі, що активно обмінюються даними, одному процесору. Незалежні підсистеми слід розподіляти по різним процесорам.

Визначивши види та кількість апаратних пристроїв, необхідно описати їх розташування і з'єднання між ними. Це містить наступні види робіт:

1. Вибрати топологію для з'єднання фізичних пристроїв. Фізичним з'єднанням часто відповідають асоціації моделі класів. Відносини клієнт-сервер також реалізуються у вигляді фізичних з'єднань. Деякі з'єднання можуть бути опосередкованими. Вартість найбільш важливих відносин слід мінімізувати.

2. Вибрати топологію дублюючих один одного блоків. Якщо потрібно підвищити продуктивність, включивши в систему кілька однотипних модулів, то необхідно вказати і їх топологію. Модель класів тут не допоможе, тому що дублювання блоків зазвичай застосовується для оптимізації на етапі проектування. Топологія дублюючих блоків зазвичай має регулярну структуру, наприклад: лінійна послідовність, матриця, дерево, зірка. Потрібно врахувати очікувані шаблони надходження даних і запропонувати паралельні алгоритми їх оброблення.

3. Вибрати форму каналів комунікації та протоколи взаємодії. Точні інтерфейси модулів на етапі проектування враховувати не потрібно, але загальні механізми взаємодії потрібно вказати. Наприклад, взаємодія може бути синхронною, асинхронною або блокуючою. Потрібно оцінити пропускну здатність і затримку каналів комунікації та вибрати найбільш відповідні типи з'єднань.

Навіть якщо з'єднання є логічними, а не фізичними, їх все одно треба розглянути. Наприклад, блоки можуть являти собою завдання, що виконуються в рамках однієї операційної системи і зв'язані між собою засобами міжпроцесної взаємодії (IPC). У більшості операційних систем виклики IPC виконуються набагато повільніше, ніж виклики підпрограм всередині однієї задачі, тому їх використання в задачах з жорсткими часовими обмеженнями може бути непрактично. У цьому випадку вам доведеться об'єднати жорстко зв'язані задачі в одну і реалізувати з'єднання як виклики підпрограм.

Керування сховищами даних

Існує декілька способів зберігання даних, що можуть використовуватися незалежно чи спільно: структури даних, файли і бази даних. Різні варіанти зберігання володіють різними сполученнями вартості, часу доступу, ємності й надійності. Наприклад, додаток на персональному комп'ютері може працювати зі структурами даних у пам'яті з файлами. Система бухгалтерського обліку може використовувати базу даних для взаємодії підсистем. Файли – це дешевий, простий і надійний засіб збереження даних. Однак операції з файлами є низькорівневими, тому в додаток доводиться включати додатковий код, що забезпечує перехід на необхідний рівень абстракції. В різних операційних системах файли реалізуються по-різному, тому додатки, що будуть переноситись, потребують акуратної ізоляції рівня файлової системи. Реалізації файлів з послідовним доступом стандартизовані найкраще, а ось команди та формати зберігання файлів з довільним доступом сильно залежать від системи. Нижче перераховані типи даних, що найкраще зберігати у файлах:

– дані великого обсягу та низької інформаційної щільності (архівні файли, записи журналів);

– середні за обсягом дані з простою структурою;

– дані, доступ до яких здійснюється послідовно;

– дані, що можна повністю завантажити у пам'ять.

Ще один варіант збереження даних – бази даних, керовані системами управління базами даних (СУБД). СУБД (у

тому числі реляційні та об'єктно-орієнтовані) випускаються різними виробниками. Вони можуть кешувати дані, що часто використовуються, в пам'яті для оптимізації вартості та продуктивності оперативної пам'яті та дискового простору. Бази даних полегшують перенесення додатків на інше обладнання та на інші платформи, тому що перенесення коду СУБД здійснюється його постачальником. Одним з недоліків СУБД є їх складний інтерфейс. Багато мов баз даних погано інтегруються з мовами програмування. Нижче перераховані типи даних, що найкраще зберігати в базах даних:

– дані, що вимагають детального оновлення безліччю користувачів;

– дані, з якими повинні працювати кілька додатків;

– дані, що вимагають координованого оновлення за допомогою транзакцій;

– великі обсяги даних, що вимагають ефективної обробки;

– дані, що потребують тривалого зберігання та мають велику цінність для організації;

– дані, що повинні бути захищені від несанкціонованого доступу та від зловмисників.

Об'єктно-орієнтовані бази даних не змогли завоювати популярність на широкому ринку. Тому їх слід розглядати тільки в тому випадку, якщо ви розробляєте особливий додаток, що працює з даними безлічі типів або потребує звернення до низькорівневих примітивів для керування даними. До таких додатків відносяться інженерні, мультимедійні додатки, бази знань та електронні пристрої з вбудованим програмним забезпеченням. Для більшості додатків досить реляційної бази даних. Такі бази домінують на ринку, їх функціональність цілком достатня для більшості додатків. При правильному використанні вони дозволяють дуже добре реалізувати об'єктно-орієнтовану модель.

Розподіл глобальних ресурсів

Проектувальник системи повинен вказати глобальні ресурси та визначити механізми управління доступу до них. Глобальні ресурси бувають декількох видів:

– фізичні пристрої: процесори, накопичувачі, а також супутники зв'язку;

– простір: дисковий простір, екран робочої станції, кнопки миші;

– логічні назви: ідентифікатори об'єктів, імена файлів, назви класів;

– доступ до загальних даних: бази даних.

Якщо ресурс являє собою фізичний об'єкт, то він може самостійно контролювати доступ до себе за допомогою протоколу керування доступом. Якщо ж ресурс являє собою логічну сутність, то виникає небезпека конфлікту доступу. Наприклад, один і той же ідентифікатор об'єкта може бути одночасно використаний паралельними завданнями.

Уникнути такого конфлікту можна за допомогою «охоронного» об'єкту, якому буде належати кожен глобальний ресурс і який буде керувати доступом до свого ресурсу. Один охоронний об'єкт може керувати кількома загальними ресурсами. Доступ до будь-якого загального ресурсу можливий тільки через його охоронний об'єкт. Віднесення глобального ресурсу до певного об'єкта – це визнання індивідуальності ресурсу.

Ви можете виконати логічне розбиття ресурсу, розподіливши його частини по різним охороняючим об'єктам, завдяки чому буде реалізований паралельний доступ до частин ресурсу. Наприклад, у паралельному розподіленому середовищі, однією зі стратегій виділення ідентифікаторів об'єктів, є попередній розподіл діапазонів можливих ідентифікаторів між процесорами, що входять до складу мережі. Кожен процесор виділяє ідентифікатори з наданого йому діапазону, завдяки чому зникає потреба у глобальній синхронізації.

У додатках, що ставлять жорсткі вимоги до тимчасових обмежень, вартість доступу до ресурсу через охоронний об'єкт іноді виявляється занадто висока. Клієнти в таких системах повинні працювати з ресурсом напряму. У цьому випадку блокування може бути встановлено на окремі частини ресурсу. *Блокування* – це логічний об'єкт, зв'язаний з деякою підмножиною ресурсу, що дає власникові блокування право на безпосередній доступ до ресурсу. Охороняючий об'єкт, в цьо-

му випадку теж повинен існувати, він займається видачею та зняттям блокування, однак після однієї операції взаємодії з охороним об'єктом для отримання блокування, клієнт може працювати з ресурсом напряму. Такий підхід небезпечніший, тому що кожен користувач ресурсу сам відповідає за свою поведінку під час роботи з цим ресурсом. Не використовуйте прямий доступ до загальних ресурсів, якщо є інші варіанти.

Вибір стратегії керування програмним забезпеченням

В аналітичній моделі взаємодії уявляються як події, що передаються між об'єктами. Апаратне забезпечення в цьому відношенні близько аналітичній моделі, а от у програмному забезпеченні керування може бути реалізовано кількома способами. Не обов'язково, щоб усі підсистеми використовували одну й ту саму реалізацію, але краще всього вибирати для всієї системи єдиний стиль керування. Існує два основних варіанти керування програмною системою: зовнішнє керування та внутрішнє керування.

Зовнішнє керування (*external control*) – це потік видимих ззовні подій між об'єктами системи. Існує три варіанти керування, здійснюваного через зовнішні події: процедурне послідовне, подієве послідовне і паралельне. Оптимальний вид керування залежить від доступних ресурсів (надаються мовою програмування і операційною системою), а також від видів взаємодії всередині додатку.

Внутрішнє керування (internal control) – це потік керування всередині процесу. Воно з'являється тільки на етапі реалізації, а тому не можна сказати, що йому від початку притаманні паралельність або послідовність. Проектувальник може розбити процес на кілька задач для забезпечення логічної ясності або для підвищення продуктивності (у разі наявності декількох процесорів). На відміну від зовнішніх подій внутрішні переходи управління (наприклад, виклики процедур або виклики між задачами) знаходяться під контролем програми та можуть бути структуровані так, як зручно. Широко поширені три типи операцій передання керування: виклики процедур, квазіпаралельні виклики між зада-

чами та паралельні виклики між задачами. *Квазіпаралельні завдання* (співпрограми чи програмні потоки) – це конструкції, розроблені для зручності програміста. Кожний програмний потік має власний стек та адресний простір, але тільки один з них може бути активним у конкретний момент часу.

У послідовній системі з *процедурним керуванням* хід виконання визначається кодом програми. Процедури запитують зовнішні дані та чекають на їх надходження. Коли вхідні дані надходять до системи, виконання програми поновлюється з тієї процедури, що запрошувала ці дані. Значення лічильника команд, вміст стека викликів процедур і значення локальних змінних у цьому випадку визначають стан системи.

Основна перевага процедурного керування полягає в тому, що його легко реалізувати в більшості широко поширених мов. Недолік же в тому, що цей варіант потребує відображення Властивої об’єктам паралельності на послідовний потік управління. Проектувальник повинен перетворювати події в операції між об’єктами. Зазвичай операція відповідає парі подій: події виведення, що виконує виведення та запитує введення, і події введення, що доставляє до системи нові значення. Ця парадигма незручна для опису асинхронного введення, тому що програма має явно запитувати вхідні дані. Процедурне управління годиться тільки в тому випадку, якщо модель станів демонструє регулярне чергування подій введення та виведення. Більш гнучкі інтерфейси користувача та системи керування побудувати на основі цієї парадигми досить складно.

Зверніть увагу, що основні об’єктно-орієнтовані мови, такі як *C ++* і *Java,* є процедурно-орієнтованими. Нехай вас не вводить в оману словосполучення «передача повідомлень». *Повідомлення* – це виклик процедури з вбудованим оператором case, що залежать від класу цільового об’єкта. Основний недолік звичайних об’єктно-орієнтованих мов у тому, що вони не підтримують властиву об’єктам паралельність. Існують і паралельні об’єктно-орієнтовані мови, але вони поки поширені недостатньо широко.

У послідовній системі, керованій подіями, контроль здій-

снює диспетчер або монітор, що надається мовою чи операційною системою або виконаний у вигляді окремої підсистеми. Розробники зв'язують процедури-додатки з подіями, і диспетчер викликає ці процедури, коли відбуваються відповідні їм події. Процедури передають диспетчеру вихідні дані або дозволяють введення вхідних даних, але вони не блокуються в очікуванні їх надходження. Всі процедури повертають управління диспетчеру (а не зберігають його до тих пір, поки не будуть отримані будь-які дані). Тому стан системи не може бути описано лічильником програми і стеком. Процедури повинні використовувати глобальні змінні для збереження інформації про стан або ж локальна інформація про стан повинна підтримуватися диспетчером. Подієве керування реалізувати на стандартних мовах складніше, ніж процедурне, але результат у багатьох випадках вартий витрачених зусиль.

Керовані подіями системи виходять більш гнучкими, ніж процедурні. Такі системи моделюють співробітництво процесів усередині однієї багатопоточної задачі. Неправильно написана процедура може заблокувати всі додатки, тому розробникам доводиться бути уважними. Особливо зручними виходять подієві підсистеми інтерфейсу користувача.

Використовуйте керовану подіями систему для зовнішнього керування замість процедурної скрізь, де це можливо, тому що перетворення подій в програмні конструкції в цій парадигмі виконувати простіше. Такі системи виходять «більш модульними» і краще обробляють аварійні ситуації, ніж процедурні системи.

У *паралельній системі* управління здійснюється кількома незалежними об'єктами паралельно. Кожний з таких об'єктів являє собою окрему задачу. У цій системі події реалізуються як односторонні повідомлення (не такі, як «повідомлення» в об'єктно-орієнтованих мовах) між об'єктами. Задача може очікувати введення даних, але інші задачі при цьому продовжують виконуватися. Операційна система планує виконання задач і надає механізм постановки подій в чергу, тому події не губляться в тому випадку, якщо задача у момент їх надходження зайнята чимось іншим. У разі наявності декількох

процесорів різні задачі виконуються дійсно паралельно.

Внутрішнє керування. У процесі проектування розробник розкриває операції на об'єктах у вигляді послідовностей операцій більш низького рівня на тих же самих чи інших об'єктах. Внутрішні взаємодії об'єктів багато в чому схожі зі зовнішніми взаємодіями, тому що для них можуть використовуватися ті ж самі механізми реалізації. Однак існує й важлива відмінність: зовнішні взаємодії завжди мають на увазі очікування подій, тому що об'єкти не залежать один від одного і не можуть змусити один одного відповідати саме в потрібний момент. Внутрішні операції об'єктів являють собою частину алгоритму реалізації, тому їх відгук може бути передбаченим. Отже, внутрішні взаємодії часто можна розглядати у вигляді викликів процедур: активний об'єкт надсилає запит і чекає на нього відповіді. Існують алгоритми паралельної обробки, проте дуже багато обчислювальних задач добре вирішуються послідовними алгоритмами та можуть бути укладені в єдиний потік керування.

Існують і інші парадигми. Наприклад, бувають системи, засновані на правилах, системи логічного програмування й інші форми програм, що не мають процедур. Вони висловлюють інший стиль керування: явне керування замінюється декларацією тверджень і неявними правилами обчислень, що можуть бути недетермінованими або складними. На сьогодні такі мови використовуються у вузьких областях, таких як штучний інтелект та програмування баз знань, але з часом вони можуть почати застосовуватися ширше.

Облік граничних умов

Проект системи здебільшого визначається її поведінкою в стаціонарному режимі, проте обов'язково потрібно розглянути пограничні умови та врахувати такі моменти, як ініціалізацію, завершення та відмову системи.

Система повинна перейти зі статичного початкового стану в прийнятний стаціонарний стан. Вона повинна виконати ініціалізацію констант, параметрів, глобальних змін, задач, охороняючі об'єкти та, можливо, навіть саму ієрархію класів. У

процесі ініціалізації лише невелика частина функціональності системи зазвичай буває доступною. Особливо складна ініціалізація системи з паралельними задачами, тому що на цьому етапі незалежні об'єкти не повинні піти далеко вперед або відстати від усіх інших.

Завершення зазвичай виявляється простішим ініціалізації, тому що більшість об'єктів можна просто знищити. Задача повинна звільнити всі зарезервовані нею зовнішні ресурси. У паралельній системі задача повинна повідомляти про своє завершення всі інші задачі.

Відмова – це незаплановане завершення системи. Відмова може бути викликана помилками користувача, проблемами системних ресурсів або зовнішньою поломкою. Проектувальник системи повинний враховувати можливість її відмови. Відмова може бути викликана й помилками всередині системи та часто виявляється як «неможливе» протиріччя. В ідеальній системі таких помилок бути не може, проте хороший проектувальник повинен передбачити можливість коректного завершення у разі фатальних помилок. Система повинна залишити середовище виконання якомога більш «чистим» і зберегти якомога більше інформації про причини відмови перед тим, як остаточно завершити роботу.

<u>Встановлення пріоритетів</u>

Проектувальник системи повинен установити пріоритети, якими потрібно буде керуватися на наступних етапах розробки. Пріоритети вирішують конфлікти між бажаними, але несумісними цілями. Наприклад, частіше за все роботу системи можна прискорити, додавши до неї пам'яті, але при цьому вона буде споживати більше енергії та коштувати дорожче. Компроміси зв'язані не тільки з самою програмою, але і з процесом її розроблення. Іноді буває потрібно пожертвувати повнотою функціональності для того, щоб у потрібний момент випустити продукт на ринок. Іноді пріоритети вказуються при постановці задачі, але частіше відповідальність за суміщення несумісного лягає на проектувальника.

Проектувальник повинен визначити відносну важливість різних критеріїв, на підставі чого можна буде надалі йти на компроміси. Проектувальник не зобов'язаний приймати всі рішення відразу, він просто встановлює правила, за якими ці рішення будуть прийматися.

Рішення проектувальника впливають на систему в цілому. Успіх чи провал кінцевого продукту може залежати від правильності вибору цілей. Якщо не встановити пріоритети в масштабі всієї системи, окремі її частини можуть бути оптимізовані за різними параметрами (субоптимізація), в результаті чого система буде даремно витрачати ресурси. Навіть у невеликих проектах програмісти часто забувають про справжні цілі і починають турбуватися про «ефективність» тоді, коли насправді вона не має принципового значення.

Встановлення пріоритетів для прийняття рішень найчастіше дає досить туманні результати. Не слід очікувати точних чисел, як то: швидкість 53 %, пам'ять 31 %, переносимість 15 %, вартість 1 %. Пріоритети рідко виявляються абсолютними. Якщо швидкість виконання важливіша, ніж пам'ять, це не означає, що будь-який приріст швидкості, навіть невеликий, може виправдати необмежене збільшення пам'яті, що витрачається. Не можна навіть скласти повний список критеріїв прийняття рішень. Пріоритети – це твердження філософії проекту. Реальне прийняття рішень на подальших етапах все одно потребує оцінок та інтерпретації поставлених цілей.

7.2 Поширені архітектурні стилі

Серед існуючих систем широко поширені кілька архітектурних стилів, що можуть послужити вам як прототипи для побудови ваших додатків. Кожний стиль оптимальний для системи певного типу.

Коротко розглянемо такі різновиди стилів:

– пакетне перетворення (*batch transformation*) – одноразове перетворення всіх вхідних даних;

– безперервне перетворення (*continuous transformation*) – перетворення змінного вхідного сигналу, що виконується безперервно;

– інтерактивний інтерфейс (*interactive interface*) – система, в якій домінують зовнішні взаємодії;

– динамічне моделювання (*dynamic simulation*) – система, що моделює розвиваючі об'єкти реального світу;

– система реального часу (*real-time system*) – система, що відповідає жорстким тимчасовим обмеженням;

– адміністратор транзакцій (*transaction manager*) – система, що займається збереженням і оновленням даних, часто з підтримкою паралельного доступу з різних фізичних точок.

