МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЧЕРНІГІВСЬКА ПОЛІТЕХНІКА»
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ЕЛЕКТРОННИХ
ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

МЕТОДИЧНІ ВКАЗІВКИ

з проходження проектно-технологічної практики
для здобувачів вищої освіти
першого (бакалаврського)
рівня вищої освіти за спеціальністю
121 – "Інженерія програмного забезпечення"

Обговорено і рекомендовано
на засіданні кафедри інформаційних технологій та програмної інженерії
протокол № 1 від 31.08.2021 р.

Чернігів 2021

Інженерія програмного забезпечення. Методичні вказівки з проходження проектно-технологічної практики для здобувачів вищої освіти першого (бакалаврського) рівня вищої освіти / Укл. В.В. Нехай, М.М. Войцеховська. – Чернігів: НУ «Чернігівська політехніка», 2021. – 60 с., укр. мовою.

Укладачі: Нехай Валентин Валентинович, викладач кафедри інформаційних технологій та програмної інженерії;

Войцеховська Марія Михайлівна, д.ф., викладач кафедри інформаційних технологій та програмної інженерії

Відповідальний за випуск: Войцеховська М.М., д.ф., викладач кафедри інформаційних технологій та програмної інженерії

Рецензент: Ткач Юлія Миколаївна, д.пед.н., професор, завідувач кафедри кібербезпеки та математичного моделювання

ЗМІСТ

ВСТУП

Проект – це унікальний процес, що складається із сукупності скоординованих та керованих видів діяльності з датами початку та завершення, створений для досягнення мети, яка відповідає конкретним вимогам, і включає обмеження по термінах, вартості та обсягу ресурсів. Також це роботи, плани, заходи та інші завдання, спрямовані на створення нового продукту (пристрою, роботи, послуги).

Виконання проекту становить проектну діяльність.

Проект має низку властивих йому характеристик.

Тимчасовість – будь-який проект має чіткі часові рамки, в разі, якщо таких обмежень немає, діяльність називається операцією та може тривати як завгодно довго.

Унікальні продукти, послуги, результати – проект повинен породжувати унікальні результати, досягнення, продукти.

Послідовна розробка – будь-який проект розвивається з часом, проходячи через визначені раніше етапи або кроки, але при цьому складання специфікацій проекту строго обмежено змістом, встановленим на етапі ініціалізації.

Проектування являє собою процес, що полягає в створенні проекту, прототипу, праобразу передбачуваного або можливого об'єкта (продукту).

Проектування передбачає розробку проектної, конструкторської та іншої технічної документації, призначеної для створення інформаційної системи (ІС).

Проектування ІС охоплює три основні області:

- проектування об'єктів даних, які будуть реалізовані в базі даних;
- проектування програм, екранних форм, звітів, які будуть забезпечувати виконання запитів до даних;
- облік конкретного середовища або технології, а саме: топології мережі, конфігурації апаратних засобів, використовуваної архітектури (файл-сервер або клієнт-сервер), паралельної обробки, розподіленої обробки даних і т.п.

Вимоги до звіту

В період проходження практики кожен ЗВО повинен систематично, грамотно і акуратно вести щоденник (додаток 1). Окремо складається звіт про проходження практики. Зміст щоденника-звіту має охоплювати всю програму практики. Щоденник-звіт необхідно заповнювати кожен день, описувати всю роботу виконану практикантом, її організацію. Крім того необхідно зробити висновки і пропозиції, рекомендації на основі аналізу та узагальнення власного досвіду роботи.

Щоденник підписується начальником бази практики, завіряється печаткою (за наявності).

У щоденник має бути внесений висновок безпосереднього керівника практики.

За результатами проходження практики ЗВО пишуть індивідуальні звіти. Обсяг звіту 15-35 аркушів формату А4. Титульний лист звіту оформляється за формою (додаток А). Звіт має містити відомості про конкретну роботу, що виконана ЗВО під час практики за вказаними темами програми. Звіт повинен мати наскрізну нумерацію сторінок. Для узагальнення результатів ЗВО відводиться 3 дні в кінці практики.

Загальні відомості щодо оформлення звіту:

а) розміри полів: ліве – не менше 20-25 мм, праве – 15 мм, верхнє – 20 мм, нижнє – 20 мм;

б) нумерація сторінок звіту наскрізна;

в) номер сторінки ставиться в верхньому правому куті;

г) сторінки, зайняті таблицями та ілюстраціями, включаються в наскрізну нумерацію;

д) кожна таблиця повинна мати власний номер і тематичну назву;

е) ілюстрації нумеруються в межах розділу; під рисунком необхідно наводити підпис, що розкриває його зміст; якщо рисунок запозичений, то обов'язково вказується в квадратних дужках номер джерела за списком літератури;

є) у список літератури включають ті джерела, на які зроблені посилання в тексті, а найменування джерел розташовують у порядку появи посилань;

ж) додаток (за наявності) оформлюється як продовження звіту; у додатку розміщують лістинги, об'ємні таблиці, діаграми та інші виробничі матеріали.

Захист практики відбувається після завершення практики, але не пізніше трьох робочих днів.

Звіт перевіряється і затверджується керівником практики.

Після закінчення практики безпосередній керівник практики на кожного ЗВО-практиканта оформляє у щоденнику відгук.

Підведення підсумків практики

Після закінчення терміну практики ЗВО звітують про виконання програми та індивідуального завдання.

Щоденник-звіт разом з висновками безпосереднього керівника практики, характеристикою, індивідуальним завданням, направленням подається на рецензування керівнику практики від навчального закладу.

Щоденник з практики захищається ЗВО. Керівник практики приймає залік у ЗВО на протязі 3-х днів після закінчення практики у навчальному

закладі. Оцінка за практику вноситься в заліково-екзаменаційну відомість і в залікову книжку ЗВО за підписом керівника практики від вузу. Залік оцінюється оцінкою за шкалою оцінок, що її прийнято в НУ «Чернігівська політехніка».

Здобувачі вищої освіти, які не виконали програму практики і отримали незадовільні характеристики на базі практики, направляються на практику повторно (але вже без відриву від навчання). Якщо ж результат повторного проходження практики є теж незадовільним, ЗВО можуть бути відраховані з вищого навчального закладу.

Керівник практики інформує керівника програми та адміністрацію НУ «Чернігівська політехніка» про фактичні терміни початку та кінця практики, про склад груп, які виконали програму практики, про стан дисципліни, стан охорони праці на підприємстві та про інші питання організації та проведення практики.

Правила щодо ведення та оформлення щоденника

Щоденник є основним документом ЗВО під час проходження практики.

Для ЗВО, який проходить практику за межами міста, в якому знаходиться вуз, щоденник є також посвідченням про відрядження та тривалість перебування ЗВО на практиці.

Під час практики ЗВО щодня стисло чорнилом повинен записувати до щоденника все, що він зробив за день по виконанню календарного графіка проходження практики.

Після закінчення практики щоденник разом зі звітом повинен переглядатись керівниками практики, які складають відгук та підписують його.

Оформлений щоденник разом зі звітом ЗВО повинен здати на кафедру.

Без заповненого щоденника практика не зараховується.

ПРАКТИЧНЕ ЗАНЯТТЯ №1

ПАТЕРНИ ПРОЕКТУВАННЯ

Мета заняття: вивчити принципи побудови таких патернів проектування як одинак (singleton), декоратор, RAII, спостерігач (observer), абстрактна фабрика.

Теоретична частина

Патерни проектування – повторювана архітектурна конструкція, що представляє собою рішення певної проблеми проектування в рамках деякого часто виникаючого контексту.

Патерни поділяють на три категорії:

1) Породжуючі. Патерни, які абстрагують для програміста спосіб створення об'єктів. Наприклад, одинак, абстрактна фабрика;
2) Поведінкові. Патерни, визначають взаємодію між об'єктами. Наприклад, спостерігач;
3) Структурні. Патерни, які змінюють інтерфейс існуючих об'єктів або їх реалізацію спрощуючи розробку або реалізацію. Наприклад, декоратор.

Шаблон Одинак (Singleton)

Мета singleton шаблону полягає в гарантії того, що у класі буде тільки один екземпляр об'єкта і надання доступу до цього об'єкта. Для цього забороняється створення об'єкта через конструктор (в тому числі і конструктор копіювання), забороняється копіювання об'єкта через присвоювання, створюється спеціальний метод, який буде «віддавати» об'єкт. Приклад реалізації даного шаблону наведено нижче.

```
class CSingleton {
 CSingleton () {};
 CSingleton (CSingleton& root);
 CSingleton& operator=(CSingleton &); public:
 static CSingleton& get_instance()
  { static CSingleton instance; return instance; }
// поведінка об'єкта;
```

Шаблон Декоратор (Decorator)

Метою шаблону декоратор є реалізація схеми, при якій є можливість динамічно підключати функціональність об'єкту. Досягається це в такий спосіб: для об'єкта, поведінку якого необхідно розширити, створюється клас-спадкоємець – декоратор; до класу-спадкоємця (декоратор) додається покажчик на розширюваний об'єкт; в програмі створюється екземпляр базового класу;

створюється екземпляр класу декоратора, в конструктор якого передається «розширюваний» клас.