Звичайно, це не повний список відомих систем та архітектур. Для деяких задач потрібно придумувати абсолютно нову архітектуру, але в більшості випадків можна обійтися існуючим стилем або якоюсь з його модифікацій. Багато задач поєднують різні аспекти декількох архітектур.

Пакетне перетворення складається з декількох обчислювальних процесів. Програма отримує вхідні дані і має обчислити результат. Ніякої взаємодії з зовнішнім світом у проміжку не передбачається. Як приклади, можна привести стандартні обчислювальні завдання: компілятори, програми для розрахунку бухгалтерських відомостей, програми автоматичного проектування та багато інших. Для таких завдань модель станів тривіальна або зовсім вироджена. Модель класів достатньо важлива: вона описує введення, виведення і проміжні стадії обчислень. Модель взаємодії документує обчислення та зв'язує між собою моделі класів. Найбільш важливим аспектом рішення задачі є визначення чіткої послідовності етапів.

Пакетне перетворення проектується у наступній послідовності: розбити перетворення на етапи, кожен з яких полягає у виконанні частині перетворення; підготувати моделі класів для вводу, виводу та проміжних даних. Кожний етап враховує тільки ті уявлення, що зв'язують його з сусідами; деталізувати кожний етап до тих пір, поки його реалізація не стане очевидною; оптимізувати отриманий конвеєр.

Безперервне перетворення – це характеристика системи, вихідний сигнал якої залежить від змінних сигналів на входах. На відміну від пакетного перетво-

рення, яке обчислює результат один раз, безперервне перетворення повинно оновлювати вихідні сигнали досить часто (теоретично – безперервно, проте на практиці обчислення здійснюється з великою частотою дискретизації). Із-за жорстких тимчасових обмежень система не може обчислювати весь набір вихідних даних заново кожного разу при зміні вхідного сигналу (інакше це було б просто пакетне перетворення).

Замість цього система повинна розраховувати збільшення вихідних даних. В ролі типових прикладів програм цього класу можна навести: обробники сигналів, віконні системи, інкрементні компілятори та системи контролю виробничих процесів. Для цих систем моделі класів, станів і взаємодії повинні бути приблизно такими ж, як і для пакетних перетворень.

Безперервне перетворення проектується у наступній послідовності: розбити перетворення на етапи, кожен з яких повинен описувати будь-яку частину перетворення; визначити моделі вхідних, вихідних і проміжних даних, як для пакетного перетворення; продиференціювати кожну операцію так, щоб працювати з прирощеннями; додати оптимізуючі проміжні об'єкти.

Інтерактивний інтерфейс – це система, в якій домінують взаємодії з зовнішніми агентами (людьми або пристроями). Зовнішні агенти не залежать від системи, тому вона не може їх контролювати, а може лише запитувати в них уведення даних. Інтерактивний інтерфейс звичайно становить лише частину деякого додатку, яка часто може розглядатися незалежно від обчислень. Як приклад інтерактивної системи можна навести: інтерфейс запитів, заснований на формах, віконний менеджер робочої станції або панель управління моделюванням.

Найбільш важливі аспекти інтерактивного інтерфейсу – це протокол взаємодії системи з зовнішніми агентами, синтаксис можливих взаємодій, подання вихідних даних (варіанти відображення на екрані), потік керування всередині системи, продуктивність і обробка помилок. Інтерактивні інтерфейси визначаються головним чином моделлю станів. Модель кла-

сів описує елементи взаємодії (вхідні і вихідні маркери, формати подання). Модель взаємодії описує взаємодію діаграм станів.

Інтерактивний інтерфейс проектується у такій послідовності: ізолювати класи інтерфейсу від класів додатку; використовувати зумовлені класи для опису взаємодії із зовнішніми агентами скрізь, де це можливо; використовуйте модель станів як структуру програми (інтерактивні інтерфейси найкраще реалізуються з використанням паралельного керування або подієвого керування); ізолюйте фізичні події від логічних; повністю опишіть функції програми, що викликаються інтерфейсом; переконайтеся, що в моделі є достатньо інформації для їх реалізації.

Динамічне моделювання (dynamic simulation) – це моделювання, або відстеження, об'єктів реального світу. Як приклади можна привести моделювання траєкторій руху молекул, розрахунок траєкторій космічних кораблів, економічні моделі та відеоігри. Моделювання найпростіше проектувати з використанням об'єктно-орієнтованих технологій. Об'єкти та операції беруться безпосередньо з моделі додатку. Управління може бути реалізоване двома способами: зовнішній керівний об'єкт може імітувати кінцевий автомат, або ж об'єкти можуть обмінюватися повідомленнями між собою, як це відбувається в реальному світі.

На відміну від інтерактивної системи, внутрішні об'єкти при динамічному моделюванні відповідають об'єктам реального світу, тому модель класів у таких завданнях має істотне значення та часто виходить досить складною. Моделі станів і взаємодії також важливі.

Динамічна модель проектується у такій послідовності: ідентифікувати активні об'єкти реального світу в моделі класів, ці об'єкти мають періодично оновлені атрибути; ідентифікувати дискретні події; ідентифікувати безперервні залежності.

Зазвичай модель управляється циклом з невеликим кроком за часом. Дискретні події між об'єктами також часто включаються в цей цикл.

Зазвичай найскладніше при динамічному моделюванні – забезпечити адекватну продуктивність. В ідеальному світі довільна кількість паралельних процесорів могла б виконувати моделювання в точній відповідності з тим, як події розвиваються в реальному світі. На практиці проектувальнику доводиться оцінювати обчислювальну вартість кожного циклу та забезпечувати систему достатніми обчислювальними ресурсами. Безперервні процеси доводиться апроксимувати дискретними шагами.

Система реального часу – це інтерактивна система з жорсткими тимчасовими обмеженнями на час відгуку. Системи реального часу бувають жорсткими та гнучкими. Жорсткі – це життєво важливі додатки, що вимагають гарантованого відгуку в заданий час. Гнучкі системи реального часу теж повинні бути високонадійними, але для них допустиме рідкісне порушення обмежень. Типові додатки реального часу – це управління процесами, зчитування даних, комунікаційні пристрої, керуючі пристрої та ін.

Проектування систем реального часу – складне завдання, що вимагає вирішення таких підзавдань, як оброблення переривань, розподіл за пріоритетами та координація роботи безлічі процесорів. На жаль, системи реального часу часто використовуються в режимах, близьких до гранично припустимих, тому для досягнення необхідної продуктивності доводиться змінювати проект системи. У результаті втрачається переносимість та зручність обслуговування системи.

Адміністратор транзакцій – це система, основне призначення якої полягає в зберіганні та видачі даних. Більшість адміністраторів транзакцій мають справу з безліччю користувачів, що одночасно зчитують і записують дані. Адміністратор повинен забезпечувати захист даних від несанкціонованого доступу і від випадкових збоїв. Адміністратори транзакцій часто реалізуються у вигляді надбудови над системою управління базами даних (СУБД). СУБД надає універсальну функціональність для управління даними, яка може бути використана повторно. Як приклади адміністраторів транзакцій можна навести

системи бронювання авіаквитків, контролю інвентарю та виконання замовлень.

У таких системах домінує модель класів. Модель станів буває важливою в окремих випадках, зокрема для опису еволюції об'єкта, а також обмежень і методів, що застосовуються в різні моменти часу. Модель взаємодії буває суттєвою досить рідко.

Адміністратор транзакцій проектується у наступній послідовності: перетворити модель класів до структур бази даних; визначити паралельні модулі – такі, які не можуть використовуватися спільно; визначити одиницю транзакції – набір ресурсів, що одночасно задіяні в одній транзакції; виконати проектування паралельного управління транзакціями.

7.3 Проектування системи на основі компонентів

В останні роки спостерігається підвищена увага засобам створення програмного забезпечення на базі компонентів. Це пояснюється тим, що використання компонентів дозволяє прискорити процес розробки. Незважаючи на те, що компонентний похід до створення програм тісно пов'язаний з об'єктно-орієнтованим підходом, компоненти дозволяють уявляти проблеми та їх вирішення на більш високому рівні порівняно з об'єктами.

Компонент – це фізична частина системи, фрагмент реалізації, програма. Компоненти найчастіше сприймаються як файли, що виконуються. Але компонента може бути також частиною системи, що не є модулем, що безпосередньо виконується (наприклад файл вихідних даних, бібліотека).

Компоненти мають наступні властивості:

– являє собою незалежний програмний блок, що розгортається (вона ніколи не розгортається частково);

– може служити будівельним блоком для стороннього розроблювача (вона достатньо незалежна та документована, щоб її могли використати в інших компонентах);

– має тупикових станів;

– це частина системи, яка може бути замінена іншим компонентом, що узгоджується з тим же інтерфейсом;

– виконує чітку функцію і з логічної і з фізичної точки зору утворює єдине ціле;

– компонента може бути вкладена до інших компонентів.

Подібно до класів компоненти реалізують інтерфейси. Але по-перше компонента це фізична абстракція, що розгортається на певному комп'ютерному вузлі. Клас являє собою логічну сутність, що для того, щоб діяти як фізична абстракція, повинна бути реалізована за допомогою компонента.

По-друге компонента показує тільки деякі інтерфейси класів, що містяться в ній. Багато інших інтерфейсів інкапсульовані компонентом – вони використовуються тільки всередині, класами, що кооперуються, і не видимі іншим компонентам.

Інтерфейс, що конкретизується компонентом може бути реалізований в окремому класі. Подібний клас називається домінантним класом. Оскільки домінантний клас являю собою інтерфейс компонента, будь-який об'єкт усередині компонента можна досягти з домінантного класу через зв'язки композиції.

Вузли являють собою місце розташування виконання компонент. Компоненти функціонують на вузлах. Компоненти розгортаються на вузлах. Вузли разом з їх компонентами іноді називають елементами розміщення.

Компонентні моделі

Компонентна модель являє собою набір вимог, яким повинні задовольняти компоненти та середовище. Суть компонентних моделей в основному зводиться до обмежень, які накладаються на загальнодоступні інтерфейси програмних компонентів. Обмежуючи можливі інтерфейси компонентів, компонентна модель забезпечує певний рівень взаємодії між компонентами, додатками, що використовують ці компоненти, а також компонентами та середовищем їх виконання.

Головним елементом є сам компонент. Він володіє рядом характеристик. Найважливішою характеристикою є набір інтерфейсів, що надаються.

Крім інтерфейсів, що надаються самим компонентом, у компонентній моделі враховуються інтерфейси контейнера, що використовуються компонентом. Набір подій, генеруємих компонентом, і події, що обробляються цим компонентом, також входять до складу моделі. Крім того, компонента модель визначає стандартні способи подання атрибутів. І нарешті, модель визначає стандартні способи впливу на поведінку компонента та значення атрибутів.

Інтерфейси, визначені в рамках компонентної моделі, задають особливості взаємодії між компонентами й іншими фрагментами коду. Ці фрагменти коду можуть належати контейнеру, додатку, іншим компонентам або службам, що працюють у мережі. Одні компонентні моделі повністю вказують кожний тип інтерфейсу, інші лише визначають ті інтерфейси, за допомогою яких компоненти взаємодіють із додатком.

Інтерфейси підрозділяються на наступні категорії:

– *API*-компонента. Прикладний програмний інтерфейс компонента – це набір стандартних інтерфейсів, за допомогою яких додаток може користуватися послугами компонента;

– шаблони проекту – являють собою набори класів і інтерфейсів, а також зв'язків між ними. Додаток уточнює шаблон, реалізуючи конкретне поводження елемента;

– *SPI (Service Provider Interface)* – компонента. *SPI* надає набір стандартних інтерфейсів, реалізованих виробниками служб. Завдяки наявності *SPI* додаток може працювати з різними службами, при цьому не виникає необхідності в модифікації коду додатка або контейнера;

– інтерфейси між компонентами та контейнерами. Цей тип інтерфейсу визначає взаємодію компонента з контейнером, у якому виконується цей компонент.

Характеристики інтерфейсів:

– залежність від мови. В одних моделях для опису інтерфейсів використовуються засоби не залежні від мови, в інших моделях інтерфейси визначаються в термінах конкретної мови прог-рамування. Для того, щоб визначення не залежало від мови, інтерфейс задається спеціальною мовою опису ін-

терфейсу, а потім відображається в конкретну мову програмування;

– залежність від платформи. Деякі компонентні моделі орієнтовані на конкретну операційну систему й апаратні засоби, однак більшість моделей не залежить від платформи;

– залежність від комунікаційного протоколу. У деяких моделях компоненти описуються на більш високому рівні абстракції, завдяки цьому досягається незалежність від протоколу;

– залежність від подання даних. У деяких моделях визначається формат переданих даних через інтерфейси компонентів;

– синхронна й асинхронна взаємодія. В одних моделях передбачається синхронна взаємодія з компонентами, при якій фрагмент програми, що звертається до компонента, припиняє своє виконання доти, поки робота компонента не завершиться. В інших моделях використавується асинхронна взаємодія, коли фрагмент програми, що викликається, обмінюється з компонентом повідомленнями/подіями.

– ступінь конкретизації при визначенні інтерфейсу. Одні компонентні моделі визначають конкретні операції, дозволяючи при цьому додавати операції, обумовлені користувачем. Інші моделі задають шаблони, яких необхідно дотримуватися, реалізуючи необхідні операції;

– керування поведінкою. Одні компоненти надають користувачам доступ до засобів, що управляють їх поведінкою, та дозволяють модифікувати ці засоби (компоненти типу «білий ящик»). Інші компоненти обмежують дії користувача інтерфейсами призначеними для взаємодії з компонентами; деталі компонента недоступні користувачеві. Існують також компоненти, що допускають часткове керування своєю поведінкою через настроювання окремих елементів.

Стандартні компонентні моделі

У цей час існує велика кількість стандартних компонентних моделей. Наприклад:

– аплети. *Java*-аплети визначають прості інтерфейси взаємодії між компонентом і контейнером, а також між двома

контейнерами, що дозволяє *Java*-коду виконуватися у середовищі контейнера, вбудованого у *Web*-браузер;

– *JavaBeans*. Компонентна модель *JavaBeans* визначає стандартну модель побудови *Java*-коду, що допускає настроювання в процесі виконання та надає контейнеру *JavaBeans* атрибут, події й інтерфейси. Як правило, контейнери додаються до складу інтегрованого середовища розробки. Це дозволяє робити настроювання компонентів *JavaBeans*, використовуючи інструментальні засоби з графічним інтерфейсом користувача. Компонентна модель *JavaBeans* також визначає стандартний інтерфейс контейнера та стандартний інтерфейс взаємодії між компонентами під назвою *InfoBus;*

– *CORBA (Common Object Request Broker).* Архітектура *CORBA* визначає стандартну модель доступу до розподілених об'єктів, що реалізовані на будь-якій мові програмування. Стандартне середовище *ORB* перетворить звертання клієнта в дані, передані серверу за допомогою стандартного комунікаційного протоколу, і у виклики методів розподілених об'єктів. Крім стандартної інфраструктури взаємодії компонентів і вилучених викликів, модель *CORBA* визначає набір високорівневих розподілених служб.

– *RMI (Remote Method Invocation).* Модель *RMI* визначає засоби доступу до розподілених компонентів на базі *Java*. Середовище контейнера *RMI* є складовою частиною платформи *Java* та розширене для підтримки взаємодії за протоколом, що використовується *CORBA*.

– *COM (Component Object Model). COM* – стандартна модель взаємодії між компонентами, що реалізовані на платформі Microsoft. *DCOM(Distributed Component Object Model)* являє собою варіант моделі *COM* для роботи з розподіленими об'єктами. Моделям *COM* і *DCOM* передували технології *Microsoft Active і OLE.*

– *Java API/SPI. Java* – це не тільки мова. Різні *Java API* надають додаткам можливість звертатися до баз даних *JDBC* , службам імен і каталогам *JNDI*, службам підтримки транзакцій (*JTA* і *JTS*), службі повідомлень (*JMS*) і засобам поставки послуг (*Jini*). Більшість *API* дозволяють працювати з універ-

сальними засобами, що забезпечують інтерфейс із різними реалізаціями служб.

– *Web*-компоненти *J2EE*. *Java*-сервлети й *Java Service Page* являють собою компоненти, що працюють на стороні сервера, і використовуються для обробки клієнтських запросів і генерування відповідей. Ці *Web*-компоненти працюють у середовищі, що забезпечується Web-контейнером J2EE.

– *J2EE Enterprise JavaBeans*. *EJB*-компоненти, що відповідають стандартної моделі, що виконується на стороні сервера. Вони працюють у середовищі, що забезпечується *EJB*-контейнерами й серверами. Сервери надають *EJB* набір послуг, таких як підтримка транзакцій, забезпечення безпеки та масштабованість.

– *Microsoft Microsoft Distributed Network Archtecture (DNA)*. *DNA* являє собою архітектуру обслуговування додатків, що виконуються на платформі *Microsoft*. *DNA* поєднує багато стандартних компонентів *Microsoft*, що призначені для додатків рівня підприємства.

7.4 Практичні завдання

Завдання 1. Напишіть реферати на одну з таких тем:

– шаблони корпоративних програмних додатків;

– типи багатошарових систем;

– особливості проектування баз даних з використанням UML;

Завдання 2. Для систем розглянутих у завданнях 2 та 3 з розділу 5, та завданні 1 розділу 6 виконайте архітектурний аналіз.

Завдання 3. Створіть свій метрологічний сервер на базі Raspberri Py та Thomas More Extension Board. Створіть інформацію про розробників. Коротку теоретичну інформацію. Форму зворотнього зв’язку. Форму реєстрації.

Завдання 4. Для кожної з наведеної далі системи вкажіть архітектурні стилі що можна до них використати:

– електронний гравець у шахи;

– симулятор водного мотоциклу для гри;

– сонар – система знаходження підводних об’єктів та розрахунку відстані до них.

Завдання 5. Використовуючи вільне прогармне забезбеячення для Raspberry Py розробіть проект систeмти на реалізуюте

Оброблення відео-зображення з Raspberry Pi та обобку даних датчика оствещенності з графіком залежностей

Оброблення відео-зображення з Raspberry Pi та обобку даних датчика запиленості з графіком залежностей

Оброблення відео-зображення з Raspberry Pi та обобку даних датчика температури з графіком залежностей

Завдання 6. Для Basys2 board розробіть гру Mario. Обгрунтуйте вибір програмного та апаратного забезпечння.