Розглянемо приклад такого шаблону:

```
class library_item { int num_copies;
public:
 void set_num_copies(int value) { num_copies = value; }
 int get_num_copies(void) { return num_copies; }
 virtual void display(void) = 0;
};
class book : public library_item {
 book(); string author, title; public:
 book(string author, string title, int num_copies) :
  author(author), title(title) { set_num_copies(num_copies); }
 void display(void) {
  cout << "Book ------ " << endl
     << "Author : " << author << endl
     << "Title : " << title << endl
     << "# Copies : " << get_num_copies() << endl;
 }
};
class video : public library_item {
 video(); string director, title; public:
 video(string director, string title, int num_copies) :
  director(director), title(title) { set_num_copies(num_copies); }
 void display(void) {
  cout << "Video ------ "<< endl
     << "Director : "<< director << endl
     << "Title : "<< title << endl
     << "# Copies : " << get_num_copies() << endl;
 }
};
class decorator : public library_item {
 decorator();
protected:
 library_item* li; public:
 decorator (library_item* li) : li(li) {}
 void display(void) {
  li->display(); }
  int get_num_copies(void) { return li->get_num_copies();
 } 5

};
class decorator_borrow : public decorator {
```

```
        list <string> borrowers;
       public:
        decorator_borrow(library_item* li) : decorator(li) {}
        void borrow(string name) { borrowers.push_back(name); }
        void display() {
         list <string>::iterator i;
         for(i = borrowers.begin(); i != borrowers.end(); ++i) {
          cout << *i << endl;
         }
        }
       };
       int main() {
        book *b1 = new book("Autor1","Book name1", 10);
        b1->display();
        video *v1 = new video("Autor2", "Book name2", 100);
        v1->display();
        decorator_borrow dec_b(v1);
        dec_b.borrow("Name 1");
        dec_b.borrow("Name 2");
        dec_b.display();
}
```

У прикладі використовуються дві сутності, вирощені через класи book і video і мають спільну поведінку – метод display. Загальний клас декораторів (decorator) і класи video, book мають спільного предка – клас library_item. Динамічна поведінка реалізується за допомогою класу decorator_borrow, в якому реалізована функціональність позики сутності.

Патерн RAII

Основною ідеєю принципу програмування RAII (Resource Acquisition Is Initialization – Отримання ресурсу є ініціалізація) є асоціація об'єкта класу з ресурсом для того, щоб спростити логіку роботи з виділенням і звільненням ресурсу. А саме: при створенні об'єкта класу ресурс виділяється, а при видаленні об'єкта – звільняється. Прикладом застосування патерну RAII може бути робота з файлом:

```
       class File {
        FILE* file_handler;
       public:
        File(string filename) {
         file_handler = fopen(filename.c_str(), "w");
         if ( !file_handler )
          throw runtime_error(string("Can not open file:") + filename) ;
        }
```

```
~File() {
 if ( fclose(file_handler) != 0 ) 6

  throw runtime_error("Can not close file.");
}
void write () {/* write code here */}
};
```

Патерн Спостерігач (Observer)

При створенні об'єкта такого класу «захоплюється» ресурс файлу і утримується протягом усього існування об'єкта в приватному атрибуті класу. Звільнення виконується при знищенні об'єкта. Патерн спостерігач реалізує схему видавець-передплатник, тобто реалізує механізм оповіщення всіх підписаних класів на певну подію.

Приклад реалізації патерна:

```
class Subject {
 vector < Observer* > views; int value;
public:
 void attach(Observer *obs) {
  views.push_back(obs); }
 void setVal(int val) { value = val; notify(); }
 int getVal() { return value; }
 void notify() {
 for (int i = 0; i < views.size(); i++)
  views[i]->update();
 }
};
class Observer {
 Subject *model; int denom;
public:
 Observer(Subject *mod, int div) {
  model = mod; denom = div; model->attach(this);
 }
 virtual void update() = 0;
protected:
 Subject *getSubject() {return model;}
 int getDivisor() {return denom;}
};
class DivObserver: public Observer {
public:
```

```
 DivObserver(Subject *mod, int div): Observer(mod, div){}
 void update() {
  int v = getSubject()->getVal(), d = getDivisor();
  cout << v << " div " << d << " is " << v/d << '\n';
 }
};
class ModObserver: public Observer {
public:
 ModObserver(Subject *mod, int div): Observer(mod, div){}
 void update() {
  int v = getSubject()->getVal(), d = getDivisor(); 7

  cout << v << " mod " << d << " is " << v%d << '\n';
 }
};
int main() {
 Subject subj;
 DivObserver divObs1(&subj, 4);
 DivObserver divObs2(&subj, 3);
 ModObserver modObs3(&subj, 3);
 subj.setVal(14);
}
```

Патерн Абстрактна фабрика

Патерн Абстрактна фабрика необхідний для абстрагування від створення об'єктів класів: створення сутностей виконується не з допомогою оператора new, а за допомогою спеціальних інтерфейсів. Нижче наведено приклад створення об'єктів різних класів локалізації:

```
class Locale {
protected:
 string language;
public:
 string getLanguage() {return language;}
};
class RussianLocale : public Locale {
public:
 RussianLocale() {language = string("russian");};
};
class EnglishLocale : public Locale {
public:
 EnglishLocale() {language = string("english");};
```

```
};
class Factory {
public:
 static Locale* getLocale(bool isRussian = true) {
  static vector <Locale*> ll;
  if (isRussian) {
   ll.push_back(new RussianLocale());
  } else {
   ll.push_back(new EnglishLocale());
  }
  return ll.back();
 }
};
```

Хід роботи

1) Реалізувати функції програми з використанням відповідних патернів відповідно до варіанту.
2) Звіт повинен містити вихідний код програми і приклади запуску програми, що ілюструє використовуваний патерн.

Варіанти завдань

1) Використовуючи патерн «одинак», реалізувати клас логування повідомлень програми. В лог має виводитися: тип запису (INFO, DEBUG, WARNING, ERROR), час запису, клас і рядок, з якого викликаний метод логування. Показати, що створюється тільки один об'єкт логування.
2) Створити клас «автомобіль». Використовуючи патерн «декоратор» динамічно додати/видалити додаткову функціональність «покупки/продажу».
3) Використовуючи патерн RAII реалізувати власний клас, в якому реалізуються функції (виділення ресурсу, звільнення ресурсу, доступ до ресурсу) розумного покажчика «unique_ptr».
4) Створити класи «кафе» і «відвідувач». Використовуючи патерн «спостерігач» реалізувати підписку кількох «відвідувачів» на події в «кафе». Показати, що всі підписані на події «Відвідувачі» отримують повідомлення.
5) Використовуючи патерн «абстрактна фабрика» реалізувати створення класів згідно з переданими до фабрики параметрам – назва класу і версія.
6) Використовуючи патерн «одинак» реалізувати менеджер управління пам'яттю. Менеджер повинен мати методи для виділення заданого обсягу пам'яті під змінні, звільнення пам'яті під змінну. Показати, що створюється тільки один об'єкт менеджера.

7) Використовуючи патерн «декоратор» динамічно додати/видалити додаткову функціональність «додавання/видалення користувачів» для сутності «сайт».
8) Використовуючи патерн RAII реалізувати свій клас, в якому реалізуються функції (виділення ресурсу, звільнення ресурсу, доступ до ресурсу) розумного покажчика «shared_ptr».

Приклад

Шаблон «спостерігач» застосовується в тих випадках, коли система має такі властивості:

- існує, щонайменше, один об'єкт, що розсилає повідомлення;
- є не менше одного одержувача повідомлень, причому їх кількість і склад можуть змінюватися під час роботи програми;
- немає потреби дуже сильно зв'язувати взаємодіючі об'єкти, що корисно для повторного використання.

Даний шаблон часто застосовують в ситуаціях, в яких відправника повідомлень не цікавить, що роблять одержувачі з наданою їм інформацією.

При реалізації шаблону «спостерігач» зазвичай використовуються наступні класи:

Observable – інтерфейс, який визначає методи для додавання, видалення та оповіщення спостерігачів;

Observer – інтерфейс, за допомогою якого спостерігач отримує сповіщення;

ConcreteObservable – конкретний клас, який реалізує інтерфейс Observable;

ConcreteObserver – конкретний клас, який реалізує інтерфейс Observer.

Опис завдання: Метеостанція відстежує погодні умови. Додаток має відображати 3 візуальних елемента: поточну погоду (температуру, вологість, тиск), накопичену статистику по погоді і передбачення погоди. Всі дані оновлюються в реальному часі, по мірі того, як метеостанція отримує дані з датчиків. Також необхідно передбачити можливість в майбутньому додавати 4, 5 і т.п. візуальні елементи.