Завдання 7. Для SMT32 Discovery спроектуйте архітектуру квадрокоптера.

7.5 Контрольні запитання

1. З яких етапів складається створення архітектури системи?
2. У чому різниця між бібліотеками та каркасами?
3. Якими якостями повинні володіти «хороші» бібліотеки класів?
4. Що розуміють під терміном шаблон?
5. У чому різниця між шаблонами та каркасами?
6. Дайте визначення підсистеми.
7. Як можуть бути організовані відносини між підсистемами?
8. Як організована багатошарова система?
9. У чому різниця розділів та шарів при організації системи?
10. Який засіб є основним при виділені паралелізмів?
11. 11.Які кроки необхідно виконати при розподілі підсистем?
12. 12.Які типи даних краще зберігати у файлах, а які в базах даних? Наведіть приклад.
13. Які глобальні ресурси повинен виділити проектувальник?

14. Які варіанти управління програмною системою існують?
15. Яким граничним умовам проектувальник повинний приділити увагу?
16. Назвіть існуючі архітектурні стилі. Дайте їм визначення та коротку характеристику.
17. Дайте визначення компонента. В чом його відмінність від класу та пакету?
18. Які властивості мають компоненти?
19. Які основні категорії інтерфейсів?
20. Назвіть характеристики інтерфейсів.
21. Які стандартні компонентні моделі вам відомі? Дайте їх визначення та коротку характеристику.

7.6 Література до розділу

1. P. Arras, Architectural Characteristics and Educational Possibilities of the Remote Laboratory in Materials Properties/ P. Arras, G. Tabunshchyk, Ye. Kolot, B. Tanghe// REV2014 Conference26 – 28 February 2014, Porto, Portugal, PP. 94-97
2. Remote and virtual tools in engineering: student textbook / general editorship Dr.Ing.Karsten Henke. – Zaporizhzhya: Dike Pole, 2016. – pp. 250.
3. Tabunshchyk G. Multipurpose Educational System based on Raspberry Pi/ G. Tabunshchyk, D. Van Merode , O.Petrova, V. Okhmak // Proceedings of the International Symposium on Embedded Systems and Trends in Teaching Engineering, Nitra, Slovakia, 12-15 September, 2016 – pp. 202-206
4. Ohmak V.A. Conveyor Model for Embedded Systems Study /V.A. Ohmak, G.V Tabunshik// Proceedings of the International Symposium on Embedded Systems and Trends

in Teaching Engineering, Nitra, Slovakia, 12-15 September, 2016 –pp.60-64.

5. Merode D. Van Merode Interactive university platform // Merode D. Van, Tabunshchyk G., Goncharov Y., Patrakhalko K., Staroverov V.// Сучасні проблеми і досягнення в галузі радіотехніки, телекомунікацій та інформаційних технологій : тези доповідей VIII Міжнародної науково-практичної конференції (21–23 вересня 2016 р., м. Запоріжжя). – Запоріжжя : ЗНТУ, 2016. – 344 с.
6. M. Fowler Refactoring: Improving the Design of Existing Code / M. Fowler,K. Beck,J. Brant,W. Opdyke,D. Roberts. – USA: ADDISON-WESLEY, 2012. – 455 p.
7. Эванс Э. Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. [Текст] / Эванс Э. – М.:Вильямс, 2010. – 448 с.

АЛФАВІТНО-ПРЕДМЕТНИЙ ПОКАЖЧИК

A
Agile 23

B
Balanced scorecard 68

J
JAD-метод 57

K
Kanban 30

R
Raspberri Pi 195

S
Scrum 24
Scrum документи 26
- журнал продукту 26
- журнал спринту 26
-графік спринту26
Scrum-практики 27
-спринт 28
- скрам 28
- демонстраційне засідання 29
STM32F4DISCOVERY плата 209

U
UML 14, 188

А
Абстрагування 15
Адміністратор транзакцій 174
Аналітична модель 9,82
Анкетування 55
Архітектура системи 151
Атрибути якості 61

Б
Багатошарова система 156
Бібліотека 152
Бізнес-архітектура 68

В
Відмова 169
Види структурних ієрархій 17
–агрегація 17
–ієрархії «is-a» і «is-like-a» 17-18
Вивчення документів і програмних систем 56

Е
Етапи інтеграції ПЗ 11
Етапи супроводження 11
– підтримання експлуатації 11
– адаптивне супроводження 11
– супроводження для поліпшення 12

Ж
Життєвий цикл ПЗ 8
Журнал продукту 26

І
Ініціативи консорціуму OMG 21
Інтерфейсом користувача 120
Ітеративна розробка 23
Інтерв'ю 55

К

Каркас 153
Керування 165
Клас 15
Контролер 145
– зовнішній 145
– прецеденту 145
Концептуалізація системи 9, 53
Компоненти об'єктної моделі 14, 176
– абстрагування 15
– інкапсуляція 16
– модульність 19
– ієрархія 19

М

Метод швидкого розроблення додатків 57
Моделювання 70
– прецедентів 106
Модель
– аналітична 9, 82
– об'єктна 14
– предметної області 10, 82
– станів додатку 122

О

Об'єкт 15
Об'єктно-орієнтована методологія та мова UML 14, 187
Об'єктна модель 14
Основні види абстракцій ООП 15
– клас 15
– об'єкт 15
Об'єктно-орієнтовані методи UML 20
– Граді Буча 20
– Джеймса Румбаха 20
– Айвара Джекобсона 20
Об'єктний аналіз 72
Операційний ризик 76

П

Перетворення 171
Підхід ВСЕ 121
Підсистема 154
Проектування архітектури системи 10
Проектування класів 10
Проектування системи 151
Підходи до розроблення ПЗ 12
– Структурний підхід 12
– Об'єктно-орієнтований підхід 13
Повторне використання 38
Потік керування 158
Простір імен 19
Призначення мови UML 22
Повторне використання коду 37
Прототипування 56
Процес 68
Пакет 97

Р

Реалізація ПЗ 10
Розробки інформаційних систем 23
– Ітеративна розробка 23
Ролі Scrum-проектів 25
– Власник продукту 25
– Scrum-майстер 25
– Scrum-команда 25-26

С

Системна подія 143
Системні операції 143
Специфікація вимог 9
Споріднені типові рішення 147

Ш

Шаблон 128, 153
– Creator 133
– Controller 142
– Information Expert128
– Low Coupling 135
– High cohesion 135

ДОДАТОК А. УНІФІКОВАНА МОВА МОДЕЛЮВАННЯ

UML – це не візуальна мова програмування, але її моделі прямо транслюються в текст на мовах програмування та утаблиці реляційної БД.

Словник *UML* утворюють три різновиди будівельних блоків: предмети, відносини та діаграми.

Предмети – це абстракції, які є основними елементами в моделі, відносини пов'язують ці предмети, діаграми групують колекції предметів.

А.1 Предмети

У *UML* є чотири різновиди предметів: структурні предмети; предмети поведінки; предмети групування; предмети-коментарі.

Ці предмети є базовими об'єктно-орієнтованими будівельними блоками. Вони використовуються для написання моделей.

Структурні предмети є іменниками в UML моделях. Вони являють собою статичні частини моделі, понятійні або фізичні елементи.

1. ***Клас*** – це опис багатьох об'єктів, що поділяють однакові Властивості, операції, відносини та семантику. Клас реалізує один або декілька інтерфейсів (рис. А.1).

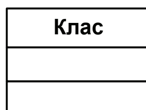

Рисунок А.1 – Клас

Активний клас – це клас чиї об'єкти мають один або декілька процесів (або потоків) і тому можуть ініціювати керу-

ючу діяльність. Активний клас схожий на звичайний клас за винятком того, що його об'єкти діють одночасно з об'єктами інших класів.

2. ***Інтерфейс*** – набір операцій, що визначають послуги класу або компонента (рис. А.2). Інтерфейс описує поведінку елемента, видиму ззовні. Інтерфейс може представляти повні послуги класу або компонента. Інтерфейс визначає набір специфікацій операцій, а не набір реалізацій операцій. Інтерфейс рідко показують самостійно. Зазвичай його приєднують до класу або компоненту, який реалізує інтерфейс.

Навчання ○——

Рисунок А.2 – Інтерфейс

3. ***Кооперація*** визначає взаємодію і є сукупністю ролей та інших елементів, які працюють разом для забезпечення колективної поведінки більш складного, ніж проста сума всіх елементів. (рис. 2.8) Таким чином, кооперації мають як структурне, так і поведінкове вимірювання. Конкретний клас може брати участь в декількох коопераціях. Ці кооперації являють собою реалізацію патернів, які формують систему.

Рисунок А.3 – Кооперація

4. ***Актор*** – набір узгоджених ролей, які можуть грати користувачі при взаємодії з системою. Кожна роль вимагає від системи певної поведінки.

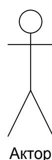

Рисунок А.4 – Актор

5. Елемент ***Use-Case*** (*Варіант використання* або *Прецедент*) – це опис послідовності дій (чи декількох послідовностей), що виконуються системою в інтересах окремого актора та формують видимий для актора результат (рис. А.5). У моделі прецедент застосовується для структурування предметів поведінки. Елемент Прецедент реалізується кооперацією.

Рисунок А.5 – Прецедент

6. ***Компонент*** – фізична і займана частина системи, що відповідає набору інтерфейсів і забезпечує реалізацію цього набору інтерфейсів (рис. А.6). У систему включаються як компоненти, які є результатами процесу розробки (файли вихідного коду), так і різні різновиди використовуваних компонентів. Зазвичай компонент – це фізична упаковка різних логічних елементів (класів, інтерфейсів і співробітництв).

Рисунок А.6 – Компонент

7. ***Вузол*** – фізичний елемент, що існує в період роботи системи і являє собою деякий ресурс, який зазвичай має пам'ять і можливості обробки (рис. А.7). У вузлі розміщується набір компонентів, що може переміщатися від одного вузла до іншого.

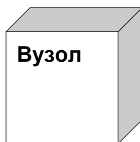

Рисунок А.7 – Вузол

Предмети поведінки – це динамічні частини UML-моделей. Вони є дієсловами моделей, уявою поведінки в часі і

просторі. Існують дві основні різновиди предметів поведінки.

1. Взаємодія – поведінка, що містить у собі набір повідомлень, якими обмінюється набір об'єктів в конкретному контексті для досягнення певної мети (рис. А.8).

Взаємодія може визначати динаміку як сукупності об'єктів, так іокремої операції. Елементами взаємодії є повідомлення, які єпослідовністю дій та зв'язку.

Рисунок А.8 – Взаємодія

2. Кінцевий автомат – поведінка, яка визначає послідовність станів об'єкта або взаємодії, що виконуються в ході його існування увідповідь на події (рис. А.9). З допомогою кінцевого автомата може визначатися поведінка індивідуального класу або кооперації класів. Елементами кінцевого автомата є стани, переходи (від стану до стану), події (предмети, що викликають переходи) та дії (реакції на перехід).

Рисунок А.9 – Стан

Ці два елементи взаємодії і кінцеві автомати – є базисними предметами поведінки, які можуть включатися в *UML* моделі. Семантично ці елементи асоціюються з різними структурними елементами.

Предмети групування – організаційні частини *UML* моделей. Це шухляди, по яких може бути розкладена модель.

Пакет – загальний механізм для розподілу елементів за групами. У пакет можуть поміщатися структурні предмети, предмети поведінки і навіть інші групування предметів. На відміну від компонента, пакет– чисто концептуальне поняття. Це означає, що пакет існує тільки вперіод розроблення (рис. А.10).

Рисунок А.10 – Пакет

Предмети-коментарі – роз'яснюють частини *UML*-моделей. ***Примітка*** – символ для відображення обмежень і зауважень, приєднуваних до елементу або сукупності елементів.

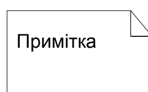

Рисунок А.11 – Примітка

А.1.1 Відношення в ***UML***

UML використовує чотири різновиди відношень:

– залежність;
– асоціацію;
– узагальнення;
– реалізацію.

Ці відносини є базовими будівельними блоками відносин. Вони використовуються при написанні моделей.

1 ***Залежність*** – семантичне відношення між двома предметами, в якому зміна в одному предметі може впливати на семантику іншого предмета (рис. А.12).

Рисунок А.12 – Відношення залежності

2. ***Асоціація*** – структурне відношення, що описує набір зв'язків, які є з'єднанням між об'єктами (рис. А.13).

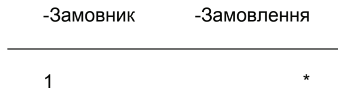

Рисунок А.13 – Асоціація

3. ***Узагальнення*** – відношення спеціалізації/узагальнення, вякому об'єкти спеціалізованого елемента (нащадка) можуть замінювати узагальнененний елемент (предка) (рис. А.14).

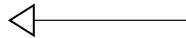

Рисунок А.14 – Відношення узагальнення

4. ***Реалізація*** – семантичне відношення між класифікаторами, де класифікатор визначає контракт, який другий класифікатор зобов'язується виконувати (рис. А.15). Відносини реалізації застосовують у двох випадках: між інтерфейсами і класами (або компонентами), що реалізовують їх; між прецедентами і кооперації, які реалізують їх.

Рисунок А.15 – Відношення реалізації

А.1.2 Механізм розширення в ***UML***

UML – розвинута мова з великими можливостями, але навіть вона не може відбити всі нюанси, що можуть виникнути при створенні різних моделей. Тому *UML* містить засоби розширення:

– обмеження;
– тегові величини;
– стереотипи.

Обмеження (*constraint*) – розширює семантику будівельного *UML*-блоку, дозволяючи додати нові правила або модифіковані існуючі. Обмеження показують як текстовий рядок, укладений у фігурні дужки {} (рис. А.16).

Сесія банкомату
Сума: Гроші
{величина кратна
$20}

Рисунок А.16 – Використання обмежень на атрибути

Тегова величина (*tagged value*) – розширює характеристики будівельного *UML*-блоку, дозволяючи створювати нову інформацію вспецифікації конкретного елемента. Тегові величини показують як рядок у фігурних дужках {}. Рядок має вигляд

ім'я тегової величини = значення;

Стереотип дозволяє розширювати словник мови, дозволяє створювати нові види будівельних блоків, похідні від існуючих та враховують специфіку нової проблеми. Елемент зі стереотипом єваріацією існуючого елемента, що має таку ж форму, але відрізняється по суті. Відображають стереотип як ім'я в подвійних кутових дужках.

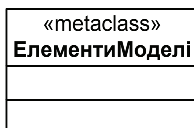

Рисунок А.17 – Приклад використання стереотипів

ДОДАТОК Б
ОПИС ПЛАТИ RASPBERRI PI

Raspberry Pi є компактним одноплатним комп'ютером. Усі компоненти розміщено на чотиришаровій друкованій платі розміром з кредитну карту, тим не менш, це не просто обчислювальна платформа, а повноцінний комп'ютер, аналогічний до ПК.

Зовнішній вигляд та складові пристрою наведені на рис Б.1.

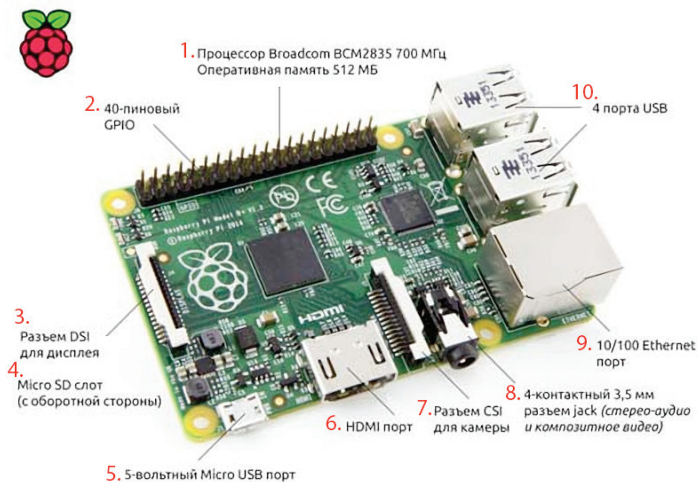

Рисунок Б.1 – Зовнішній вигляд Raspberry Pi 2

Б.1 Апаратна частина

Апаратна частина пристрою складається з наступних компонентів:

1. Основою є система-на-чипі (SoC) Broadcom BCM2835 — процесор на архітектурі ARM з тактовою частотою 1 ГГц та 1 Гб RAM. Процесор відповідає за роботу комп'ютера, а також на нього виведені контакти GPIO;

2. 40-піновий GPIO — набор контактів для взаємодії з зовнішніми пристроями;
3. Роз'єм DSI для підключення спеціально розроблени дісплеїв;
4. Micro SD слот для карти пам'яті (Raspberry Pi не має вбудованого запам'ятовуючого пристрою);
5. П'ятивольтний Micro USB порт для живлення плати;
6. HDMI-відеовихід для підключення монітору;
7. Роз'єм CSI для підлючення спецілально розроблених камер;
8. Чотириконтактний 3,5 мм роз'єм (здатний відтворювати стерео-аудіо та аналогове композитне відео);
9. Порт Ethernet для підключення до мережі за допомогою кабеля;
10. 4 USB-порти.

Б.2 Перший запуск. Робота та підключення Raspberry Pi

Для використання Raspberry Pi нам знадобляться такі компоненти: власне, плата; карта пам'яті microSD (щонайменше 4 Гб); монітор з можливістю підключення HDMI; блок живлення microUSB на 5V (підходить зарядний пристрій більшості сучасних смартфонів) та Ethernet-кабель для підключення Raspberry Pi до мережі для установки додаткових програм (необов'язково, якщо все необхідне було заздалегідь завантажено).

1. По-перше, треба записати образ операційної системи на карту пам'яті, яку ми будемо використовувати. Для цього нам потрібен комп'ютер зі зчитувачем карт пам'яті SD та перехідник microSD-SD. Для запису образу необхідно:
2. Карта пам'яті повинна бути чистою та відформатованою як fat32.