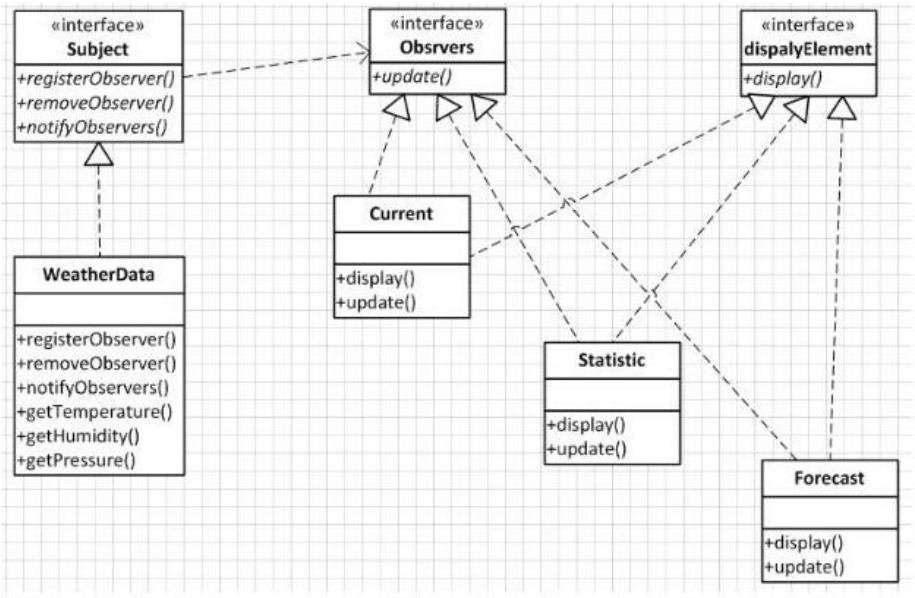

Рисунок 1.1 – Діаграма класів

1. Створюємо новий порожній проект в Visual Studio і вибираємо відповідні налаштування (рисунок 1.2).

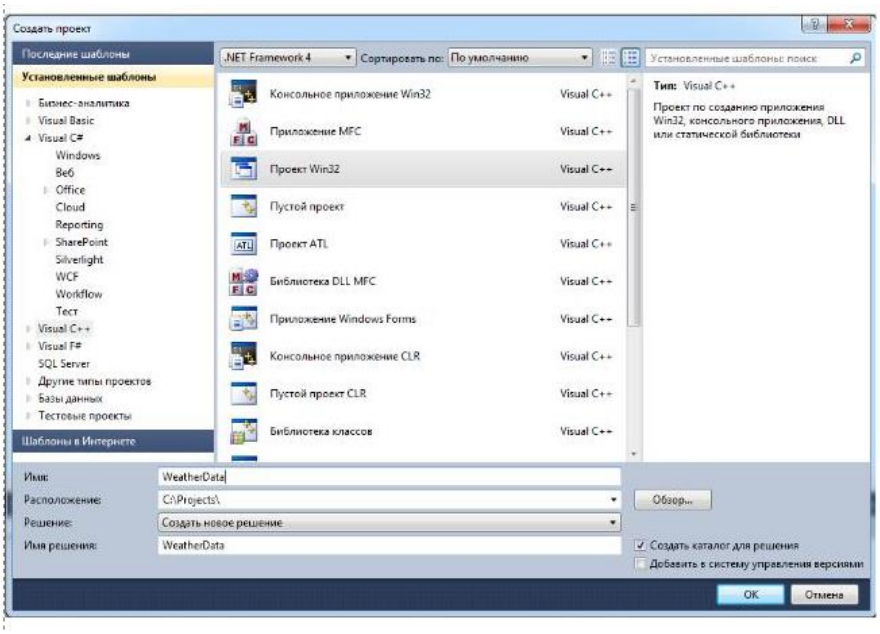

Рисунок 1.2 – Вікно створення нового проекту у Visual Studio

2. Створюємо інтерфейс Observers, від якого потім будемо наслідувати всіх Спостерігачів. У інтерфейса є тільки заголовки. Метод update() є чистим віртуальним, для реалізації динамічного поліморфізму. Метод update() призначений для поновлення даних у Спостерігачів.

```
#pragma once
class Observer
{
```

```
public:
virtual void update(float, float, float)=0;
};
```

3. Створюємо інтерфейс DisplayElement, від якого потім будемо наслідувати всіх Спостерігачів. У інтерфейса є тільки заголовки. Метод display() є чистим віртуальним, для реалізації динамічного поліморфізму. Метод display() призначений для відображення даних у Спостерігачів. Зверніть увагу, що в даному випадку буде реалізовуватися множинне успадкування для Спостерігачів.

```
#pragma once
class DisplayElement
{
virtual void display()=0;
};
```

4. Реалізуємо Спостерігача, який, наприклад, містить і відображає поточні погодні умови. Зверніть увагу на спадкування від двох інтерфейсів. У класі Спостерігача знаходиться реалізація методів update() і display(). Заводяться змінні, що описують значення вологості, температури і тиску.

Заголовки:

```
#pragma once
#include "Observer.h"#include"DisplayElement.h"classCurrentDisplay :
public Observer,public DisplayElement
{
float temperature, humidity, pressure;
public: CurrentDisplay(void); ~CurrentDisplay(void);
void update(float temperature,float humidity,float pressure);void display();
};
```

Файл реалізації:

```
#include<iostream>using namespace std;
#include "CurrentDisplay.h"
CurrentDisplay::CurrentDisplay(void)
{
temperature=0;
humidity=0;
pressure=0;
```

```
}
CurrentDisplay::~CurrentDisplay(void)
{
}
void CurrentDisplay::update(float temperature,float humidity,float pressure)
{
this->temperature=temperature;this->pressure=pressure;this-
>humidity=humidity;
}
void CurrentDisplay::display()
{
cout << "Current Temperature " << temperature << endl; cout <<"Current
Humidity " << humidity << endl;
cout << "Current Pressure " << pressure << endl;
}
```

5. Реалізуємо Спостерігача, що відображає статистичну інформацію за останніми 10 значеннями.

Заголовки:

```
#pragma once #include <iostream>
#include "observer.h"#include"DisplayElement.h"classStatisticsDisplay :
public Observer,public DisplayElement
{float average_temperature, average_humidity,
average_pressure;floatarray_temperature[10];
float array_humidity[10];floatarray_pressure[10];public:
StatisticsDisplay(void); ~StatisticsDisplay(void);
void update(float temperature,float humidity,float pressure);void display();
};
```

Файл реалізації:

```
#include "StatisticsDisplay.h"using namespace std;
StatisticsDisplay::StatisticsDisplay(void)
{
for(int i=0; i<10; i++)
{
array_temperature[i]=0; array_humidity[i]=0;
array_pressure[i]=0;
}
}
StatisticsDisplay::~StatisticsDisplay(void)
```

```
    {
    }
    void StatisticsDisplay::display()
    {
    cout << "Average Temperature: " << average_temperature << endl; cout
<<"Average Humidity: " << average_humidity << endl;
    cout << "Average Pressure: " << average_pressure << endl;
    }
    void StatisticsDisplay::update(float temperature,float humidity,float pressure)
    {
    for(int j=8;j>=0;j--)
    {
    array_temperature[j+1]=array_temperature[j];
array_humidity[j+1]=array_humidity[j]; array_pressure[j+1]=array_pressure[j];
    }
    array_temperature[0]=temperature; array_humidity[0]=humidity;
array_pressure[0]=pressure;
    float sum_temperature = 0;floatsum_humidity = 0;
    float sum_pressure = 0;for(inti=0;i<10;i++)
    {
    sum_temperature += array_temperature[i]; sum_humidity +=
array_humidity[i];
    sum_pressure += array_pressure[i];
    }
    average_temperature = sum_temperature / 10; average_humidity =
sum_humidity / 10; average_pressure = sum_pressure / 10;
    }
```

6. Реалізуємо Спостерігача, який здійснює прогнози. Прогноз для спрощення завдання робимо наступним чином: для прогнозу на завтра до поточних значень додаємо одиницю.

Заголовки:

```
    #pragma once #include <iostream>
    #include "observer.h"#include"DisplayElement.h"classForecastDisplay :
    public Observer,public DisplayElement
    {float forecast_temperature, forecast_humidity, forecast_pressure;public:
    ForecastDisplay(void);
    ~ForecastDisplay(void);
    void update(float temperature,float humidity,float pressure);void display();
    };
```

Файл реалізації:

```
#include"ForecastDisplay.h"using namespace std;

ForecastDisplay::ForecastDisplay(void)
{
forecast_temperature=0; forecast_humidity=0; forecast_pressure=0;
}


ForecastDisplay::~ForecastDisplay(void)
{
}

void ForecastDisplay::display()
{
cout << "Forecast Temperature " << forecast_temperature << endl; cout <<"Forecast Humidity " <<
forecast_humidity << endl;
cout << "Forecast Pressure " << forecast_pressure << endl;
}

void ForecastDisplay::update (float temperature,float humidity,float pressure)
{

forecast_temperature=temperature+1; forecast_humidity=humidity+1; forecast_pressure=pressure+1;

}
```

7. Реалізуємо інтерфейс Subject, який буде зв'язуватися з інтерфейсом Observers і до завдань якого входить підписка, відписка і повідомлення Спостерігачів. В інтерфейсі Subject всі методи є чистими віртуальними. Реалізація цих методів буде зроблена в класі WeatherData.

```
#pragma once
#include "Observer.h"

class Subject
{
virtual void registerObserver(Observer*)=0;virtual voidremoveObserver(Observer*)=0;virtual
voidnotifyObserver()=0;

};
```

8. Реалізація класу WeatherData. У класі WeatherData ми будемо отримувати з датчиків значення температури, вологості і тиску і встановлювати ці значення (методи getMeasurements isetMeasurements). Також в цьому класі буде реалізація методів підписки, відписки та повідомлення Спостерігачів.

Заголовки:

```
#pragma once #include"subject.h"
#include"Observer.h"classWeatherData :
public Subject
```

```
{
float temperature, humidity, pressure; Observer** observers;

public: WeatherData(void); ~WeatherData(void);

void registerObserver(Observer* a);voidremoveObserver(Observer* a);voidnotifyObserver();

void setMeasurements(float temperature,float humidity,float pressure);voidgetMeasurements();

};
```

Файл реалізації:

```
#include<iostream>using namespace std;

#include "WeatherData.h"


WeatherData::WeatherData(void)
{
this->observers=new Observer*[3];for(int i=0; i<3; i++)
{
this->observers[i]=0;
}

}

WeatherData::~WeatherData(void)
{

delete []observers;
}

void WeatherData::registerObserver(Observer *a)
{
for (int i=0; i<3; i++)
{
if(observers[i] == 0)
{
observers[i]=a;return;
}

}
cout << "No positions for Observer "<< endl;
}
void WeatherData::removeObserver(Observer* a)
{
for (int i=0; i<3; i++)
{
if(observers[i] == a)
{
observers[i]=0;return;
}

}
cout << "Observer is not in the list "<< endl;
}

void WeatherData::notifyObserver()
```

```
{
for (int i=0; i<3; i++)
{
if(observers[i] !=0)
{
observers[i]->update(this->temperature,this->humidity,this->pressure);

}

}
}

void WeatherData::setMeasurements(float temperature,float humidity,float pressure)
{
this->temperature=temperature;this->humidity=humidity;this->pressure=pressure;this->notifyObserver();
}
void WeatherData::getMeasurements()
{
float temperature, humidity, pressure; cout <<"T= ";
cin >>temperature; cout << "H= "; cin >>humidity; cout <<"P= "; cin >>pressure;
this->setMeasurements(temperature,humidity, pressure);

}
```

9. Створюємо реалізацію, тобто функцію main(). Підключаємо всі необхідні класи і інтерфейси. Заводимо змінні з типами спостерігачів і основного класу WeatherData. Реєструємо всіх Спостерігачів. Отримуємо значення з датчиків і відображаємо значення, отримані нашими Спостерігачами.

```
#include <iostream> using namespacestd;#include"WeatherData.h"
#include "CurrentDisplay.h"#include"ForecastDisplay.h"#include"StatisticsDisplay.h"#include<conio.h>


void main ()
{
WeatherData a; ForecastDisplay b;
StatisticsDisplay c;
CurrentDisplay d;

a.registerObserver(&b);
a.registerObserver(&c);
a.registerObserver(&d);

while (1)
{

a.getMeasurements();
b.display();
c.display();
d.display();
}

getch();

}
```

На початку програми запитуємо у користувача значення температури, вологості і тиску. Як результат отримуємо дані з усіх Спостерігачів.

Контрольні питання

1) Які переваги дає використання патерну Спостерігач?
2) Вкажіть на діаграмі класів де використовується механізм успадкування.
3) Що таке інтерфейс, чим він відрізняється від абстрактного класу?
4) Вкажіть на діаграмі класів де використовується механізм композиції? Які переваги в порівнянні з успадкуванням він дає?
5) Вкажіть в коді програми місце використання механізму композиції.
6) Що необхідно буде змінити в коді і на діаграмі класів на випадок додавання ще одного Спостерігача?
7) Чи обов'язкова наявність інтерфейсу Observers і чи можна обійтися без нього? До чого це призведе в разі розширення коду?

Перелік літератури

1. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования – 2021.
2. Влиссидес Д. Применение шаблонов проектирования. Дополнительные штрихи – 2003.
3. Ларман К. - Применение UML и шаблонов проектирования. – 2004г.
4. Фримен Эр., Фримен Эл., Сьерра К., Бейтс Б. - Паттерны проектирования – 2011.
5. Андрій Будай. Дизайн-патерни – просто, як двері. 2012. https://learn.ztu.edu.ua/pluginfile.php/632/mod_resource/content/1/DesignPatterns_AndriyBuday.pdf
6. Grady Booch et. al. Object Oriented Analysis And Design With Applications 3Rd Edition. The Addison-Wesley Object Technology Series, 2009. https://www.auhd.site/upfiles/elibrary/Azal2020-01-22-01-00-32-27635.pdf

ПРАКТИЧНЕ ЗАНЯТТЯ №2
КЕРУВАННЯ РИЗИКАМИ. ФУНКЦІОНАЛЬНІ ТА НЕФУНКЦІОНАЛЬНІ ВИМОГИ

Мета заняття: ознайомитися з можливими ризиками програмних проектів. Навчитися визначати та аналізувати ризики. Ознайомитися з функціональними та нефункціональними вимогами. Навчитися розробляти функціональні та нефункціональні вимоги.

Теоретична частина

Менеджер проекту

Важливою частиною роботи менеджера проекту є оцінка ризиків, які можуть вплинути на графік робіт або на якість створюваного програмного *продукту,* і розробка заходів щодо запобігання ризиків. Результати аналізу ризиків повинні бути відображені в плані проекту. Визначення ризиків і розробка заходів щодо зменшення їхнього впливу на хід виконання проекту називається *керуванням ризиками.* Спрощено ризик можна розуміти як імовірність прояву будь-яких несприятливих обставин, що негативно впливають на реалізацію проекту. Ризики можуть погрожувати проекту в цілому, створюваному програмному продукту або організації-розробникові. Можна виділити три типи ризиків.

1) *Ризики для проекту,* які впливають на графік робіт або ресурси, необхідні для виконання проекту.
2) *Ризики для розроблюваного продукту,* що впливають на якість або продуктивність розроблювального програмного продукту.
3) *Бізнес-Ризики,* що відносяться до організації-розробника або постачальника.

Звичайно, ці типи ризиків можуть перетинатися. Наприклад, якщо досвідчений програміст залишає проект, це буде ризиком для проекту (оскільки затримується строк здачі готового продукту), ризиком для продукту (тому що новий програміст, що прийде на заміну, може виявитися не досить досвідченим і припуститися помилок в програмі) і бізнес-ризиком (оскільки затримка даного проекту може негативно вплинути на майбутні ділові контакти між замовником і організацією-розробником).

Типи ризиків

Конкретні типи ризиків, які можуть вплинути на даний проект, залежать від виду створюваного програмного продукту та від організаційного оточення, де реалізується програмний проект. Разом з тим багато типів ризиків здатні вплинути на будь-які програмні проекти, ці ризики наведені в таблиці 2.1.

Таблиця 2.1 – Можливі ризики програмних проектів

Ризик	Тип ризику	Опис ризику
Плинність розробників	Ризик для проекту	Досвідчені розробники залишають проект до його завершення
Зміна в керуванні організацією	Ризик для проекту	Організація змінює свої пріоритети в керуванні проектом
Неготовність апаратних засобів	Ризик для проекту	Апаратні засоби, які необхідні для проекту, не надійшли вчасно або не готові до експлуатації
Зміна вимог	Ризик для проекту та для розроблюваного продукту	Поява великої кількості непередбачених змін у вимогах, пропонованих до розроблюваного ПЗ
Затримка в розробці специфікації	Ризик для проекту та для розроблюваного продукту	Специфікації основних інтерфейсів підсистем не надійшли до розробників відповідно до графіка робіт
Недооцінювання розміру розроблюваної системи	Ризик для проекту та для розроблюваного продукту	Розмір системи значно перевищив первісну оцінку
Недостатня ефективність CASE-засобів	Ризик для проекту та для розроблюваного продукту	CASE-засоби, призначені для підтримки проекту, виявилися менш ефективними, ніж очікувалося
Зміни в технології розробки ПЗ	Ризик для розроблюваного продукту	Основні технології побудови програмної системи заміняються новими
Поява конкуруючого програмного продукту	Бізнес-ризик	На ринку програмних продуктів до закінчення проекту зёявилася конкуруюча програмна система

Стадії процесу управління ризиками

Визначення ризиків. Визначаються можливі ризики для проекту, для розроблюваного продукту та бізнес-ризики.

Аналіз ризиків. Оцінюється ймовірність та послідовність появи ризикових ситуацій.

Планування ризиків. Плануються заходи щодо запобігання ризиків або мінімізації їхнього впливу на проект.

Моніторинг ризиків. Постійне оцінювання ймовірностей ризиків і виконання заходів щодо пом'якшення наслідків прояву ризикової ситуацій.

Процес керування ризиками, як і інші процеси планування, є ітераційним, виконуваним протягом усього строку реалізації проекту. Спочатку розробляються плани керування ризиками, потім постійно відслідковується ситуація навколо реалізації проекту. При надходженні нової інформації про можливі ризики заново проводиться аналіз ризиків і першочергова увага приділяється новим ризикам. По мірі надходження нової інформації також змінюються плани заходів щодо запобігання та пом'якшення ризиків.

Результати процесу керування ризиками документуються у вигляді планів керування ризиками. Вони повинні включати опис можливих проектних ризиків, їхній аналіз і перелік заходів, необхідних для керування ризиками.