3. Потрібно завантажити архів NOOBS (Offline and network install) з офіційного сайту [1].
4. Архів розпакувати та скопіювати увесь вміст на карту пам’яті.
5. Вставивши карту пам’яті у Raspberry, під’єднати його до монітора, потім вмикнути кабель живлення (microUSB). На перший запуск NOOBS запропонує декілька варіантів установки и обираємо Raspbian. Чекаємо, доки закінчиться процес установки.
6. Після установки відбудеться перезавантаження та з’явиться вікно програми raspi-config (програма налаштування Raspberry Pi). В цьому вікні обираємо опцію “Enable boot to Desktop” та підтверджуємо вибір, тоді за замовчуванням буде завантажуватися графічне середовище LXDE. Далі натискаємо “Done” та пристрій перезаватажується ще раз.
7. Стандартний користувач та пароль для ОС Raspbian, відповідно — *pi* та *raspberry*. Якщо було встановлено “Boot to Desktop”, авторизуватися в системі буде не потрібно.
8. Після цих дій система буде готова до роботи. Термінал та інші програми можно знайти в меню у лівому верхньому куті екрану.

За наявності підключення до Інтернет, рекомендується обновити систему за допомогою команд

```
sudo apt-get update && sudo apt-get upgrade -y
```

Б.3 Програмне забезпечення

Оскільки Raspberry Pi є повноцінним комп’ютером, він може працювати під керуванням будь-якої операційної системи, що портована на архітектуру ARM. На даний момент це переважно UNIX-подібні ОС на основі ядра Linux, найпопулярнішою з яких є Raspbian (перероблена для використання на Raspberry Pi ОС Debian *GNU*/Linux).

Первісно, Raspberry Pi проектувався як бюджетна платформа для навчання інформатиці, але зараз він набрав популярність і як навчальна платформа, і серед ентузіастів, і як компонент вбудованих систем. Такої популярності Raspberry Pi досяг завдяки можливостям дуже різноманітного застосування. Основною деталлю, яка відрізняє Raspberry Pi від звичайного компактного ПК є інтерфейс GPIO.

Б.4 Опис інтерфейсу GPIO

GPIO (General purpose input-output) — низькорівневий інтерфейс вводу-виводу з прямим керуванням. Він являє собою 40 пінів (контактів), виведених і послідовно розташованих у два ряди на одній зі сторін плати. З використанням цього інтерфейсу можливо віддавати команди або приймати сигнали з будь-якого пристрою. GPIO є цифровим інтерфейсом і оперує двома станами: логічний нуль (напруга 0В) та логічна одиниця (напруга 3,3В).

Pin#	NAME	NAME	Pin#
01	3.3v DC Power	DC Power 5v	02
03	GPIO02 (SDA1 , I2C)	DC Power 5v	04
05	GPIO03 (SCL1 , I2C)	Ground	06
07	GPIO04 (GPIO_GCLK)	(TXD0) GPIO14	08
09	Ground	(RXD0) GPIO15	10
11	GPIO17 (GPIO_GEN0)	(GPIO_GEN1) GPIO18	12
13	GPIO27 (GPIO_GEN2)	Ground	14
15	GPIO22 (GPIO_GEN3)	(GPIO_GEN4) GPIO23	16
17	3.3v DC Power	(GPIO_GEN5) GPIO24	18
19	GPIO10 (SPI_MOSI)	Ground	20
21	GPIO09 (SPI_MISO)	(GPIO_GEN6) GPIO25	22
23	GPIO11 (SPI_CLK)	(SPI_CE0_N) GPIO08	24
25	Ground	(SPI_CE1_N) GPIO07	26
27	ID_SD (I2C ID EEPROM)	(I2C ID EEPROM) ID_SC	28
29	GPIO05	Ground	30
31	GPIO06	GPIO12	32
33	GPIO13	Ground	34
35	GPIO19	GPIO16	36
37	GPIO26	GPIO20	38
39	Ground	GPIO21	40

Рисунок Б.2 — Схема розташування контактів GPIO на Raspberry Pi 2

Контакти GPIO виведені напряму з процесору і керовані їм же. Вони не є буферизованими або захищенними від перена

Первісно, Raspberry Pi проектувався як бюджетна платформа для навчання інформатиці, але зараз він набрав популярність і як навчальна платформа, і серед ентузіастів, і як компонент вбудованих систем. Такої популярності Raspberry Pi досяг завдяки можливостям дуже різноманітного застосування. Основною деталлю, яка відрізняє Raspberry Pi від звичайного компактного ПК є інтерфейс GPIO.

Б.4 Опис інтерфейсу GPIO

GPIO (General purpose input-output) — низькорівневий інтерфейс вводу-виводу з прямим керуванням. Він являє собою 40 пінів (контактів), виведених і послідовно розташованих у два ряди на одній зі сторін плати. З використанням цього інтерфейсу можливо віддавати команди або приймати сигнали з будь-якого пристрою. GPIO є цифровим інтерфейсом і оперує двома станами: логічний нуль (напруга 0В) та логічна одиниця (напруга 3,3В).

Pin#	NAME	NAME	Pin#
01	3.3v DC Power	DC Power 5v	02
03	GPIO02 (SDA1 , I2C)	DC Power 5v	04
05	GPIO03 (SCL1 , I2C)	Ground	06
07	GPIO04 (GPIO_GCLK)	(TXD0) GPIO14	08
09	Ground	(RXD0) GPIO15	10
11	GPIO17 (GPIO_GEN0)	(GPIO_GEN1) GPIO18	12
13	GPIO27 (GPIO_GEN2)	Ground	14
15	GPIO22 (GPIO_GEN3)	(GPIO_GEN4) GPIO23	16
17	3.3v DC Power	(GPIO_GEN5) GPIO24	18
19	GPIO10 (SPI_MOSI)	Ground	20
21	GPIO09 (SPI_MISO)	(GPIO_GEN6) GPIO25	22
23	GPIO11 (SPI_CLK)	(SPI_CE0_N) GPIO08	24
25	Ground	(SPI_CE1_N) GPIO07	26
27	ID_SD (I2C ID EEPROM)	(I2C ID EEPROM) ID_SC	28
29	GPIO05	Ground	30
31	GPIO06	GPIO12	32
33	GPIO13	Ground	34
35	GPIO19	GPIO16	36
37	GPIO26	GPIO20	38
39	Ground	GPIO21	40

Рисунок Б.2 — Схема розташування контактів GPIO на Raspberry Pi 2

Контакти GPIO виведені напряму з процесору і керовані їм же. Вони не є буферизованими або захищенними від перена-

вантаження, тож треба бути обережним при роботі з GPIO, не підключати прилади, що з робочим током більше 50 мА, і не подавати напругу більшу ніж 3,3В.

GPIO включає декілька типів контактів — 5V DC, 3,3V DC, GND (земля, 0V), та власне керовані контакти GPIO (пронумеровані від 2 до 27). Схема розташування контактів наведена на рис Б.3.

Б.5 Способи роботи з GPIO

Існує багато способів керувати пінами GPIO на Raspberry Pi. Існують бібліотеки для роботи з GPIO для більшості популярних мов програмування: C/C++, C#, Java, Python, Perl, Ruby, Free Pascal, BASIC, Shell (Bash). Ми розглянемо два з них: Shell (Bash), C/C++.

Б.5.1 Bash та командна строка

У системах на базы ядра Linux усе, зокрема і пристрої, представлено як файл. GPIO не є виключенням. Для роботи використовуються команди cat (для зчитування) та echo (для запису). Також, потрібний адміністративний доступ (root), бо GPIO є системним ресурсом.

Приклад запису:

```
sudo echo «4» > /sys/class/gpio/export # оголошуємо
використання 4-го піна
sudo echo «out» > /sys/class/gpio/gpio4/direction #
оголошуємо викорисаний пін як вихід
sudo echo «1» > /sys/class/gpio/gpio4/value # виво-
димо логічну одиницю
sudo echo «0» > /sys/class/gpio/gpio4/value # виво-
димо логічний нуль
```

Приклад зчитування:

```
sudo echo «0» > /sys/class/gpio/export # оголошуємо
використання 0-го піна
sudo echo «in» > /sys/class/gpio/gpio0/direction #
оголошуємо використаний пін як вхід
sudo cat /sys/class/gpio/gpio0/value # зчитуємо зна-
чення
```

Б.5.2 C/C++

Для легкої роботи з GPIO на C/C++ існують декілька бібліотек, найпопулярніші серед них: WiringPi (докладніше за посиланням [2]) та bcm2835 (завантажити можна з ресурсу [4]).

WiringPi

WiringPi є подібним до мови Wiring, що використовується на Arduino. WiringPi використовує власну нумерацію пінів, відповідність нумерації WiringPi до номерів GPIO наведена на рис Б.3. Простий приклад використання бібліотеки WiringPi для керування світлодіодом:

```
#include <wiringPi.h>
int main (void) {
 int pin = 7; # використовуємо пін GPIO4
 wiringPiSetup();
 pinMode (pin1, OUTPUT); # оголошуємо пін як вихід
 while(1) {
 digitalWrite (pin1, HIGH); # подаємо на вихід ло-
гічну одиницю
 delay(500); # чекаємо 500 мілісекунд
 digitalWrite (pin1, LOW); # подаємо на вихід логіч-
ний нуль
 delay(500);
 }
 return 0;
}
```

Raspberry Pi GPIO Header

BCM	WiringPi	Name	Physical	Physical	Name	WiringPi	BCM
		3.3v	1	2	5v		
2	8	SDA.1	3	4	5V		
3	9	SCL.1	5	6	0v		
4	7	1-Wire	7	8	TxD	15	14
		0v	9	10	RxD	16	15
17	0	GPIO. 0	11	12	GPIO. 1	1	18
27	2	GPIO. 2	13	14	0v		
22	3	GPIO. 3	15	16	GPIO. 4	4	23
		3.3v	17	18	GPIO. 5	5	24
10	12	MOSI	19	20	0v		
9	13	MISO	21	22	GPIO. 6	6	25
11	14	SCLK	23	24	CE0	10	8
		0v	25	26	CE1	11	7
0	30	SDA.0	27	28	SCL.0	31	1
5	21	GPIO.21	29	30	0v		
6	22	GPIO.22	31	32	GPIO.26	26	12
13	23	GPIO.23	33	34	0v		
19	24	GPIO.24	35	36	GPIO.27	27	16
26	25	GPIO.25	37	38	GPIO.28	28	20
		0v	39	40	GPIO.29	29	21
BCM	WiringPi	Name	Physical		Name	WiringPi	BCM

Рисунок Б.3 — Схема відповідності пінів GPIO до нумерації WiringPi

Збережімо файл як blink.c. Скомпілювати та запустити приклад в ОС Raspbian можна послідовним виконанням команд:

```
gcc -o blink blink.c -lwiringPi
sudo ./blink
```

bcm2835

Бібліотека bcm2835 використовує власний синтаксис.

Приклад з репозиторія [5]:

```
#include <bcm2835.h>
#define PIN RPI_V2_GPIO_P1_04 # використовуємо пін
GPIO4

int main() {
 if (!bcm2835_init()) // Ініціалізація GPIO
 return 1; // Завершуємо програму за умови помилки
 bcm2835_gpio_fsel(PIN, BCM2835_GPIO_FSEL_OUTP); //
Встановлюємо порт P1_04 на вихід
 bcm2835_gpio_write(PIN, LOW); // Встановлюємо порт
в 0, діод горить
 bcm2835_delay(1000); // Чекаємо 1 секунду
 bcm2835_gpio_write(PIN, HIGH); // Встановлюємо порт
в 0, діод не горить
 return 0;
}
```

Збережімо файл як bcm_test.c. Компіляція та запуск выдбуваються командами:

```
gcc -o bcm_test bcm_test.c -lbcm2835
sudo ./bcm_test
```

Б.6 Підтримувані інтерфейси передачі даних

GPIO реалізує декілька інтерфейсів передачі даних:

1. пряма передача сигналів (наприклад, за допомогою бібліотеки WiringPi);
2. UART (піни 8, 10);

3. I2C (піни 3, 5);
4. PCM (піни 12, 35, 38, 40);
5. SPI (піни 19, 21, 23, 24, 26).

Б.7 Плата розширення Export More

Дана плата є спеціально спроектованою для навчання платою розширення для Raspberry Pi. Вона містить на собі датчики освітленості, температури та 8 світлодіодів. Схема плати наведена на рис Б.5.

Використані приклади коду з репозиторія [5] розробників плати Export More.

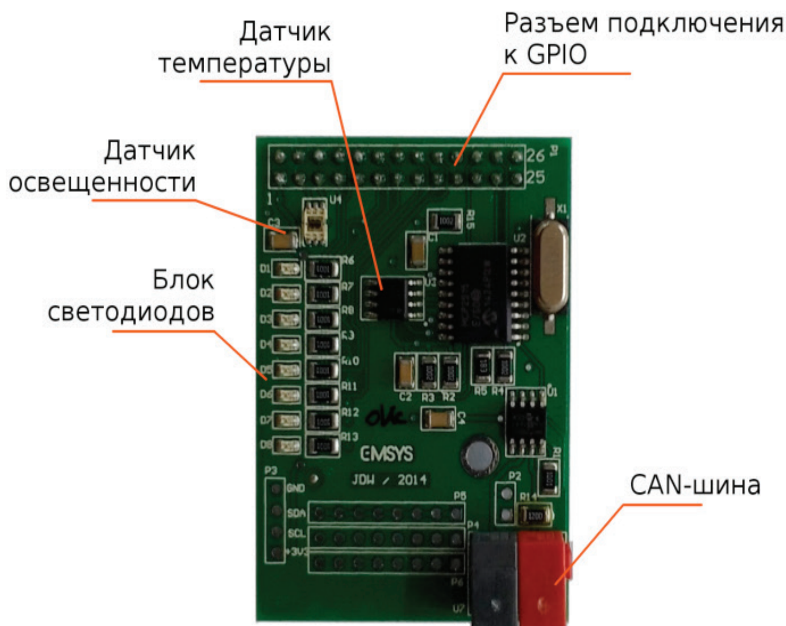

Рисунок Б.4 — Плата розширення Export-More

Для роботи з датчиками використовується бібліотека bcm2835. Самі датчики для передачі даних використовують

інтерфейси I2C та SPI: датчик освітленості використовує I2C, а датчик температури, відповідно, SPI.

Основні функції бібліотеки bcm2835, які ми будемо використовувати у програмах, наведено у таблиці Б.1.

Таблиця Б.1 — Основні функції бібліготеки bcm2835

Ім`я функції	Аргументи	Опис
bcm2835_init	-	Ініціалізація бібліотеки, відкриття пристрою /dev/mem та регістрів процесора. Необхідно викликати перед використанням будь-яких інших функцій бібліотеки
bcm2835_close	-	Закінчення роботи з бібліотекою, звільнення пам'яті та пристрою /dev/mem. Необхідно викликати в кінці програми
bcm2835_delay	unsigned int *millis*	Забезпечує затримку виконання коду на вказану кількість мілісекунд

Б.7.1 Установка бібліотеки bcm2835

Для роботи з бібліотекою нам знадобиться встановити її у систему Raspbian. Для цього потрібно:

1. Скачати архів з бібліотекою з ресурса [4] та розпакувати його на Raspberry Pi (подальші кроки виконуються саме на ньому).

2. Далі потрібно відкрити термінал та за допомогою команди cd перейти до каталогу, у який було розпаковано бібліотеку. Припустимо, що це домашній каталог користувача (/home/pi) та версія бібліотки — 1.50. Тоді команда виглядає так:

```
cd /home/pi/bcm2835-1.50.tar.gz
```

3. У цьому каталозі розташовані вихідній файли бібліотеки та конфігураційні файли. Для установки у систему необхідно виконати наступну послідовність команд:

```
./configure
make
sudo make install
```

Після цього бібліотека буде доступна до використання.

Б.7.2 Датчик освітленості. Інтерфейс I2C

Як було вказано вище, датчик освітленості плати Export-More використовує для передачі даних інтерфейс I2C.

I2C (Inter-Integrated Circuit) — послідовна шина передачі даних. Інтерфейс був винайдений у 1980-х роках компанією Philips для зв'язку внутрішніх блоків пристроїв. Сьогодні I2C реалізовано у більшості мікроконтролерів, а виробники різноманітних модулів використовують для обміну даними саме його.

Обмін даними в I2C організований за принципом «ведучий-ведений» («master-slave»), при чому стандарт дозволяє присутність декількох ведучих пристроїі на одній лінії.

Обмеженням I2C є швидкість передачі — 100 Кб/с, а також максимальна кількість підключених пристроїв — 112.

Для роботи інтерфейс I2C використовує два провода — послідовна лінія передачі даних (SDA) та послідовна лінія тактування (SCL).

На Raspberry Pi 2 I2C реалізований на пінах 3 та 5 (SDA та SCL, відповідно). Для роботи з інтерфейсом ми будемо використовувати вбудовані функції бібліотеки bcm2835.

Приклад коду для зчитування даних з датчику:

```
#include <bcm2835.h>
#include <stdio.h>
int main(int argc, char **argv) {
 if (!bcm2835_init())
 return 1;
 char temp[1];
```

```
 int ret;
 int ad[2];
 bcm2835_i2c_begin();
 bcm2835_i2c_setSlaveAddress(0x29);
 bcm2835_i2c_set_baudrate(1000);
 temp[0] = 0xa0;
 bcm2835_i2c_write(temp,1);
 temp[0] = 0x03;
 bcm2835_i2c_write(temp,1);
 bcm2835_delay(500);
 bcm2835_i2c_read(temp,1);
 printf("%x - if 33 the device is turned on\
n",temp[0]);
 temp[0] = 0xac;
 bcm2835_i2c_write(temp,1);
 bcm2835_i2c_read(temp,1);
 ad[1]= (int)temp[0];
 temp[0] = 0xad;
 bcm2835_i2c_write(temp,1);
 bcm2835_i2c_read(temp,1);
 ad[0] = (int)temp[0];
 printf("ad value:%d\n",ad[0]*256+ad[1]);
 bcm2835_i2c_end();
 bcm2835_close();
 return 0;
}
```

У таблиці Б.2 наведено перелік основних функцій, що використовуються для роботи з I2C, їх короткий опис та аргументи.

Таблиця Б.2 — Функції бібліотеки bcm2835 для роботи з I2C

Ім`я функції	Аргументи	Опис
bcm2835_i2c_begin	-	Початок роботи з I2C. Переводить потрібні піни у відповідний режим
bcm2835_i2c_ setSlaveAddress	uint8_t *addr*	Встановлює адрес веденого пристрою, для плати Export More це 0x29

Продовження Таблиці Б.2 —
Функції бібліотеки bcm2835 для роботи з I2C

Ім`я функції	**Аргументи**	**Опис**
bcm2835_i2c_set_ baudrate	uint32_t *baudrate*	Встановлює частоту роботи вбудованого таймера в бодах. Ми використовуємо значення за замовчуванням — 1000
bcm2835_i2c_write	const char* *buf*, uint32_t *len*	Відправляє дані (байти з буфера *buf* довжиною *len*) на встановлений ведений пристрій
bcm2835_i2c_read	char * *buf*, uint32_t *len*	Зчитує *len* байтів з веденого пристрою до буфера *buf*
bcm2835_i2c_end	-	Завершує усі операції з I2C та повертає піни у початковий стан. Необхідна в кінці програми

Б.7.3 Датчик температури. Інтерфейс SPI

Датчик температури плати Export-More використовує в роботі інтерфейс SPI.