Визначення ризиків

Визначення ризиків – перша стадія процесу керування ризиками. На цій стадії описуються ризики, які можуть виявитися при реалізації проекту. У принципі на цій стадії не повинна оцінюватися ймовірність і значимість ризиків, але на практиці малоймовірні ризики з незначними наслідками зазвичай відкидаються відразу.

Визначення ризиків може виконуватися в режимі командної роботи з використанням підходу "мозковий штурм" або ґрунтуватися на досвіді менеджера. При визначенні ризиків може допомогти наведений нижче список можливих категорій ризиків.

1. *Технологічні ризики.* Виникають із програмних і апаратних технологій, на основі яких розробляється система.

2. *Ризики, пов'язані з персоналом.* Пов'язані зі членами команди розроблювачів.

3. *Організаційні ризики.* Виникають із організаційного оточення, у якому виконується проект.

4. *Інструментальні ризики.* Пов'язані з використовуваними CASE-засобами й іншими засобами підтримки процесу створення ПЗ.

5. *Ризики, пов'язані із системними вимогами.* Проявляються при зміні вимог, пред'явлених до розроблюваної системи.

6. *Ризики оцінювання.* Зв'язані з оцінюванням характеристик програмної системи та ресурсів, необхідних для реалізації проекту.

У таблиці 2.2 представлені деякі приклади, що відносяться до кожної з описаних категорій ризиків. Результатом етапу визначення ризиків буде довгий перелік можливих ризиків, які можуть вплинути на розроблювальний програмний продукт, проект або організацію-розробника.

Таблиця 2.2 – Категорії ризиків

Категорія ризиків	Приклади ризиків
Технологічні ризики	База даних, що використовується в програмній системі, не забезпечує обробку очікуваного обсягу транзакцій. Програмні компоненти, які використовуються повторно, мають дефекти, що обмежують їхні функціональні можливості.
Ризики, пов'язані з персоналом	Неможливо підібрати працівників з необхідним професійним рівнем. Провідний розробник занедужав у самий критичний час. Неможливо організувати необхідне навчання персоналу.
Організаційні ризики	В організації, що виконує розробку ПЗ, відбулася реорганізація, в результаті чого змінилися пріоритети в керуванні проектом. Фінансові труднощі в організації привели до зменшення бюджету проекту. Недооцінення часу виконання проекту.
Інструментальні ризики	Програмний код, що генерується CASE-засобами, не ефективний. CASE-засоби неможливо інтегрувати з іншими засобами підтримки проекту.
Ризики, пов'язані із системними вимогами	Зміни вимог приводять до значних повторних робіт з проектування системи. Первинне нечітке формулювання користувальницьких вимог привело до значних змін системних вимог, що виявилися на пізніх стадіях розробки проекту.

Модифіковані арифметичні типи

При аналізі для кожного певного ризику обчислюється ймовірність його прояву та збиток, що він може нанести. Не існує простих методів виконання аналізу ризиків – значною мірою він заснований на думці та досвіді менеджера. Не претендуючи на виняткову точність, можна привести наступну шкалу ймовірностей ризиків та їхніх наслідків.

Імовірність ризику вважається дуже низькою, якщо вона має значення менше 10%; низькою, якщо її значення від 10 до 25%; середньою при значеннях від 25 до 50%; високою якщо значення коливається від 50 до 75%; дуже високою при значеннях більше 75%.

Можливий збиток від ризиків можна розділити на катастрофічний, серйозний, прийнятний та незначний.

Результати аналізу ризиків повинні бути представлені у вигляді таблиці ризиків, упорядкованих за ступенем можливого збитку. У табл. 2.3 наведений упорядкований список ризиків, приведених у таблиці 2.2; там же зазначені ймовірності цих ризиків. Тут імовірності ризиків і ступінь збитку від них зазначені довільно. На практиці для їхнього визначення необхідна докладна інформація про проект, технологію створення ПО, команду розробників та про саму організацію.

Таблиця 2.3 – Перелік ризиків після проведення аналізу

Ризик	Ймовірність	Ступінь збитку
Фінансові труднощі в організації привели до зменшення бюджету проекту	Низька	Катастрофічний
Неможливо підібрати працівників з необхідним професійним рівнем	Висока	Катастрофічний
Провідний розробник занедужав у самий критичний час	Середня	Серйозний
Програмні компоненти, які використовуються повторно, мають дефекти, що обмежують їхні функціональні можливості	Середня	Серйозний
Зміни вимог приводять до значних повторних робіт із проектування системи	Середня	Серйозний
В організації, що виконує розробку ПЗ, відбулася реорганізація, у результаті чого змінилися пріоритети в керуванні проектом	Висока	Серйозний
База даних, що використовується в програмній системі, не забезпечує обробку очікуваного обсягу транзакцій	Середня	Серйозний
Недооцінення часу виконання проекту	Висока	Серйозний
CASE-засоби неможливо інтегрувати з іншими засобами підтримки проекту	Висока	Прийнятний
Первинне нечітке формулювання користувальницьких вимог привело до значних змін системних вимог, що виявилися на пізніх стадіях розробки проекту	Середня	Прийнятний
Неможливо організувати необхідне навчання персоналу	Середня	Прийнятний
Швидкість виявлення дефектів у системі нижче раніше запланованої	Середня	Прийнятний
Розмір системи значно перевищує спочатку розрахований	Висока	Прийнятний
Програмний код, що згенерований CASE-засобами, неефективний	Середня	Незначний

Звичайно, як імовірність ризиків, так і можливий збиток від них повинні переглядатися при надходженні додаткової інформації про ці ризики та по мірі реалізації заходів щодо керування ними. Тому подібні таблиці ризиків повинні перероблятися на кожній ітерації процесу керування ризиками.

Планування ризиків

Після проведення аналізу ризиків визначаються найбільш значимі ризики, які потім відслідковуються протягом усього терміну виконання проекту. Визначення цих значимих ризиків залежить від їхніх імовірностей і можливого збитку. У загальному випадку завжди відслідковуються ризики з катастрофічними наслідками, а також ризики із серйозним збитком, значення ймовірності яких перевищує середнє.

Кількість ризиків, які необхідно відслідковувати, залежить від конкретного проекту. Це може бути п’ять ризиків, а може – п’ятнадцять. Але, звісно, кількість ризиків, по яких проводиться моніторинг, повинне бути доступним для огляду. Велика кількість ризиків, що відслідковуються, зажадає величезної кількості збираємої інформації. Зі списку ризиків, представлених у таблиці 2.3, для моніторингу варто відібрати всі вісім ризиків, які можуть привести до катастрофічних і серйозних наслідків.

Види вимог

Вимоги до програмної системи часто класифікуються як функціональні, нефункціональні та вимоги предметної області.

1. *Функціональні вимоги.* Це перелік сервісів, які повинна виконувати система, причому повинне бути зазначене, як система реагує на ті або інші вхідні дані, як вона поводиться в певних ситуаціях і т.д. У деяких випадках вказується, що система не повинна робити.

2. *Нефункціональні вимоги.* Описують характеристики системи і її оточення, а не поводження системи. Тут також може бути наведений перелік обмежень, що накладаються на дії та функції, виконувані системою. Вони включають тимчасові обмеження, обмеження на процес розробки системи, стандарти й т.д.

3. *Вимоги предметної області.* Характеризують ту предметну область, де буде експлуатуватися система. Ці вимоги можуть бути функціональними й нефункціональними.

У дійсності чіткої границі між цими типами вимог не існує. Наприклад, користувальницькі вимоги, що стосуються безпеки системи, можна віднести до нефункціональних. Однак при більш детальному розгляді таку вимогу можна віднести до функціональних, оскільки вона породжує необхідність включення до системи засобу авторизації користувача. Тому, розглядаючи далі ці види вимог, ми повинні завжди пам'ятати, що дана класифікація в значній мірі штучна.

Функціональні вимоги

Ці вимоги описують поводження системи та сервіси (функції), які вона виконує, і залежать від типу розроблювальної системи та від потреб користувачів. Якщо функціональні вимоги оформлені як користувальницькі, вони, як правило, описують системи в узагальненому виді. На противагу цьому функціональні вимоги, оформлені як системні, описують систему максимально докладно, включаючи її вхідні та вихідні дані, виключення й т.д.

Функціональні вимоги для програмних систем можуть бути описані різними способами. Розглянемо для прикладу функціональні вимоги до бібліотечної системи університету, призначеної для замовлення книг і документів з інших бібліотек.

1. Користувач повинен мати можливість проводити пошук необхідних йому книг і документів або по всій множині доступних каталожних баз даних або по певній їхній підмножині.

2. Система повинна надавати користувачеві відповідний засіб перегляду бібліотечних документів.

3. Кожне замовлення має містити унікальний ідентифікатор (ORDER_ID), що копіюється у формуляр користувача для постійного зберігання.

Ці функціональні користувальницькі вимоги визначають властивості, якими повинна володіти система. Вони взяті з документа, що містить користувальницькі вимоги, і показують, що функціональні вимоги можуть бути описані з різним рівнем деталізації (порівняйте першу та третю вимоги).

Багато проблем, що виникають при розробці систем, пов'язані з неточністю та "розмитістю" специфікації вимог. Природньо, що розробники інтерпретують вимоги, що допускають двояке тлумачення, так, щоб систему було простіше реалізувати. Але це тлумачення може не збігатися з очікуваннями замовника. Така ситуація приводить до розробки нових вимог і внесення змін у систему. Це, у свою чергу, веде до затримки здачі готової системи та її подорожчанню.

Розглянемо другу вимогу до бібліотечної системи з наведеного вище списку та звернемо увагу на вираз " відповідний засіб перегляду документів". Бібліотечна система може надавати документи в широкому спектрі форматів. У вимозі мається на увазі, що система повинна надати засоби для перегляду документів у будь-якому форматі. Але оскільки ця умова чітко не виписана, розробники у випадку дефіциту часу можуть використовувати простий засіб для перегляду текстових документів і наполягати на тому, що саме таке рішення потрібно за даною вимогою.

В принципі специфікація функціональних вимог повинна бути комплексною та несуперечливою. Комплексність має на увазі опис (визначення) всіх системних сервісів. Несуперечність означає відсутність несумісних і взаємовиключних визначень сервісів. На практиці для великих і складних систем вкрай важко розробити комплексну та несуперечливу специфікацію функціональних вимог. Причина криється частково в складності самої розроблювальної системи, а частково – в неузгоджених опорних точках зору на те, що повинна робити система. Ця неузгодженість може не виявитися

на етапі первісного формулювання вимог – для її виявлення необхідний більш глибокий аналіз специфікації. Коли неузгодженість системних функцій виявиться на якому-небудь етапі життєвого циклу програми, до системної специфікації доведеться зробити відповідні зміни.

Нефункціональні вимоги

Нефункціональні вимоги не пов’язані безпосередньо з функціями, виконуваними системою. Вони пов'язані з такими інтеграційними властивостями системи, як надійність, час відповіді або розмір системи. Крім того, нефункціональні вимоги можуть визначати обмеження на систему, наприклад на пропускну здатність пристроїв введення-виведення, або формати даних, використовуваних у системному інтерфейсі.

Багато нефункціональних вимог відносяться до системи в цілому, а не до окремих її засобів. Це означає, що вони більш значимі та критичні, ніж окремі функціональні вимоги. Помилка, допущена у функціональній вимозі, може знизити якість системи, помилка в нефункціональних вимогах може зробити систему непрацездатною.

Разом з тим нефункціональні вимоги можуть висуватися не тільки до самої програмної системи: одні можуть висуватися до технологічного процесу створення ПЗ, інші – містити перелік стандартів якості, що накладаються на процес розробки. Крім того, у специфікації нефункціональних вимог може бути зазначено, що проектування системи повинне виконуватися тільки певними CASE-засобами, і наведено опис процесу проектування, якому необхідно слідувати.

Нефункціональні вимоги відображають користувальницькі потреби; при цьому вони ґрунтуються на бюджетних обмеженнях, враховують організаційні можливості компанії-розробника та можливість взаємодії розроблювальної системи з іншими програмними та обчислювальними системами, а також такі зовнішні фактори, як правила техніки безпеки, законодавство про захист інтелектуальної власності й т.п. На рисунку 2.1 показана класифікація нефункціональних вимог.

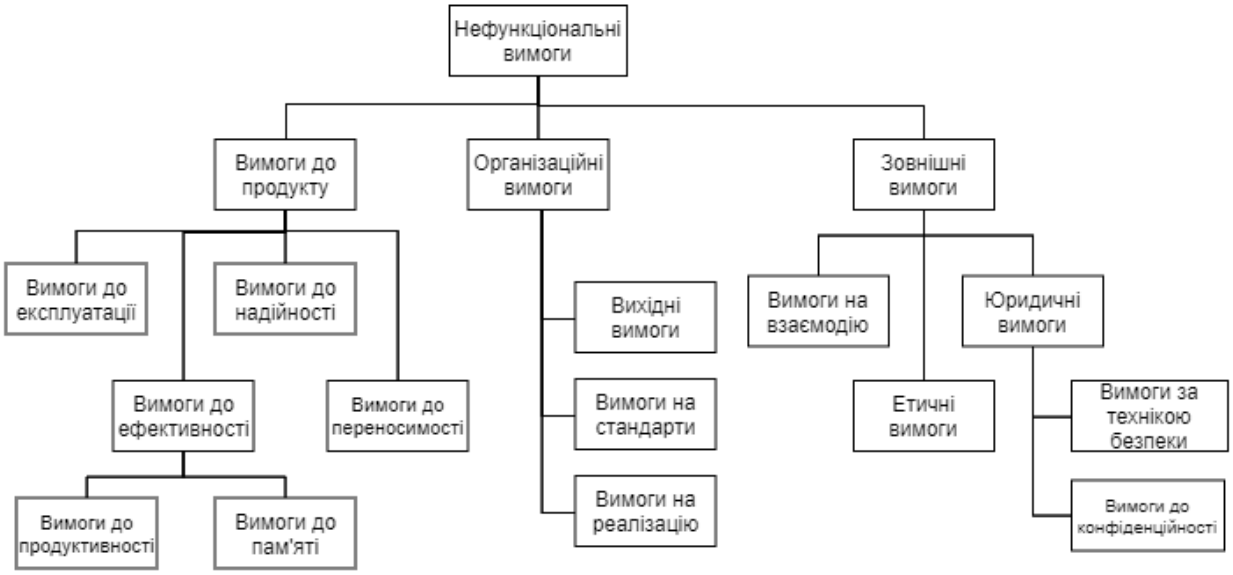

Рисунок 2.1 – Типи нефункціональних вимог

Всі нефункціональні вимоги, приведені на рисунку 2.1, можна розбити на три великі групи.

1. *Вимоги до продукту.* Описують експлуатаційні властивості програмного продукту. Сюди відносяться вимоги до продуктивності системи, обсягу необхідної пам'яті, надійності (визначає частоту можливих збоїв у системі), переносимості системи на різні комп'ютерні платформи та зручності експлуатації.

2. *Організаційні вимоги.* Відображають політику та організаційні процедури замовника та розробника ПЗ. Вони включають стандарти розробки програмного продукту, вимоги до реалізації ПЗ (тобто до мови програмування та методів проектування), вихідні вимоги, які визначають строки виготовлення програмного продукту, і супровідну документацію.

3. *Зовнішні вимоги.* Враховують фактори, зовнішні по відношенню до розроблювальної системи та процесу її розробки. Вони включають вимоги, що визначають взаємодію даної системи з іншими системами, юридичні вимоги, відповідність яким гарантує, що система буде розроблятися та функціонувати в рамках існуючого законодавства, а також етичні вимоги. Останні повинні гарантувати, що система буде прийнятна для користувачів або замовника.

Основна проблема нефункціональних вимог полягає в тому, що їхнє виконання важко перевірити. Часто вони пишуться для того, щоб відобразити загальні цілі замовника системи, такі, як простота експлуатації, можливість відновлення після збоїв або швидка відповідь на запити користувача. Реалізація подібних вимог може виявитися складною для системних розробників, оскільки вони нечітко сформульовані та відкривають простір для різних тлумачень.

Системні цілі і перевірка вимог

Системна мета

Система повинна бути простою в експлуатації для досвідченого оператора та зводити кількість його помилок до мінімуму.

Нефункціональна вимога, що перевіряється

Досвідченому операторові повинні бути доступні всі системні функції після двох годин навчання роботи з даною системою. Після такого навчання середня кількість помилок оператора не повинна перевищувати двох за робочий день.

В ідеалі нефункціональні вимоги повинні виражатися через кількісні показники, які можна об'єктивно виміряти. У таблиці 2.4 наведені показники, за допомогою яких можна специфікувати нефункціональні системні властивості.

На практиці виразити нефункціональні вимоги за допомогою кількісних показників досить важко. Часто замовник ПО не може оформити своє бачення майбутньої системи за допомогою вимог, виражених кількісними показниками. Або деякі системні вимоги, наприклад зручність супроводу, взагалі не можна виразити через кількісні показники. Крім того, витрати на об'єктивний вимір кількісних нефункціональних вимог можуть виявитися вкрай високими. Тому часто документ, що специфікує вимоги до системи, містить опис системних цілей разом із чітко сформульованими вимогами. Ці системні цілі корисні, оскільки відбивають бачення (і пріоритети) замовника про майбутню систему. Разом з тим замовник повинен розуміти, що його системні цілі можуть трактуватися різними способами і їх неможливо об'єктивно проконтролювати.

Таблиця 2.4 – Кількісні показники для нефункціональних вимог

Показник	Одиниці виміру
Швидкість	Кількість виконаних транзакцій в секунду; час реакції на дії користувача; час відновлення екрану
Розмір	Кілобайти; кількість модулів пам’яті
Простота експлуатації	Час навчання персоналу; кількість статей у довідковій системі
Надійність	Середня тривалість часу між двома послідовними проявами помилок у системі; імовірність виходу системи з ладу; коефіцієнт готовності системи
Стійкість до збоїв	Час відновлення системи після збою; відсоток подій, що призводять до збоїв; імовірність псування даних при збоях
Переносимість	Відсоток машинно-залежних операторів; кількість машинно-залежних підсистем

Функціональні та нефункціональні вимоги в документі, що описує вимоги до системи, повинні бути рознесені по різних розділах. Але на практиці цю умову виконати непросто. Якщо нефункціональні вимоги помістити окремо від функціональних, буде важко простежити взаємозв'язки між ними. Якщо всі вимоги зібрані в одному списку, складно провести аналіз функціональних і нефункціональних вимог окремо та визначити вимоги, що висуваються до системи в цілому. Вид подання вимог в одному документі також істотно

залежить від типу системи, що специфікується. Але в кожному разі повинні бути виділені вимоги, що описують інтеграційні властивості системи. Для цього їх можна помістити в окремий розділ або який-небудь інший спосіб відокремити від інших вимог.