SPI (Serial Peripheral Interface) – послідовний синхронний інтерфейс передачі даних. Був винайдений компанією Motorola. Разом з I2C, SPI є одним з найпоширеніших інтерфейсів з'єднання мікросхем.

Передача даних у SPI також організована за принципом «ведучий-ведений», де найчастіше ведучим пристроєм є мікроконтролер, а веденими — різноманітні мікросхеми, як от запам'ятовуючі пристрої або датчики.

Для роботи SPI використовує чотири провада, передає дані пакетами (за стандартом це 8 біт) та є високошвидкісним.

У Raspberry Pi 2 піни, що реалізують SPI:

— Pin 19 — SPI_MOSI;

— Pin 21 — SPI_MISO;
— Pin 23 — SPI_CLK;
— Pin 24 — SPI_CE0_N;
— Pin 26 — SPI_CE1_N.

Функції для роботи з SPI також присутні в бібліотеці bcm2835.

Приклад використання SPI для отримання даних з датчику температури:

```
#include <bcm2835.h>
#include <stdio.h>
int main(int argc, char **argv)
{
 if (!bcm2835_init())
 return 1;
 char buffer[4];
 buffer[0] = 50;
 int i,temp;
 bcm2835_spi_begin();
 bcm2835_spi_setBitOrder(BCM2835_SPI_BIT_ORDER_
MSBFIRST);
 bcm2835_spi_setDataMode(BCM2835_SPI_MODE3);
 bcm2835_spi_setClockDivider(BCM2835_SPI_CLOCK_
DIVIDER_65536);
 bcm2835_spi_chipSelect(BCM2835_SPI_CS1);
 bcm2835_spi_setChipSelectPolarity(BCM2835_SPI_CS1,
LOW);
 buffer[0]=buffer[1]=buffer[2]=buffer[3]=0;
 buffer[0] = 0x58; //read the id
 bcm2835_spi_transfern(buffer,2);
 printf("id:%02X\n",buffer[1]);
 buffer[0] = 0x50;
 bcm2835_spi_transfern(buffer,3);
 printf("status %02X %02X\n",buffer[1],buffer[2]);
 temp = buffer[1];
 temp = temp<<8;
 temp = temp + ( buffer[2] & 0xF8);
 printf("status %08x\n",temp);
 temp = temp>>3;
 temp = temp/16;
 printf("temp:%d\n",temp);
 sleep(1);
```

```
 bcm2835_spi_end();
 bcm2835_close();
 return 0;
}
```

У таблиці Б.3 наведено перелік основних функцій, що використовуються для роботи з SPI, їх короткий опис та аргументи.

Таблиця Б.3 — Функції бібліотеки bcm2835 для роботи з SPI

Ім`я функції	Аргументи	Опис
bcm2835_spi_begin	-	Початок роботи з SPI. Переводить потрібні піни у відповідний режим
bcm2835_spi_setBitOrder	uint8_t *order*	Встановлює порядок бітів SPI. Використовуємо значення BCM2835_SPI_BIT_ORDER_MSBFIRST
bcm2835_spi_setDataMode	uint8_t *mode*	Встановлює режим роботи SPI, полярність та фазу таймера. Використовуємо значення BCM2835_SPI_MODE3
bcm2835_spi_setClockDivider	uint16_t *divider*	Встановлює частоту роботи вбудованого таймера. Використовуємо значення BCM2835_SPI_CLOCK_DIVIDER_65536
bcm2835_spi_chipSelect	uint8_t *cs*	Обирає пін, до якого приєднаний ведений пристрій. Для плати Export More це BCM2835_SPI_CS1
bcm2835_spi_setChipSelectPolarity	uint8_t *cs*, uint8_t *active*	Встановлює полярність передачі для вказаного піна. Використовуємо, відповідно, BCM2835_SPI_CS1 та LOW
bcm2835_spi_transfern	char * *buf*, uint32_t *len*	Передає *len* байтів із буфера *buf* на ведений пристрій та отримує стільки ж байтів у відповідь. Вони також записуються у буфер *buf*
bcm2835_spi_end	-	Завершує усі операції з SPI та повертає піни у початковий стан. Необхідна в кінці програми

ДОДАТОК В
ОПИС STM32F4DISCOVERY

STM32F4DISCOVERY – це налагоджувальна плата, яка була створена для більш зручного вивчення мікроконтролерів серії STM32F4.

Компоненти налагоджувальної плати:

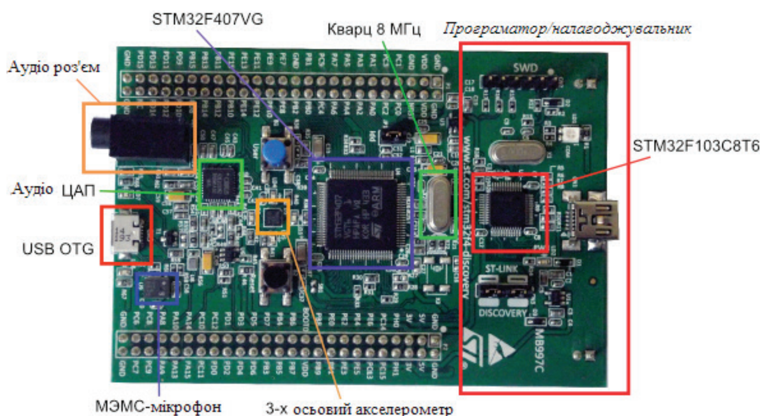

Рисунок В.1 – Налагоджувальна плата

З іншої сторони плати знаходяться штирові виводи.

Основні характеристики плати:

1. 32-бітний мікроконтролер STM32F407VGT6 з ядром ARM Cortex-M4F з 1 Мб пам'яті програм та 193 Кб ОЗУ в 100-вивідному корпусі LQFP100 з тактовою частотою 168 МГц. Вбудовані операції с плаваючою крапкою (FPU).
2. Вбудований програматор/відлагоджувальник ST-LINK/V2 з можливістю вибору режиму роботи (дозволяє програмувати зовнішні мікросхеми, використовуючи SWD-коннектор для програмування та відлагоджування)
3. Живлення плати: через шину USB або від зовнішнього 5В джерела живлення. Живлення для зовнішніх пристроїв: 3В та 5В.

4. 3-осьовий MEMC акселерометр на базі мікросхеми LIS302DL компанії ST.
5. Всенаправлений цифровий MEMC мікрофон на базі мікросхеми MP45DT02 компанії ST.
6. Аудіо ЦАП CS43L22 з вбудованим підсилювачем класу D.
7. Вісім світлодіодів: LD1(червоний/зелений) для індикації активності шиниUSB, LD2(червоний) для живлення 3.3В, 4 діода користувача: LD3(помаранчевий), LD4(зелений), LD5(червоний) та LD6(синій), 2 діода USB OTG: LD7(зелений) для VBus та LD8(червоний) при перевантаженні.
8. Дві кнопки (Reset та User).
9. USB OTG з раз'ємом micro-AB.
10. Вивідні колодки для всіх контактів вводу/виводу мікроконтролера для швидкого підключення до макетної плати та простого проведення вимірювань.

Значний плюс – це наявність у мікроконтролері модуля для роботи з числами з плаваючою крапкою, що збільшує швидкість обробки в додатках пов'язаних, наприклад, зі спектральним аналізом або для алгоритмів орієнтації.

Недолік – відсутність роз'єму JTAG для тестування з використанням зовнішнього програматора ST-LINK.

Структурна схема плати зоюражна на рисунку В.2.

Згідно схеми через діод Шоттки D3 напруга +3 В не строго стабілізована. Струм навантаження через контакт «*+3V*» не повинен перевищувати 150 мА. Якщо замість *J1* підєднати амперметр, то можливо помітити що враховуючи споживаний мікроконтролером струм відображає 80-130мА.

Живлення 5V подається через міні-*USB* роз'єм *«CPU»* з комп'ютера. Через діод *D1* на мікро-*USB* роз'єм *«OTG»* буде трішки менша напруга.

На колодці штирових контактів є вихід «*+5 V*» до якого можливо під'єднати зовнішнє джерело живлення. Завдяки діоду *D1* є можливість одночасно використовувати зовнішній та внутрішній джерела живлення.

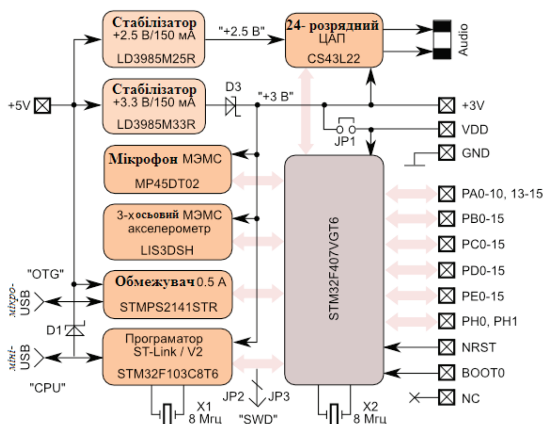

Рисунок В.2 – Структурна схема плати

Програматор *ST-Link* реалізований на мікроконтролері *STM32F103C8T6*. Основний програмований мікроконтролер приєднується для програмування через два джампера *J2* та *J3*. Для програмування зовнішніх мікросхем, використовуючи роз'їм «*SWD*» ці джампери слід видалити.

Для роботи цифрового МЕМС-мікрофона, 3-х осьового акселерометра и аудіо ЦАП задіяні кілька ліній портів мікроконторолера. Дані передаються по шинам *SPI*, *I2C*, *I2S*.

В таблиці В.1 наведені вільні входи та їх призначення.

Таблиця В.1 Вільні входи та їх призначення

Вихід	**Призначення**
PA0	Кнопка «User»
BOOT0	Вхід бутлоудера, сигнал*BOOT0*
PB2	Вхід бутлоудера, сигнал*BOOT1*
PA1-PA3, PA8, PA15, PB0, PB1, PB4, PB5, PB7, PB8, PB11, PB13-PB15, PC1, PC2, PC4-PC6, PC8, PC9, PC11, PD0-PD3, PD6-PD11, PE2, PE4-PE15	Вільні лінії*I/O*, максимальне навантаження ±25 мА, pull-up/down резистори 30...50 кОм (всього 46 ліній)

Продовження таблиці В.1 Вільні входи та їх призначення

Вихід	**Призначення**
PB12	Вільна лінія з pull-up/down резистором 8...15 кОм
PC13	Вільна лінія з навантаженням ±3 мА
PA5-PA7, PE0, PE1, PE3	3-х осьовий акселерометр LIS3DSH
PA9-PA12, PC0, PD5	Роз'єм мікро-USB(OTG)
PA13, PA14, PB3	Роз'єм програматораSWD
PB10, PC3	Вбудований цифровий мікрофон MP45DT02
PC14, PC15	Кварцевий резонатор 32 кГц
PD12-PD15	Зелений, помаранчевий, червоний, синій світлодіоди
PH0, PH1	Кварцевий резонатор 8 МГц для МК
NRST	Зовнішній початковий сброс МК
+3V, +5V, VDD, GND, NC	Ланцюги живлення 3 В, 5 В, МК, «земля», порожній контакт

ДОДАТОК Г
ОПИС СИСТЕМНОЇ ПЛАТИ DIGILENT BASYS2

BASYS2 – відлагоджувальна плата компанії Digilent на базі програмованої логічної інтегральної схеми (ПЛІС) сімейства Spartan-3E XC3S250E с ємністю 250 000 вентилів (рис. Г.1).

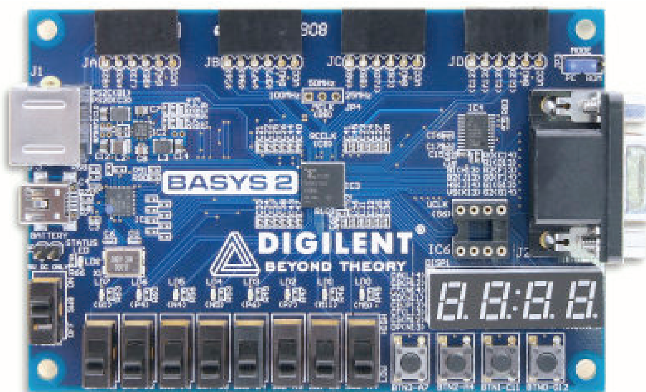

Рисунок Г.1 – Блок-схема плати Basys2

На плату встановлено ланцюг живлення, конфігураційна пам'ять, USB-контролер Full-speed, інтерфейси PS / 2 і VGA, 8 світлодіодів, 4 × 7-сегментних індикатора, 4 спеціальні кнопки, 8 перемикачів, що дозволяє розробнику відразу приступити до проектування майбутнього устрою.

Для плати є велика кількість периферійних модулів (АЦП, ЦАП, управління двигунів, датчики), що підключаються до 6-вивідного роз'єму розширення (табл. Г.1). Для програмування плати можна використовувати фірмовий софт Xilinx ISE WebPack або софт від Digilent Adept.

Відмінні риси плати:

- ПЛІС: XC3S250E (Spartan-3E);
- кількість вентилів: 250 000;
- 18-розрядні умножители;
- конфігураційна пам'ять Flash ROM для зберігання конфігурації ПЛІС XCF02;

– посадочне місце для установки кварцу 25/50/100 МГц;
– 8 світлодіодів;
– 4 × 7-сегментних індикатора;
– 4 спеціальні кнопки;
– 8 перемикачів;
– порти: PS / 2 і VGA (8 розрядів);
– порти розширення: 4х 6-вивідних порту;
– захист від ESD і КЗ на всіх висновках I / O.

Таблиця Г.1 – Основні параметри плати Basys2

Параметр	**Значення**
Інтерфейс підключення	USB
Ядро базового елементу	ПЛИС
Базовий компонент	XC3S250E
Розрядність, біт	8/16
Допоміжний компонент	AT90USB2
LTC3545	
RAM	
Цільове напруга, В	3.3 / 2.5 / 1.2
Напруга живлення, В	5.0
Джерело живлення	USB/зовнішній

На плату встановлено ланцюг живлення, конфігураційна пам'ять, USB-контролер Full-speed, інтерфейси PS / 2 і VGA, 8 світлодіодів, 4 × 7-сегментних індикатора, 4 спеціальні кнопки, 8 перемикачів, що дозволяє розробнику відразу приступити проектування майбутнього устрою.

Для плати є велика кількість периферійних модулів (АЦП, ЦАП, управління двигунів, датчики), що підключаються до 6-вивідного роз'єму розширення. Для програмування плати можна використовувати фірмовий софт Xilinx ISE WebPack або софт від Digilent Adept.

Побудована навколо масиву Xilinx Spartan-3E Field Programmable Gate і Atmel AT90USB2 контролера USB, плата Basys2 забезпечує повний, готовий до використання апаратний підхід для створення схем, починаючи від основних логічних пристроїв до складних контролерів. Велика кількість

на платі пристроїв введення / виводу і всі необхідні схеми підтримки FPGA дозволяють створювати незліченні проекти без необхідності будь-яких інших компонентів (рис. Г.2).

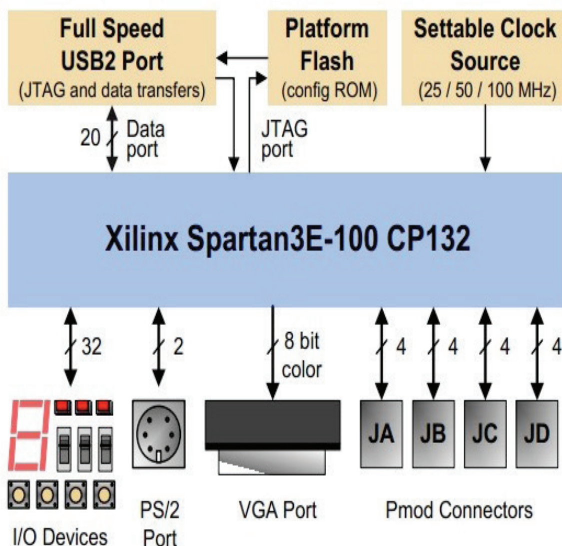

Рисунок Г.2 – Блок-схема плати Basys2

FPGA (скорочено від «Field-Programmable Gate Array» – з англ. «Програмована користувачем вентильная матриця») – напівпровідниковий пристрій, що може бути налаштоване виробником або розробником після виготовлення і який є однією з архітектурних різновидів ПЛІС.

Особливості плати Basys2:

– 100,000-gate Xilinx Spartan 3E FPGA;

– Atmel AT90USB2 повній швидкості USB2 порт, що забезпечує плату живленням і інтерфейсом передачі даних для програмування;

– Xilinx Platform Flash ROM для зберігання конфігурацій FPGA;

– 8 світлодіодів, 4-значний 7-сегментний дисплей, 4 кнопки, 8 перемикачів з інтерфейсом PS / порт 2 і 8-бітний VGA порт;

– встановлюються користувачем годинник (25/50/100 МГц), а також роз’єм для других годин;

– чотири 6-контактних роз’ємів;

– ESD і захист від короткого замикання на всіх сигналів введення / виводу.

Чотири стандартних роз’ема дозволяють конструкції розширюватися поза плати Basys2 за допомогою макетів, схем власних наборів плат, або Pmods (Pmods недорогих аналогових і цифрових модулів введення / виведення, які пропонують A / D & D / A перетворення, приводи двигунів, входи для датчиків і багато інші функції). Сигнали на 6-контактних роз’ємах захищені від електростатичних розрядів і коротких замикань, забезпечуючи довгий термін служби в будь-яких умовах. Basys2 плата працює спільно з усіма версіями інструментів Xilinx ISE, в тому числі вільного WebPack. Він поставляється з кабелем USB, що забезпечує живлення і інтерфейс програмування, тому ніякі інші джерела живлення або кабелі для програмування не потрібні.

Basys2 плата може отримувати живлення і програмуватися за допомогою USB2 порту. У вільному доступі на базі персонального комп’ютера (ПК) програмне забезпечення Digilent автоматично виявляє плати Basys2, забезпечує програмний інтерфейс для FPGA і платформи Flash ROM, а також дозволяє передає призначені для користувача дані.