Хід роботи

1) Визначити функціональні, нефункціональні вимоги та ризики системи відповідно до власного варіанту.
2) Оформити звіт (функціональні вимоги, нефункціональні вимоги, ризики, висновки).

Варіанти завдань

2) Інформаційна система ВНЗ
3) Інформаційна система торгової організації
4) Інформаційна система медичних організацій міста
5) Інформаційна система автопідприємства міста
6) Інформаційна система проектної організації
7) Інформаційна система авіабудівельного підприємства
8) Інформаційна система військового округу
9) Інформаційна система будівельної організації
10) Інформаційна система бібліотечного фонду міста
11) Інформаційна система спортивних організацій міста
12) Інформаційна система автомобілебудівельного підприємства
13) Інформаційна система готельного підприємства
14) Інформаційна система магазину автозапчастин
15) Інформаційна система аптеки
16) Інформаційна система туристичного клубу.

Перелік літератури

1. Eric J. Braude, Michael E. Bernstein. Software Engineering: Modern Approaches. Second Edition. Waveland Press, Inc., 2016, 782 p.

ПРАКТИЧНЕ ЗАНЯТТЯ №3
ВИМОГИ ПРЕДМЕТНОЇ ОБЛАСТІ.
ВИМОГИ КОРИСТУВАЧА

Мета заняття: ознайомиться з вимогами предметної області. Навчиться розробляти вимоги предметної області. Ознайомиться з користувальницькими вимогами. Навчиться розробляти користувальницькі вимоги.

Теоретична частина

Вимоги предметної області відображають умови, у яких буде експлуатуватися програмна система. Вони можуть бути представлені у вигляді нових функціональних вимог, у вигляді обмежень на вже сформульовані функціональні вимоги або у вигляді вказівок, як система повинна виконувати обчислення. Ці вимоги дуже важливі, оскільки відображають ту предметну область, де буде використовуватися дана система. Невиконання вимог предметної області може привести до виходу системи з ладу.

Як приклад розглянемо вимоги до бібліотечної системи:

1. Стандартний користувальницький інтерфейс, що надає доступ до всіх бібліотечних баз даних.
2. Для забезпечення авторських прав деякі документи повинні бути вилучені із системи відразу після одержання. Для цього, залежно від бажання користувача, ці документи можуть бути роздруковані або на локальному системному сервері, або на мережному принтері.

Перша вимога є обмеженням на системну функціональну вимогу. Вона вказує, що користувальницький інтерфейс до баз даних повинен бути реалізований згідно відповідного бібліотечного стандарту. Друга вимога є зовнішньою і спрямована на виконання закону про авторські права, застосовувані до бібліотечних матеріалів. Із цієї вимоги випливає, що система повинна мати засіб "відправити_на_друк", застосовуваний автоматично для деяких типів бібліотечних документів.

Приклад вимог предметної області

Приведений приклад вимоги предметної області вказує, як повинні виконуватися обчислення. Він взятий зі специфікації системи автоматичного гальмування поїзда. Ця система повинна автоматично зупиняти поїзд на червоний сигнал семафора. Дана вимога вказує спосіб обчислення швидкості поїзда при гальмуванні. Тут використана термінологія, застосовувана при розрахунках швидкостей поїзда. Щоб розібратися в ній, необхідні відповідні знання про системи керування поїздами та їхніми характеристиками.

Гальмування поїзда обчислюється за формулою

$$D_{\text{поїзд}} = D_{\text{керування}} + D_{\text{градієнт}},$$

де $D_{\text{градієнт}}$ дорівнює 9,81 м·с^2 градієнт, що компенсує,/альфа. Значення 9,81 м·с^2/альфа відомо для всіх типів поїздів.

Наведений приклад показує основну проблему, пов'язану з вимогами предметної області. Вимоги цього типу використовують мову та позначення, властиві даній предметній області, що ускладнює їх розуміння розробниками ПЗ. Внаслідок цього вимоги предметної області не завжди виконуються так, як мається на увазі замовниками програмної системи.

Користувальницькі вимоги

Оператори

Користувальницькі вимоги до системи повинні описувати функціональні та нефункціональні системні вимоги так, щоб вони були зрозумілі навіть користувачеві, що не володіє спеціальними технічними знаннями. Ці вимоги повинні визначати тільки зовнішню поведінку системи, уникаючи по можливості визначення структурних характеристик системи. Користувальницькі вимоги повинні бути написані природною мовою з використанням простих таблиць, а також наочних і зрозумілих діаграм.

Разом з тим при описі вимог природною мовою можуть виникнути різні проблеми.

1. *Відсутність чіткості викладу.* Іноді нелегко викласти яку-небудь думку природною мовою чітко та недвозначно, не зробивши при цьому текст багатослівним і важким для читання.
2. *Змішання вимог.* У користувальницьких вимогах відсутній чіткий поділ на функціональні та нефункціональні вимоги, на системні цілі та проектну інформацію.
3. *Об'єднання вимог.* Кілька різних вимог до системи можуть описуватися як єдина користувальницька вимога.

Приклад користувальницької вимоги

Для точного позиціонування структурних елементів схеми користувач може відобразити на екрані сітку, параметри якої можуть задаватися (у сантиметрах або дюймах) за допомогою спеціальної опції на панелі керування. За замовчуванням сітка не відображається. Сітка може бути виведена на екран або прихована в будь-який момент сесії редагування, також у будь-який момент є можливість переходу із сантиметрів на дюйми та навпаки. Крок сітки повинен підлаштовуватися під розмір схеми.

У цій вимозі переплітаються не менше трьох різних вимог.

1. Концептуальна функціональна вимога: система редагування повинна мати в своєму розпорядженні можливість відображення сітки. Це основна причина появи даної вимоги.

2. Нефункціональна вимога, що надає докладну інформацію про те, у яких одиницях буде вимірюватися крок сітки (сантиметри або дюйми).

3. Нефункціональна користувальницька вимога, що висувається до інтерфейсу: як користувач може відобразити або приховати сітку.

Описана вимога містить й іншу інформацію, зокрема необхідну для ініціалізації системи. У вимозі сказано, що за замовчуванням сітка відключена. Разом з тим нічого не сказано про те, які одиниці виміру обрані за замовчуванням. Далі сказано, що користувач може перемикатися між сантиметрами та дюймами, але не сказано, чи він може змінювати крок сітки.

Коли користувальницька вимога містить так багато інформації, це ускладнює її розуміння та обмежує свободу розробника в пошуку рішення задачі, поставленої у вимозі. Користувальницькі вимоги повинні просто описувати основні можливості системи.

Для порівняння приведена відредагована користувальницька вимога, що фокусує увагу тільки на самому засобі відображення сітки без деталізації його властивостей.

Засіб відображення сітки

Редактор повинен мати засіб виводу на екран сітки, що складається з паралельних горизонтальних і вертикальних ліній, і повинна відображатися у вигляді фону на екрані редактора. Сітка – пасивний елемент, що полегшує вирівнювання користувачем структурних елементів схем.

Обґрунтування. Сітка повинна допомагати користувачеві створювати акуратну схему із правильно розміщеними елементами. Хоча активна сітка (така, у якій елементи "прив’язуються" до вузлів сітки) також може бути корисною, вона не забезпечує точного позиціонування елементів. Користувач визначить положення елементів схеми краще, ніж це зробить автоматизований засіб.

Зверніть увагу на обґрунтування вимоги. Воно допомагає розроблювачам зрозуміти, чому ця вимога включена до специфікації та у якій мірі вона може змінитися в майбутньому. Наприклад, в обґрунтуванні вимоги на засіб відображення сітки сказано, що також може бути корисною і активна сітка, де елементи схеми автоматично "прив’язуються" до вузлів сітки. Однак тут перевага свідомо віддана пасивній сітці. Якщо надалі виникне необхідність внести зміни в дану вимогу, то із цього обґрунтування буде видно, що варіант пасивної сітки обраний навмисно, а не з'явився на етапі реалізації системи.

Специфікація вимог

Наступний приклад вимоги також висувається до системи редагування схеми структури ПЗ. Це деталізована специфікація системної функції. У цьому випадку вимога включає список дій, які повинен виконати користувач для реалізації даної функції; іноді необхідно записати подібну послідовність дій, оскільки деякі функції повинні виконуватися тільки строго певним способом. Інформація про те, як реалізується дана функція, у цій вимозі відсутня.

Щоб звести до мінімуму неясності при написанні користувальницьких вимог, рекомендується дотримуватися наведених нижче правил.

1. Розробіть стандартну форму для запису користувальницьких вимог і неухильно її дотримуйтеся. Стандартна форма запису зменшує неясності у формулюванні вимог і дозволяє легко їх перевірити. Рекомендується включати

у форму запису вимоги не тільки саме її формулювання, але її обґрунтування та посилання на більше деталізовану специфікацію вимог.