Basys2 плата розрахована на роботу з вільним системи автоматизованого проектування (САПР) ISE WebPack від Xilinx. WebPack може бути використаний для визначення схем з використанням схеми або HDLs, моделювання і синтезування ланцюгів, а також створення запрограмованих файлів.

Г.1.1 Живлення плати

Basys2 плати, як правило, працюють з кабелем USB, але з’єднувач батареї також розроблений таким чином, щоб можна було використовувати зовнішні джерела (рис. Г.3). Щоб використовувати USB Power, необхідно просто підключити кабель USB. Для живлення Basys2 з використанням акумуля-

тора або іншого зовнішнього джерела, необхідно підключити акумуляторну батарею 3.5В – 5.5В (або інше джерело живлення) для 2-контактного, 100-мл роз'єму, розташованого на батареї (три АА елемента в послідовності забезпечать необхідні 4.5+ / – Вольта). Напруги вище, ніж 5,5 на будь-якому з роз'ємів живлення може привести до серйозних пошкоджень.

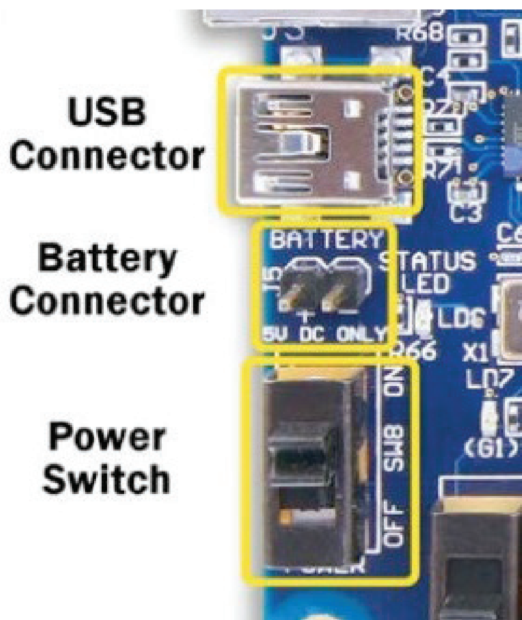

Рисунок Г.3 – Ланцюг живлення плати Basys2

Вхідна потужність проходить через вимикач живлення (SW8) до чотирьох 6-контактних роз'ємів і регулятору напруги Linear Technology LTC3545. LTC3545 виробляє основну живлення 3,3 для плати, а також виробляє 2.5 В і 1.2 В напруги, необхідні для FPGA. Сумарний струм плати залежить від конфігурації FPGA, тактової частоти і зовнішніх зв'язків. Необхідний струм можна збільшувати, якщо налаштовувати в FPGA більші схеми або якщо додавати периферійні плати.

Basys2 плата використовує чотири шари PCB, з внутрішніми шарами, відведені VCC і GND площин. FPGA та інші мікросхеми на платі мають великі додаткові керамічні блокувальні

конденсатори, розташовані якомога ближче до кожного VCC штифту, в результаті чого забезпечується низький рівень шуму блока живлення.

Г.1.2 Конфігурація

Після включення, FPGA на платі Basys2 повинен бути налаштований, перш ніж він може виконати будь-які функції. Під час налаштування, невеликий файл передається в пам'яті в FPGA, щоб визначити логічні функції і схеми з'єднань. Вільна САПР ISE/WebPack від Xilinx може бути використана для створення бітових файлів з VHDL, Verilog або вихідних файлів на основі схем.

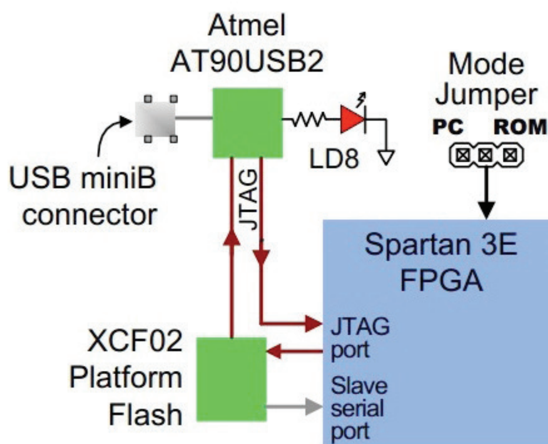

Рисунок Г.4 – Програмування плати Basys2

ПК-програма Digilent під назвою Adept може бути використана для конфігурації FPGA з будь-яким обраним двійкового файлом, який зберігається на комп'ютері. Adept використовує кабель USB для передачі обраного файлу з комп'ютера в FPGA (через порт програмування FPGA JTAG). Adept може запрограмувати невеликий файл в незалежне ПЗУ на платі під назвою «Platform Flash». Після програмування Platform Flash може автоматично передавати збережений бітовий файл в

FPGA на наступного включення живлення або скидання події, якщо диск має режим Jumper (JP3). FPGA залишатиметься сконфігурованим, поки не буде скинутий. Флеш-пам'ять платформи буде зберігати файл, поки не вона буде перепрограмовано, незалежно від потужності.

Для програмування плати Basys2 (рис. Г.4), необхідно встановити перемикач на ПК і підключіть кабель USB до плати.

Запустити ПЗ Adept (рис. Г.5) і очікувати визначення FPGA і платформи Flash ROM. Використовуючи функцію попереднього зв'язати потрібний .BIT файл з FPGA і / або бажані .mcs файли з платформи Flash ROM. Щоб запрограмувати плату, необхідно натиснути правою кнопкою миші на пристрої і вибрати функцію «Програма». Конфігураційний файл буде відправлений на FPGA або платформу Flash, і потім ПЗ буде вказувати, чи було програмування успішним. «Індикатор стану» LED (LD_8) також буде блимати після успішного налаштування FPGA.

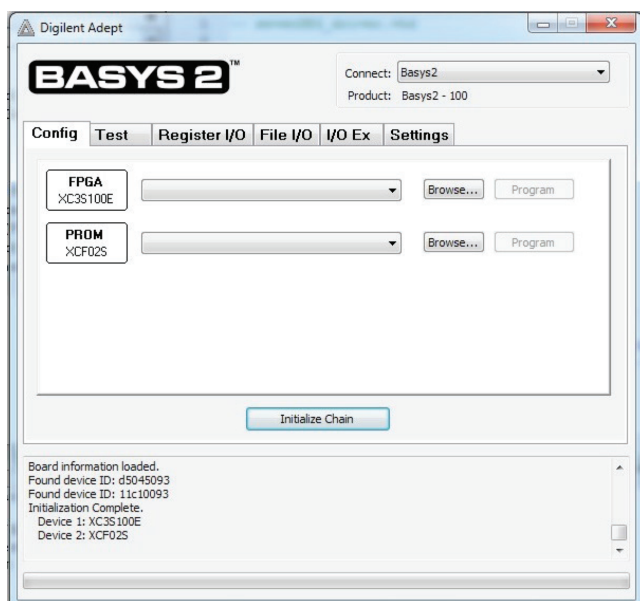

Рисунок Г.5 – Програмування плати Basys2 за допомогою ПЗ Adept

Г.2 Робота з периферійними пристроями

Програмована плата Basys2 може взаємодіяти з такими видами периферійних пристроїв, як дисплей, клавіатура, монітор, миша та інші.

1.2.2 Призначені для користувача входи і виходи I/O

Чотири кнопки і вісім перемикачів передбачені як входи схеми (рис.1.6). Кнопки генерують низький рівень сигналу, високий рівень сигналу генерується тільки при натисканні кнопки.

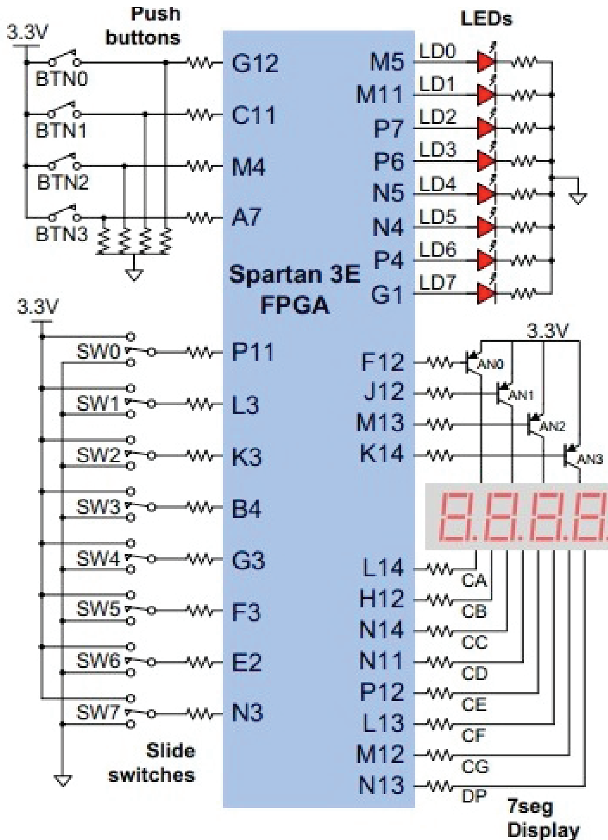

Рисунок Г.6 – Basys2 I/O схема

Повзункові перемикачі генерують високі або низькі сигнали в залежності від положення. Кнопки і перемикачі мають резистори для захисту від коротких замикань (коротке замикання може статися, якщо FPGA контакт кнопки або перемикача був ненавмисно визначено як вихід).

Вісім світлодіодів і семісегментний чотиризначний світлодіодний дисплей передбачені як виходи схеми. Світлодіодні аноди наводяться в рух від FPGA через струмообмежуючі резистори: вони будуть горіти, коли '1' записується у відповідний FPGA пін. Дев'ятий світлодіод передбачений в якості живлення світлодіодного індикатора, і десятий світлодіод (LD-D) загоряється в будь-який час, коли плата була успішно запрограмована.

Г.2.1 Семисегментний дисплей

Кожна з чотирьох цифр семисегментного світлодіодного дисплея складається з семи світлодіодних сегментів, розташованих за шаблоном «figure 8». Світлодіоди сегментів можуть бути індивідуально висвітлені, так що будь-який з 128 шаблонів можуть бути відображені у вигляді цифри при підсвічуванні певних сегментів LED. З цих 128 можливих моделей, десять відповідних десятковим цифрам є найбільш корисними.

Аноди з семи світлодіодів, що утворюють кожну цифру пов'язані в один вузол схеми із загальним анодом, а катоди світлодіода залишаються розділеними. Загальні сигнали анода доступні у вигляді чотирьох вхідних сигналів для 4х розрядного дисплея. Катоди подібних сегментів на всіх чотирьох дисплеях з'єднані в сім вузлів схеми, зафіксованих від CA до CG (так, наприклад, чотири катода «D» з чотирьох цифр, групуються разом в один вузол ланцюга під назвою «CD»). Ці сім катодних сигналу доступні в якості вхідних сигналів на 4х розрядному дисплеї. Ця схема підключення сигналу утворює мультиплексованих дисплей, де катодні сигнали є загальними для всіх цифр, але вони можуть висвітлювати тільки сегменти цифри, для якої підтверджується відповідний анодний сигнал.

Схема контролера дисплея сканування може бути використана для відображення на цьому дисплеї чотиризначного

номера. Ця схема управляє анодними сигналами і відповідно моделлю катода кожної цифри в повторюваної, безперервній послідовності, зі швидкістю оновлення швидше, ніж може помітити людське око. Кожна цифра підсвічується лише чверть часу, але як так очей не може сприймати потемніння цифри, перш ніж вона знову загориться, цифра здається постійно світиться. Якщо швидкість відновлення сповільнюється до одпределенной точки (близько 45 герц), то більшість людей починають бачити на дисплеї мерехтіння.

Для того, щоб кожна з чотирьох цифр рівномірного підсвічувалася всі чотири цифри повинні здаватися в інтервалі від 1 до 16 мс (при частоті оновлення екрану 1KHz 60 Гц). Наприклад, в схемі при оновленні 60 Гц, весь дисплей буде оновлюватися кожні 16мс, і кожна цифра буде освітлена для ¼ циклу оновлення, або 4 мс. Контролер повинен буде гарантувати, що використовується правильний шаблон катода, коли задається відповідний сигнал анода.

Щоб проілюструвати цей процес, наприклад, якщо для AN1 установалени одночасно з CB і CC, то «1» буде відображатися в першій цифрі. Далі, якщо для AN2 встановлені одночасно з CA, CB і CC то буде відображатися «7». Якщо на A1 для CB, CC постійно подається сигнал кажді 4 мс, а потім на A2 для CA, CB, CC постійно подається сигнал кожні 4 мс в, на дисплеї в перших двох цифрах з’явиться напис «17». На малюнку 1.7 показаний приклад тимчасової діаграми для чотиризначного контролера семи сегментів.

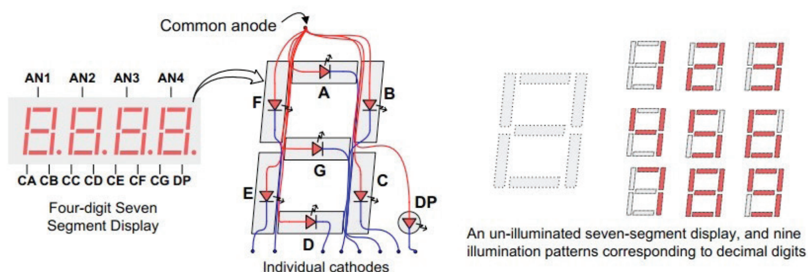

Рисунок Г.7 – Семисегментний дисплей

Г.2.2 Клавіатура

Клавіатура використовує драйвери з відкритим колектором, тому підключений керуючий пристрій (системна плата) може керувати клавіатурою по двухпроводной шині (якщо керуючий пристрій не надсилатиме дані на клавіатуру, то можуть використовуватися тільки порти для введення).

Клавіатури типу PS / 2 використовують скан-коди для взамодействия з ключами натиснутих даних. Кожному ключу привласнюється код, який надсилається щоразу, коли натиснута кнопка; якщо ключ утримується в натиснутому положенні, код сканування відправлятиметься повторно приблизно кожні 100 мс. При відпуску клавіші, відправляється код «F0», а за потім код перевірки вивільняється ключа. Якщо ключ може бути «зсунутим» для створення нового символу (наприклад великої літери), то символ зсуву посилається на додаток до коду сканування, і хост повинен визначити, який ASCII-символ використовувати. Деякі клавіші, так звані розширені ключі, відправляють «E0» перед кодом сканування (і вони можуть посилати більше одного коду сканування). При відпуску розширеної клавіші, код «E0 F0» в верхньому регістрі посилається, після якого слід код сканування. Скан-коди для більшості клавіш показані на малюнку 1.8.

Рисунок Г.8 Скан-коди клавіатури

Хост-пристрій також може передавати дані клавіатурі. Нижче наведено короткий список деяких загальних команд, які може послати хост (табл. 1.2).

Таблиця Г.2 – Керуючі команди від Basys2 до клавіатури

ED	Налаштування Num Lock, Caps Lock і Scroll Lock світлодіодів. Клавіатура повертає «FA» після отримання «ED», а потім хост посилає байт для установки стану світлодіодного індикатора: Біт 0 встановлює Scroll Lock; біт 1 встановлює режим Num Lock; і Біт 2 встановлює Caps Lock. Біти з 3 по 7 ігноруються.
EE	Висновок (тестовий). Клавіатура повертає «EE» після отримання «EE».
F3	Налаштування сканування частоти повторення коду. Клавіатура повертає «F3» на прийом «FA», а потім хост посилає другий байт, щоб встановити швидкість повтору.
FE	Повторна відправка. «FE» направляє на клавіатуру запит для повторного відправлення останнього скан-коду.
FF	Скидання. Перевстановлення клавіатури.

Клавіатура може передавати дані на хост тільки тоді, коли обидва і дані і годинник лінії є високими (або в режимі очікування). Так як хост є «провідним пристроєм шини», то клавіатура повинна перевірити, чи буде хост посилає дані перед манірулірованія шиною. Для полегшення цього завдання, годинник лінія використовується як сигнал типу «очистити, щоб відправити». Якщо хост переводить лінію годин до низького рівня, то клавіатура не повинна відправляти будь-які дані, поки годинник не будуть скинуті.

Клавіатура відправляє дані на хост 11-розрядними словами, які містять стартовий біт'0 ‘, а потім 8-біт скан-коду (LSB першим), з подальшим непарних бітом парності і завершуватися стоп-бітом ’1‘. Клавіатура генерує 11 тактових переходів (на рівні близько 20 – 30кГц), при передачі даних.

Г.2.3 Мишка

Мишка генерує сигнал для годин і даних при переміщенні; в іншому випадку, ці сигнали залишаються в логічній «1».

Кожен раз, коли миша переміщується, три 11-розрядних слів передаються від миші до хост-пристрої (рис. 1.9).

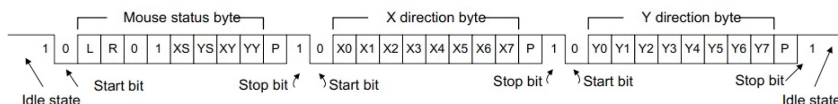

Рисунок Г.9 Формат даних миші

Кожен з 11-бітових слів містить стартовий біт «0», а потім 8 біт даних (LSB першим), з подальшим непарних бітом парності, і закінчується стоп-бітом '1'. Таким чином, кожна передача даних містить 33 біта, де стартові біти 0, 11, і 22 '0', а біти 11, 21, і 33 індикатори стоп-біта. Три 8-бітових полів даних містять дані руху, як показано на малюнку вище. Дані вірні по задньому фронту тактового сигналу, і тактовий період становить від 20 до 30 кГц.

Миша переміщається по відносній системі координат, в якій при переміщенні миші вправо генерується позитивне число в поле X, при переміщенні вліво генерується негативне число. Точно так же, переміщаючи мишу вгору генерується позитивне число в поле Y, а переміщення вниз являє собою негативне число (біти XS і YS в байті стану є біти знака – '1' вказує негативне число). Величина числа X і Y представляють швидкість руху миші – чим більше число, тим швидше миша рухається (біти XV і YV в байті стану є індикаторами переповнення рух – '1' означає, що переповнення сталося), Якщо миша безперервно рухається, то 33-бітові сигнали передаються приблизно кожні 50 мс. Поля L і R в байті є індикаторами натискання лівої чи правої кнопки миші ('1' вказує на те, кнопка натиснута).

Г.2.4 Порт PS/2

Для підключення миші або клавіатури PS / 2 використовується 6-контактний роз'єм міні-DIN. Роз'єм PS / 2 підключений до джерела живлення + 5В.

І миша, і клавіатура використовують двухпроводную послідовну шину (годинник і дані) для зв'язку з хост-пристроєм. Обидва використовують 11-бітові слова, які включають в себе запуск, зупинку і біт парності, але пакети даних організовані по-різному, і інтерфейс клавіатури дозволяє вести двосторонню передачу даних (тому хост-пристрій може висвітлювати світлодіоди стану на клавіатурі). Синхронізація шини показана на рисунке 1.10.

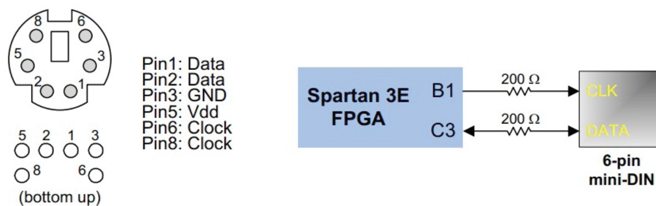

Рисунок Г.10 PS/2 connector and Basys2 PS/2 circuit

Тактові сигнали і сигнали даних наводяться в рух тільки тоді, коли відбувається передача даних, а в іншому випадку вони знаходяться в «холостому» стані з логікою '1'. Синхронізація визначає вимоги сигналів від миші до хосту зв'язку і двонаправлені зв'язку з клавіатурою. Схема інтерфейсу PS / 2 може бути реалізована в FPGA для створення інтерфейсу клавіатури або миші.

Г.2.5 Монітор та порт VGA

VGA (англ. Video Graphics Array) – компонентний відеоінтерфейс, який використовується в моніторах і відеоадаптерах. Випущений IBM в 1987 році для комп'ютерів PS / 2 Model 50 і більше старших.

Відеоадаптер VGA, на відміну від попередніх відеоадаптерів IBM (MDA, CGA, EGA), використовує аналоговий сигнал для передачі інформації кольорів. Перехід на аналоговий сигнал був зумовлений необхідністю скорочення числа проводів у кабелі. Також аналоговий сигнал давав можливість використовувати VGA-монітори з наступними відеоадаптерами, які можуть виводити більшу кількість кольорів.

Офіційним послідовником VGA став стандарт IBM XGA, фактично ж він був заміщений різними розширеннями до VGA, відомими як «Super VGA» (SVGA).

Термін VGA також використовується для позначення 15-контактного роз'єму VGA для передачі аналогового відеосигналу при різних дозволах.

VGA (роз'єм, DE15F) – 15-контактний субмініатюрний роз'єм для підключення аналогових моніторів за стандартом VGA.

VGA – аналоговий інтерфейс, розроблений в 1987 році і призначений для моніторів на електронно-променевих трубках (ЕПТ). Також цим інтерфейсом оснащуються деякі програвачі DVD і багато плазмові та РК-телевізори. VGA передає сигнал через підрядник, при цьому зміна напруги означає зміну яскравості (напруга сигналу становить 0,7-1 В), для ЕПТ воно означає зміну інтенсивності променя електронних гармат кінескопа (і, відповідно, яскравість світлової плями на екрані).

В даний час VGA вважається застарілим і активно витісняється цифровими інтерфейсами DVI, HDMI і DisplayPort. Найбільші виробники електроніки Intel і AMD оголосили про повну відмову від підтримки VGA в 2015 році.

Більшість моніторів, вже не мають роз'єму VGA, підключаються до відеоадаптера з VGA виходом через роз'єм DVI, за допомогою перехідника, оскільки частина ліній роз'єму DVI з метою сумісності є інтерфейсом VGA (за винятком формату DVI-D, в якому аналогові лінії відсутні).

На рис. 1.11 зображений роз'єм-мама. Нумерація в списку відповідає цифрам на малюнку.

Опишемо распиновку п'ятнадцяти-пинового VESA DDC2 / E-DDC роз'єму:

1. RED – червоний канал відео;
2. GREEN – зелений канал відео;
3. BLUE – синій канал відео;
4. ID2 / RES – раніше другої біт ID монітора, став зарезервований з появою E-DDC;

5. GND – земля (також, горизонтальна синхронізація);

6. RED_RTN – земля червоного каналу;

7. GREEN_RTN – земля зеленого каналу;

8. BLUE_RTN – земля синього каналу;

9. KEY / PWR – раніше ключ, зараз +5 В постійного струму;

10. GND – земля (також, вертикальна синхронізація, DDC);

11. ID0 / RES – раніше нульовий біт ID монітора, став зарезервований з появою E-DDC;

12. ID1 / SDA – раніше перший біт ID монітора, став використовуватися для I²C з появою DDC2;

13. HSync – горизонтальна синхронізація;

14. VSync – вертикальна синхронізація;

15. ID3 / SCL – раніше третій біт ID монітора, лінія тактирования I²C з появою DDC2.

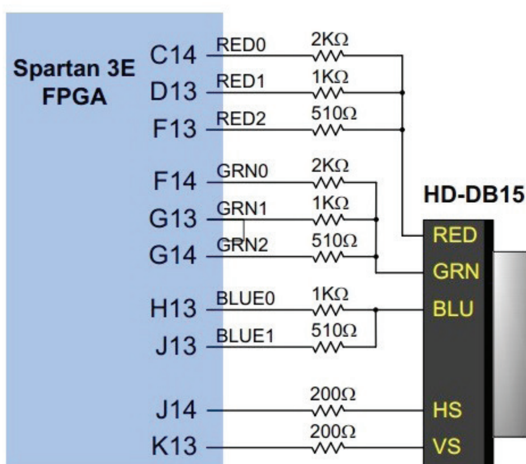

Рисунок Г.11 П'ятнадцяти-піновий VGA роз'єму для Basys2

VGA (так само, як і EGA) складається з наступних основних підсистем:

1. Графічний контролер (Graphics Controller), за допомогою якого відбувається обмін даними між центральним процесором і відеопам'яттю. Має можливість виконувати бітові операції над переданими даними.

2. Відеопам'ять (Display Memory), в якій розміщуються дані, які відображаються на екрані монітора. 256 кБ DRAM розділені на чотири колірні шару по 64 кБ.

3. Послідовний перетворювач (Serializer або Sequencer) – перетворює дані з відеопам'яті в потік бітів, що передається контролеру атрибутів.

4. Контролер атрибутів (Attribute Controller) – за допомогою палітри перетворює вхідні дані в колірні значення.

5. Синхронизатор (Sequencer) – управляє тимчасовими параметрами відеоадаптера і перемиканням колірних шарів.

6. Контролер ЕПТ (CRT Controller) – генерує сигнали синхронізації для ЕПТ.

На відміну від CGA і EGA, основні підсистеми розташовуються в одній мікросхемі, що дозволяє зменшити розмір відеоадаптера. У комп'ютерах PS / 2 відеоадаптер VGA інтегрований в материнську плату.

У стандартних текстових режимах символи формуються в осередку 9 × 16 пікселів, можливе використання шрифтів інших розмірів: 8-9 пікселів в ширину і 1-32 пікселя в висоту. Розміри самих символів, як правило, менше, так як частина простору йде на створення зазору між символами. Функція для вибору розміру шрифту в BIOS відділена від функції вибрати режим відео, що дозволяє використовувати різні комбінації режимів і шрифтів. Є можливість завантаження восьми і одночасного виведення на екран двох різних шрифтів.

У VGA BIOS зберігаються наступні види шрифтів і функції для їх завантаження і активації:

- 8 × 16 пікселів (стандартний шрифт VGA);
- 8 × 14 (для сумісності з EGA);
- 8 × 8 (для сумісності з CGA).

Як правило, ці шрифти відповідають кодової сторінці CP437. Також підтримується програмна завантаження шрифтів, яку можна використовувати, наприклад, для русифікації.

В наявності є таке стандартні режими:

- 40 × 25 символів, 16 кольорів, дозвіл 360 × 400 пікселів.
- 80 × 25 символів, 16 кольорів, дозвіл 720 × 400 пікселів.
- 80 × 25 символів, монохромний, дозвіл 720 × 400 пікселів.

Хоча в текстових режимах VGA одне знакоместо має ширину 9 пікселів, в даних знакогенератора визначаються тільки 8 з них (8 біт одного байта на рядок); пікселі правої колонки символьної матриці визначається автоматично: порожніми (для символів в діапазоні 0x00-0xAF і 0xE0-0xFF) або такими ж, як пікселі 8-й колонки (для символів псевдографіки в діапазоні 0xB0-0xDF).

Використовуючи шрифти менших розмірів, ніж стандартний 8 × 16, можна збільшити кількість рядків у текстовому режимі. Наприклад, якщо включити шрифт 8 × 14, то буде доступно 28 рядків. Включення шрифту 8 × 8 збільшує кількість рядків до 50 (аналогічно режиму EGA 80 × 43).

У текстових режимах для кожного осередку з символом можна вказати атрибут, що задає спосіб відображення символу. Існує два окремих набору атрибутів – для кольорових режимів і для монохромних. Атрибути кольорових текстових режимів дозволяють вибрати один з 16 кольорів символу, один з 8 кольорів фону і включити або відключити мерехтіння (можливість вибору мерехтіння можна замінити на можливість вибору одного з 16 кольорів фону), що збігається з можливостями CGA. Атрибути монохромних режимів збігаються з атрибутами, доступними у MDA, і дозволяють включати підвищену яскравість символу, підкреслення, мерехтіння, інверсію і деякі їх комбінації.

На відміну від своїх попередників (CGA і EGA) відеоадаптер VGA мав режим відео з квадратними пікселями (тобто, на екрані зі співвідношенням сторін 4: 3 співвідношення горизонтального та вертикального дозволів було також 4:3). У адаптерів CGA і EGA пікселі були витягнуті по вертикалі.

Стандартні графічні режими:

– 320 × 200 пікселів, 4 кольори.
– 320 × 200 пікселів, 16 кольорів.
– 320 × 200 пікселів, 256 кольорів (новий для VGA).
– 640 × 200 пікселів, 2 кольори.
– 640 × 200 пікселів, 16 кольорів.
– 640 × 350 пікселів, монохромний.
– 640 × 350 пікселів, 16 кольорів.

– 640 × 480 пікселів, 2 кольори. При вирішенні 640 × 480 пікселів має пропорції 1: 1 (новий для VGA).

– 640 × 480 пікселів, 16 кольорів (новий для VGA).

Монітор – конструктивно закінчений пристрій, призначений для візуального відображення інформації.

Сучасний монітор складається з екрану (дисплея), блоку живлення, плат управління та корпусу. Інформація для відображення на моніторі надходить з електронного пристрою, що формує відеосигнал (в комп'ютері – відеокарта). У деяких випадках в якості монітора може застосовуватися і телевізор.

За типом екрану розрізняють наступні види моніторів:

– ЕПТ – монітор на основі електронно-променевої трубки (англ. Cathode ray tube, CRT);

– ЖК – рідкокристалічні монітори (англ. Liquid crystal display, LCD);

– Плазмовий – на основі плазмової панелі (англ. Plasma display panel, PDP, gas-plazma display panel);

– Проектор – відеопроектор і екран, розміщені окремо або об'єднані в одному корпусі (як варіант – через дзеркало або систему дзеркал); і проекційний телевізор;

– LED-монітор – на технології LED (англ. Light-emitting diode – світловипромінювальних діод);

– OLED-монітор – на технології OLED (англ. Organic light-emitting diode – органічний світловипромінювальних діод);

– Віртуальний ретинальний монітор – технологія пристроїв виведення, що формує зображення безпосередньо на сітківці ока;

– Лазерний – на основі лазерної панелі (поки тільки впроваджується у виробництво).

Як приклад можна розглянути CRT (Cathode Ray Tube) монітори.

Найважливішим елементом монітора є кінескоп, званий також електронно променевою трубкою (основні конструкційні вузли кінескопа показані на рис Г.12). Кінескоп складається з герметичної скляної трубки, усередині якої знаходиться вакуум, тобто все повітря видалений. Один з кінців трубки

вузький і довгий – це горловина, а інший – широкий і досить плоский – це екран. З фронтального боку внутрішня частина скла трубки покрита люмінофором (luminophor). Як люмінофорів для кольорових ЕПТ використовуються досить складні склади на основі рідкоземельних металів – ітрію, ербію і т.п. Люмінофор – це речовина, яка випромінює світло при бомбардуванні його зарядженими частинками.

На рис. Г.13 показано приєднання монітора за допомогою VGA порту.

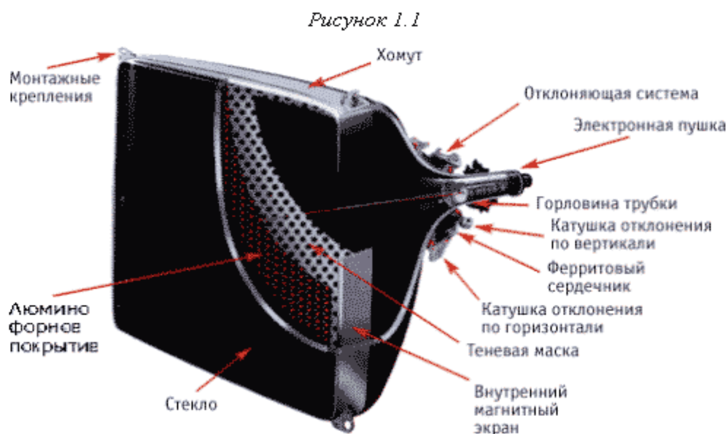

Рисунок Г.12 Конструкція ЕПТ-моніторів

ДОДАТОК Д
ОПИС VHDL

Vhdl (англ. Vhsic (very high speed integrated circuits) hardware description language) – мова опису апаратури інтегральних схем. Мова проектування VHDL є базовою мовою при розробці апаратури сучасних обчислювальних систем.

Був розроблений в 1983 році на замовлення міністерства оборони США з метою формального опису логічних схем для всіх етапів розробки електронних систем, починаючи модулями мікросхем і закінчуючи великими обчислювальними системами.

Спочатку мова призначалася для моделювання, але пізніше з нього було виділено синтезується підмножина. Написання моделі на синтезованих підмножині дозволяє автоматичний синтез схеми функціонально еквівалентної вихідної моделі. Засобами мови VHDL можливе проектування на різних рівнях абстракції (поведінковому або алгоритмічній, реєстрових передач, структурному), відповідно до технічного завдання та уподобаннями розробника. Закладена можливість ієрархічного проектування, максимально реалізує себе в екстремально великих проектах за участю великої групи розробників. Представляється можливим виділити наступні три складові частини мови: алгоритмічну – засновану на мовах ADA і PASCAL і додає мови VHDL властивості мов програмування; проблемно орієнтовану – по суті і звертають VHDL в мову опису апаратури; і об’єктно-орієнтовану, інтенсивно розвивається останнім часом.

Стандартами 1987, 1991, 1993, 1996, 1997, 1999, 2000, 2002 і 2008 рр. закріплені багато його удосконалення, так ,наприклад, починаючи зі стандарту VHDL-2000, мова набуває основи об’єктно-орієнтованої парадигми. Стандарт VHDL-93 є останнім, повністю підтримуваним засобами сапр стандартом [джерело не вказано 2192 дня].

Vhdl створений як засіб опису цифрових систем, однак існує підмножина мови – VHDL AMS (analog mixed signal), що дозволяє описувати як чисто аналогові, так і змішані, цифроаналогові схеми.

Мовою VHDL створені опису відкритих мікропроцесорів erc32 (sparc v7) і leon (sparc v8). Вихідний код доступний під ліцензіями lgpl і gpl відповідно.

На основі мови VHDL'2008 розроблена open source vhdl verification methodology (os-vvm), яка дозволяє реалізувати функціональне покриття і керовану генерацію псевдовипадкових тестів, що використовується при верифікації цифрових функціональних блоків. В рамках os-vvm написано кілька vhdl пакетів, з відкритими початковими кодами, які дозволяють досить просто виконувати генерацію псевдовипадкових тестів і інтелектуальне функціональне покриття в своїх проектах, використовуючи функції описані в пропонованих пакетах coveragepkg і randompkg. Os-vvm надає аналогічні можливості, які існують в інших мовах верифікації (systemverilog або e).

Д.1 Основні поняття VHDL

Розглянемо основних понять мови VHDL.

Інтерфейс – описує пристрій, як чорний ящик з входами і виходами, тобто головна його задача показати які входи і виходи є у пристрої для зв'язку із зовнішнім світом.

Архітектура – описує поведінку пристрою або розкриває його внутрішню структуру, тобто в архітектурі описується алгоритм функціонування пристрою.

Варто звернути увагу, що архітектура може бути описана в загальному випадку двома варіантами:

– Поведінковим стилем (описується алгоритм роботи пристрою).

– Структурним стилем (описується структура пристрою).

Поведінковий стиль зручно використовувати при описі елементів на низькому рівні ієрархії, а структурним на верхніх рівнях ієрархії, т.е.напісать багато маленьких пристроїв поведінкових стилем, а потім описати складаються з них пристрій структурним стилем, у вигляді зв'язків між ними.

Оператори мови – оператори в мові бувають послідовні, а бувають паралельні. Паралельні оператори вводяться для

того, щоб відобразити паралельність протікають в залозі процесів. Але будь-який паралельний оператор можна замінити спеціальним паралельним оператором процесу з послідовними операторами всередині його, він як раз для цього і призначений. За допомогою паралельних операторів описуються елементи схеми, які можуть працювати одночасно, при моделюванні кожному паралельному оператору ставиться у відповідність свій процес.

Сигнали – пов'язують між собою процеси. Вони є зовнішніми по відношенню до процесу, тобто процес може зчитувати сигнал і видавати значення в сигнал. Тому сигнали можуть оголошуватися тільки в області декларацій архітектури. Так само сигнали можуть зберігати значення, які необхідно передавати від процесу до процесу.

Змінні – змінні використовуються всередині паралельного оператора процесу і потрібні для опису алгоритму роботи процесу. Їх треба використовувати в тих випадках, коли не потрібно переносити інформацію від процесу до процесу.

Атрибути – це значення (характеристики), пов'язані з будь-якими об'єктами мови. Наприклад, типами, сигналами, змінними і т.д.

Типи – множини значень з якимись загальними характеристиками. Характеризуються типи набором значень, які можуть приймати об'єкти (сигнал, змінні) даного типу, а так же набором операцій, які можуть виконуватися з об'єктами даного типу.

Бібліотеки та пакети – пакети становлять собою структури в яких зберігаються описи різних функцій, процедур, компонентів, типів, констант і т.д, все пакети збираються в бібліотеки, які в наслідку підключаються до проектів.

Д.2 Структура проекту в VHDL

Структура проекту в VHDL представлена на рис. Д.1.

Як видно з малюнка, на одному рівні описується інтерфейс об'єкта проекту та архітектура об'єкта проекту, якщо потрібно до кожного інтерфейсу можна підключити потребується па-

кет. У свою чергу тіло архітектури складається з паралельних операторів, серед яких може бути оператор процесу, що містить послідовні оператори. Варто звернути увагу, що сигнали декларуються на рівні архітектури, а змінні на рівні окремих процесів і використовуються всередині їх.

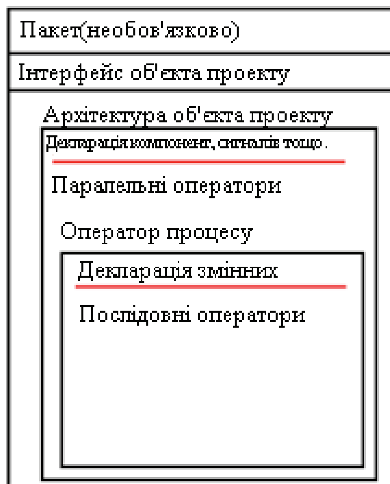

Рисунок Д.1 Структура проекта в VHDL

Д.3 Типи в VHDL

Всього є чотири класи типів: скалярний, складовою, файловий і контрольний (рис. Г.2).

Рисунок Д.2 Типи даних в VHDL

Зелененьким світлофором позначені типи, які синтезуються. Приклад оголошення типів:

– type boolean is (false, true); – тип boolean, як видно приймає 2 значення;

– type bit is (‘0’, ‘1’); – тип, являє собою, як би еквівалент одного розряду;

– type integer is range -2147483648 to 2147483647; – цілий тип integer, як видно представлений діапазоном значень;

– type real is range -1.0E308 to 1.0E308; – дійсний тип;

– type time is range -2147483647 to 2147483647 – фізичний тип, потрібен для завдання затримок при моделюванні;

```
units
 fs;
 ps = 1000 fs;
 ns = 1000 ps;
 us = 1000 ns;
 ms = 1000 us;
 sec = 1000 ms;
 min = 60 sec;
 hr = 60 min;
end units;
```

– type bit_vector is array (natural range <>) of bit; – Масив типу bit, запис natural range <> позначає, що розмір задається при оголошенні в діапазоні натуральних значень.

1.4 Способи завдання значень

У VHDL для завдання числових значень використовуються літерали (рис. Д.3) і декларації (рис. Д.4).

Рисунок Д.3 Літерали

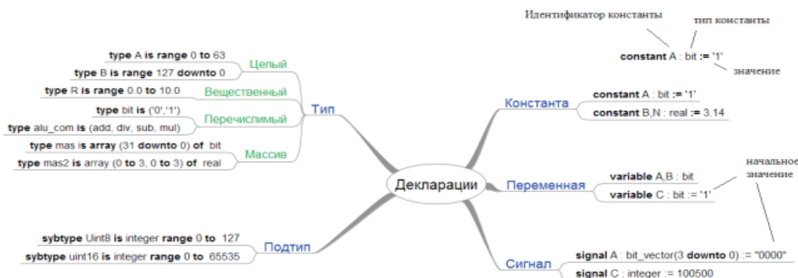

Рисунок Д.4 декларації

Операції, що використовуються в VHDL, представлені на рис. Д.5.

Клас операцій	Оператори					
Логічні	and	or	nand	nor	xor	xnor
Порівняння	=	/=	<	<=	>	>=
Додавання та конкатенація	+	-	&			
Присвоєння знаку	+	-				
Множення	*	/	mod	rem		
Змішані	**	abs	not			

Рисунок Д.5 Оператори VHDL

Д.4 Оператори мови VHDL

До послідовних операторів відносяться:

1. Оператор присвоювання значень змінної (: =) – A: = B and C;

2. Оператор присвоювання значень сигналу (<=) – A <= A and C;

При присвоєнні використовуються наступні затримки

а) inertial – інерційна затримка. Передача сигналу буде мати місце, якщо вхідний сигнал буде зберігати відповідний рівень протягом зазначеного відрізка часу – імітує час спрацьовування пристрою:

X <= inertial Y after 3 ns;

б) transport – транспортна затримка. Всі зміни сигналу буде присвоюватися з затримкою – імітує затримку на лініях:

X <= transport Y after 3 ns;

в) за замовчуванням використовується інерційна затримка:

X <= Y after 3 ns;

3. Оператор if – приклад умовного оператора:

```
if умова then
послідовні оператори
{elsif условие then
последовательные операторы}
[else
последовательные операторы]
end if;

--Приклад №1
if (A <B) then – порівнюємо A і B
 y <= X0; – Якщо умова = true
 else
 Y <= X1; – Якщо умова = false
 end if;

--Прімер №2
 if (A <B) then
 Y <= X0;
 elsif (A <C) then – замінює зв’язку else if (A <C) then Y <=
X1;
 else
 Y <= X2;
```

end if;

4. Оператор case – зручно використовувати при реалізації пристроїв описаних таблицею істинності:

case вираження is
when вибір => послідовні оператори
{When вибір => послідовні оператори}
end case;

Оператор вибирає одну з альтернатив, обрана альтернатива визначається значенням вирази. Вираз і вибір повинні бути одного типу (дискретного або одновимірний масив). Всі можливі вибори повинні бути описані або використано службове слово others (всі інші випадки).

case X is – в залежності від X видаємо в Y різні значення
when «00» => Y <= ‘0’; – Коли X = «00» і т.д.
when «01» => Y <= ‘1’;
when «10» => Y <= ‘1’;
when «11» => Y <= ‘1’;
when others => Y <= ‘0’; – Якщо використовуємо std_logic_1164, за рахунок розширеного алфавіту комбінацій буде багато більше 4.
end case;

5. Оператор loop – опис оператора представлено нижче:

[Мітка циклу:] [while умова | for ідентифікатор in діапазон дискретного типу]
loop
послідовні оператори
end loop;

У разі використання ключового слова while, спочатку обчислюється умова, якщо результат true, то виконуються послідовні оператори.

У разі використання ключового слова for, ідентифікатор визначає лічильник числа ітерацій циклу:

-- Приклад №1 – вважаємо суму всіх цифр від 0 до 9
s: = 0;
M1: for i in 0 to 9 – оператор повторення з for синтезується
loop
s: = s + i;

end loop;

-- Приклад 2 – так само вважаємо суму всіх цифр від 0 до 9

i: = 0;

M2: while (i <10) – можна синтезувати при певних умовах

loop

s: = s + i;

i: = i + 1;

end loop;

6. Оператор next – опис оператора представлено нижче:

next [мітка циклу] [when умова];

Оператор застосовується для переходу до наступної ітерації циклу.

--Приклад

i: = 0;

M2: while (i <10)

loop

s: = s + i;

next M2 when s = 3; – Пропустимо наступний рядок в разі S = 3, тобто додамо 2 до S два рази

i: = i + 1;

end loop;

7. Оператор exit – опис оператора представлено нижче:

exit [мітка циклу] [when умова];

Застосовується для дострокового завершення оператора циклу. Приклад такої ж, як і в попередньому випадку, тільки в результаті буде дострокове завершення циклу.

8. Оператор null – можна використовувати при синтезі, але це може сильно ускладнити схему. Чи не становить дій.

case X is

when «00» => Y <= ‘0’;

when «01» => Y <= ‘1’;

when «10» => Y <= ‘1’;

when «11» => Y <= ‘1’;

when others => Y <= null; – У всіх інших випадках нічого не робити

– Попередній рядок краще замінити на

– when others => Y <= ‘-’; – При всіх інших випадках нам

байдуже, з цього нехай синтезатор оптимізує схему як хоче
end case;

9. Оператор assert – можна застосовувати при налагодженні програм (при трасуванні), не синтезується:

assert умова [report вираз] [severity вираз];

Перевіряє, чи є умова істинним, і повідомляють про помилку, якщо умова є хибним (severity (ступінь серйозності) – NOTE, WARNING, ERROR, FAILURE).

- Приклад

assert (Y = ‘1’ and X = ‘1’) report «Incorrect signals» severity WARNING;

безумовно видається повідомлення

report «Happy new year» severity NOTE;

assert false report «Happy new year»;

10. Оператор wait – використовується тільки при моделюванні, при синтезі ігнорується (синтезувати можна тільки окремі випадки):

wait on список чутливості until умова for тайм-аут

Призупиняє процес до моменту, поки не зміниться деякий будильника чутливості процесу, в цей час буде вироблено обчислення умови.

Якщо умова true, виконання процесу відновлюється (тайм-аут встановлює максимальний час, після якого процес відновить своє виконання):

wait on A, B until (C = 0) for 50 ns;

Припустимо записувати одне або більше умов в операторі очікування:

wait on A, B;
wait until (C = 0);
wait for 50 ns;

Може бути більше одного оператора wait всередині оператора процесу.

Д.5 Xilinx ISE WebPack

Інтегроване середовище розробки Xilinx ISE WebPack (WebPack – варіант безкоштовної ліцензії) – містить [7]:

- Редактор коду HDL – Verilog і VHDL – з підсвічуванням і перевіркою синтаксису;

- Інструмент синтезу прошивки для ПЛІС Digilent Basys2 (Spartan 3E) – генерує файл з розширенням «.bit» в проекті;

- Симулятор ISim входить в комплект дистрибутива і інтегрований в середу розробки (дозволяє відразу переглядати графіки сигналів модулів в зручному графічному інтерфейсі);

- Вбудовані засоби програмування (заливки) прошивки на ПЛІС Digilent Basys2 вимагають додаткової настройки.

Набір інструментарію ISE Design Tools для проектування CPLD / FPGA від Xilinx дозволяє встановити і використовувати деякі функції і можливості безкоштовно. Цей режим установки пакета ISE Design Tools називається ISE WebPack. Далі розглянуто процес установки ISE WebPack і отримання для нього безкоштовної ліцензії по кроках .

1. Якщо у користувача ще немає облікового запису Xilinx, необхідно зареєструватися. Це необхідно, тому що отримання всіх посилань на закачування (і отримання ліцензій, навіть безкоштовних) походить від імені зареєстрованого користувача. Реєстрація безкоштовно доступна на сайті xilinx.com (пройдіть по посиланню Sign In у верхній частині головної сторінки сайту).

2. Завантажити дистрибутив ISE Design Tools. Посилання на закачування можна знайти на сторінці xilinx.com -> Downloads -> ISE Design Tools -> All Platforms -> Split installer Base Image. Дистрибутив поставляється у вигляді розділеного на частини TAR / GZIP-архіву, і займає близько 8 гігабайт. Цей дистрибутив підходить для установки як на Linux, так і на Windows 32 і 64 біта (я пробував встановлювати на Windows XP, 32-розрядний на Windows 7 64 біта).

3. Розпакувати архів в будь-яку тимчасову папку. Запустити cmd з правами системного адміністратора, і запустити в ньому установник xsetup.exe.

4. Коли установник запросить варіант установки, вибрати ISE WebPack. Цей варіант передбачає отримання безкоштовної постійної ліцензії від Xilinx на більшість інструментів проектування Xilinx. В процесі установки на всі питання майстра

відповідайте за замовчуванням, поки не дійде черга до отримання ліцензії.

Під час установки будуть встановлені драйвера для USB-адаптерів програмування Xilinx, тому перед установкою необхідно переконатися, що всі адаптери відключені від комп'ютера.

5. Коли установка завершиться, запуститься Xilinx License Configuration Manager. На першій закладці Acquire a License виберіть варіант Get Free Vivado / ISE WebPack License і натисніть Next. Запуститься браузер, який (після введення логіна і пароля облікового запису Xilinx) відкриє сторінку Product Licensing. На першій закладці Create New Licenses, в розділі Certificate Based Licenses поставте галочку на ISE WebPACK License, і натисніть на кнопку Generate Node-Locked License. Згенерує файл ліцензії, і ліцензію можна буде переглянути на закладці Manage Licenses (в списку буде ліцензія ISE WebPACK License).

6. Файл ліцензії Xilinx.lic автоматично буде вислано на вказаний при реєстрації e-mail. Якщо це чомусь не сталося, то можна в будь-який момент запросити повторну висилку файлу ліцензії, якщо натиснути на кнопку E-mail на сторінці Product Licensing -> Manage Licenses (кнопка з червоним конвертиком в нижній частині екрана, нижче списку ліцензій). Необхідно зберегти цей файл у будь-який зручний місце на диску, наприклад в папку інсталяції (зазвичай це папка на зразок C: \ Xilinx \ 14.7 \ ISE_DS \).

7. Перейти в вікно Xilinx License Configuration Manager, відкрити закладку Manage Licenses. Натиснути на кнопку Load License ..., і вибрати присланий файл ліцензії Xilinx.lic. Після цього список встановлених ліцензій оновиться, і користувач зможете безкоштовно користуватися більшістю можливостей Xilinx ISE WebPack.

Для користувачів WebPACK завжди активується функція WebTalk. WebTalk ігнорує налаштування користувача і процедури установки, коли генеруються конфігураційні дані програмованих схем логіки (bitstream) під керуванням ліцензії WebPACK. Якщо розробка заснована на пристрої, включено-

му в WebPACK, і доступна ліцензія WebPACK, то завжди буде використовуватися ліцензія WebPACK. Щоб змінити це необхідно переглянути Answer Record 34746.

АВТОРИ

ТАБУНЩИК Галина Володимирівна – к.т.н., доцент, професор Запорізького національного технічного університету. Закінчила Запорізький державний технічний унверситет за спеціальність програмне забезпечення автоматизованих систем, захистила дисертацію на звання кандидат технічних наук зі спецальності 05.13.03 – системи і процеси керування. Автор понад 100 наукових праць. Наукові інтереси – інженерія програмного забезпечення, верифікація інформаційних систем, програмування вбудованих систем, керування ризиками
galina.tabunshchik@gmail.com

Galyna TABUNSHCHYK – PhD, Prof of Software Tool Department of Zaporizhzhya National Technical Univerity. Graduated from Zaporizhzhya National Technical University with speciality Software Engineering, in 2004 finished PhD work in control systems and process. Have more than 100 scientific works. Scientific interests – Software Engineering, System Verification, Embeded Systems, Risk Management

КАПЛІЄНКО Тетяна Ігорівна – кандидат технічних наук, доцент кафедри програмних засобів Запорізького національного технічного університету. Закінчила магістратуру за спеціальністю «Програмне забезпечення автоматизованих систем», аспірантуру за спеціальністю «Інформаційні технології». Захищено дисертацію за темою «Інформаційна технологія динамічного планування та моніторингу процесу розроблення web-орієнтованих інформаційних систем» у 2015 році. Автор 33 наукових публікацій, 3 авторських свідоцтв і одного патенту. Основні напрямки наукових досліджень: аналіз та верифікація якості програмного забезпечення; керування програмними проектами; керування ризиками.
bragina.zntu@gmail.com

Tetiana KAPLIIENKO, Ph.D., an associate professor of Software Tools Department at Zaporizhzhya National Technical University. She has a master's degree in "Software Engineering". She finished PhD study in "Information technology", and defended thesis «Information technology for dynamic planning and monitoring the development process of web-oriented information systems» in 2015. She is the author of 33 scientific publications, 3 Certificates for invention and 1 patent. Main research fields: the analysis and verification for the software quality; software design managements; risks management.

ПЕТРОВА Ольга Анатоліївна, аспірант кафедри програмних засобів Запорізького національного технічного університету, завідуючий лабораторією GRID-технологій та хмарних обчислень. Закінчила магістратуру за спеціальністю «Комп'ютерні системи та мережі», навчається в аспірантурі за спеціальністю «Інформаційні технології». Наукові інтереси – віддалені лабораторії, системи позиціонування та навігації.

savenkoolja@mail.ru

Olga PETROVA, PhD student of Software Tools Department of Zaporizhzhya National Technical University, Head of Laboratory of GRID-technologies and Cloud Computing. She has a master's degree in «Computer systems and networks». Scientific interests – remote laboratory, positioning and navigation systems.

Galyna Tabunshchyk
Tetiana Kapliienko
Olga Petrova

Software Design and Modeling for Modern Informational Systems

Students Textbook

Co-funded by the
Tempus Programme
of the European Union

Навчальне видання

Табунщик Галина Володимирівна
Каплієнко Тетяна Ігорівна
Петрова Ольга Анатоліївна

Проектування та моделювання програмного забезпечення сучасних інформаційних систем

Навчальний посібник

Комп'ютерний набір	*Г. В. Табунщик*
Технічний редактор	*Л. А. Рябоконь*
Коректор	*Н. В. Чечеко*
Художник	*А. П. Кондаков*

Формат 60x84/16.
Папір офсетний. Гарнітура *Times*. Друк офсетний.
Підписано до друку 15.12.2016. Ум. друк. арк. 14,415.

Наклад 300 прим.

Видавництво «Дике Поле»
Україна, 69063, м. Запоріжжя, вул. Троїцька, 31-А.
Тел.: (061) 213–75–95; 213–75–05.

Свідоцтво суб'єкта видавничої справи
ЗЗ № 004 від 23.08.2001 р.

Табунщик Г. В.
Т-12 Проектування та моделювання програмного забезпечення сучасних інформаційних систем / Г. В. Табунщик, Т.І. Каплієнко, О.А. Петрова – Запоріжжя : Дике Поле, 2016. – 250 с.
ISBN 978-966-2752-07-0

Навчальний посібник містить підходи та засоби аналізу, проектування та моделювання програмного забезпечення сучасних інформаційних систем, зокрема, вбудованих систем.

Видання призначене для студентів комп'ютерних спеціальностей вищих навчальних закладів, а також може бути використаним аспірантами, науковими та педагогічними працівниками, фахівцями-практиками.

Навчальний посібник видано за підтримки проекту Tempus 544091-TEMPUS-1-2013-1-BE-TEMPUS-JPCR Розробка курсів з вбудованих систем з використанням інноваційних віртуальних підходів для інтеграції науки, освіти та промисловості в Україні, Грузії, Вірменії.

Проект фінансується при підтримці Європейської комісії. Зміст матеріалу відображає думку авторів та Європейська комісія не несе відповідальності за використання інформації що міститься в навчальному посібнику.

УДК 004.41 (075.8)
ББК 32.972-02 я73

ISBN 978-966-2752-07-0