2. Робіть розходження між обов'язковими та описовими вимогами. Тут обов'язковою вимогою є наявність засобу додавання нових структурних елементів, описовим – опис послідовності дій користувача. Описова вимога не є абсолютно необхідною для реалізації даної користувальницької вимоги та за необхідності може бути зміненою.

3. Використовуйте різні накреслення шрифту (напівжирне та курсив) для виділення ключових частин вимоги.

4. Уникайте по можливості комп'ютерного жаргону. Це не виключає використання технічних термінів тієї предметної області, для якої розробляється програмне забезпечення.

Хід роботи

1) Визначити вимоги предметної області та користувальницькі вимоги до системи відповідно до власного варіанту.
2) Оформити звіт (вимоги предметної області, користувальницькі вимоги, висновки).

Варіанти завдань

1) Інформаційна система ВНЗ
2) Інформаційна система торгової організації
3) Інформаційна система медицинських організацій міста
4) Інформаційна система автопідприємства міста
5) Інформаційна система проектної організації
6) Інформаційна система авіабудівельного підприємства
7) Інформаційна система військового округу
8) Інформаційна система будівельної організації
9) Інформаційна система бібліотечного фонду міста
10) Інформаційна система спортивних організацій міста
11) Інформаційна система автомобілебудівельного підприємства
12) Інформаційна система готельного підприємства
13) Інформаційна система магазину автозапчастин
14) Інформаційна система аптеки
15) Інформаційна система туристичного клубу.

Перелік літератури

1. Ian Sommerville. Software Engineering, 10th Edition. University of St Andrews, Scotland, 2015, 816 p.

ПРАКТИЧНЕ ЗАНЯТТЯ №4

СИСТЕМНІ ВИМОГИ

Мета заняття: ознайомиться із системними вимогами. Навчиться розробляти системні вимоги.

Теоретична частина

Системні вимоги

Системні вимоги – це більш деталізований опис користувальницьких вимог. Зазвичай, вони є основою для укладання контракту на розробку програмної системи і тому повинні представляти максимально повну специфікацію системи в цілому. Системні вимоги також використовуються як відправний пункт на етапі проектування системи.

Специфікація системних вимог може будуватися на основі різних системних моделей, таких, як об'єктна модель або модель потоків даних.

Методи опису вимог

Системні вимоги визначають, що саме повинна робити система, не вказуючи при цьому механізм її реалізації. Проте, з іншого боку, для повного опису системи потрібна деталізована інформація про неї, що по можливості повинна включати всю інформацію про системну архітектуру. На те існує ряд причин.

1. Первісна архітектура системи допомагає структурувати специфікацію вимог. Системні вимоги повинні описувати підсистеми, з яких складається розроблювальна система.

2. У більшості випадків розроблювальна система повинна взаємодіяти із вже існуючими системами. Це накладає певні обмеження на архітектуру нової системи.

3. В якості зовнішньої системної вимоги може виступати умова використання для розроблювальної системи спеціальної архітектури.

Специфікації системних вимог часто пишуться природною мовою. Але використання природної мови може породити певні проблеми при написанні деталізованої специфікації. Застосування природної мови має на увазі, що ті, хто пише специфікацію, і ті, хто її читає, ті самі слова й вираження розуміють однаково. Однак насправді це не так, оскільки природній мові властива певна розмитість понять. Внаслідок цього одна і та сама вимога може трактуватися різними людьми по-різному.

Щоб уникнути подібних проблем, розроблені методи опису вимог, які структурують специфікацію та зменшують розмитість визначень. Ці методи представлені в таблиці 4.1. Крім цього, розроблені інші підходи, наприклад спеціальні мови опису вимог, які використовуються відносно рідко.

Таблиця 4.1 – Способи запису специфікацій вимог

Система запису	Опис
Структурована природна мова	Використання стандартних форм і шаблонів для написання специфікації
Мови опису програм	Використання спеціальних структурованих мов, подібних мовам програмування, де специфікація вимог будується на основі обраної операційної моделі системи
Графічні нотації	Графічна мова, що використовує для опису функціональних вимог діаграми та блок-схеми, доповнені текстовими поясненнями. Найбільш відомий приклад такої графічної мови — діаграми структурного аналізу та проектування ПЗ (SADT).
Математичні специфікації	Системи нотацій, засновані на математичних концепціях, таких як теорія кінцевих автоматів або теорія множин. Це формалізований однозначний та позбавлений двозначності запис системних вимог. Однак багато замовників ПЗ не розуміють формальних специфікацій, внаслідок чого виникають певні проблеми при укладанні контрактів на розробку програмних продуктів

Хід роботи

1) Визначити системні вимоги до системи відповідно до власного варіанту.
2) Оформити звіт (системні вимоги до системи, висновки).

Варіанти завдань

1) Інформаційна система ВНЗ
2) Інформаційна система торгової організації
3) Інформаційна система медичних організацій міста
4) Інформаційна система автопідприємства міста
5) Інформаційна система проектної організації
6) Інформаційна система авіабудівельного підприємства
7) Інформаційна система військового округу
8) Інформаційна система будівельної організації
9) Інформаційна система бібліотечного фонду міста
10) Інформаційна система спортивних організацій міста
11) Інформаційна система автомобілебудівельного підприємства
12) Інформаційна система готельного підприємства
13) Інформаційна система магазину автозапчастин
14) Інформаційна система аптеки

15) Інформаційна система туристичного клубу.

Перелік літератури

1. Макконнелл С. Профессиональная разработка программного обеспечения. Санкт-Петербург, 2007.
2. Константайн. Разработка программного обеспечения. СПб.: Питер, 2004–592с.

ПРАКТИЧНЕ ЗАНЯТТЯ №5

ФОРМАЛЬНІ СПЕЦИФІКАЦІЇ ПЗ

Мета заняття: ознайомиться зі специфікаціями ПЗ. Навчиться розробляти специфікації.

Теоретична частина

Циклічні алгоритми

Визначені три рівні специфікації програмного забезпечення. Це користувальницькі, системні вимоги та специфікація структури програмної системи. Користувальницькі вимоги найбільш узагальнені, специфікація структури найбільш детальна. *Формальні математичні специфікації перебувають десь між системними вимогами й специфікацією структури. Вони не містять деталей реалізації системи, але повинні представляти її повну математичну модель.*

По мірі розробки специфікації участь замовника зменшується, а участь підрядників і безпосередньо розробників ПЗ зростає. На ранніх стадіях розробки специфікація повинна бути "орієнтована на замовника" і написана так, щоб він міг її зрозуміти. Однак на заключній стадії процесу розробки повинна бути отримана специфікація, в основному призначена для підрядників і розроблювачів ПЗ, оскільки вона буде основою для реалізації системи. Ця кінцева специфікація може бути формальною.

Розробка формальної специфікації і проектування

Етапи розробки специфікації, показані на рисунку 5.1, не є незалежними та не обов'язково проводяться в наведеній послідовності. На рисунку 5.2 показано, що розробка, специфікація та проектування можуть виконуватися паралельно, коли інформація від етапів розробки специфікації передається до етапів проектування та навпаки.

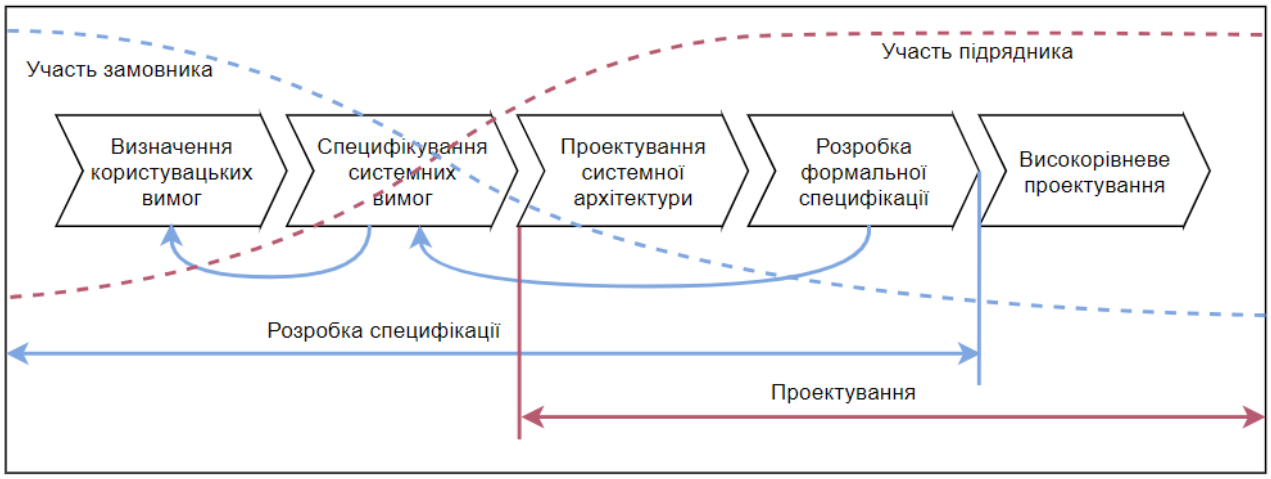

Рисунок 5.1 – Розробка специфікації та проектування

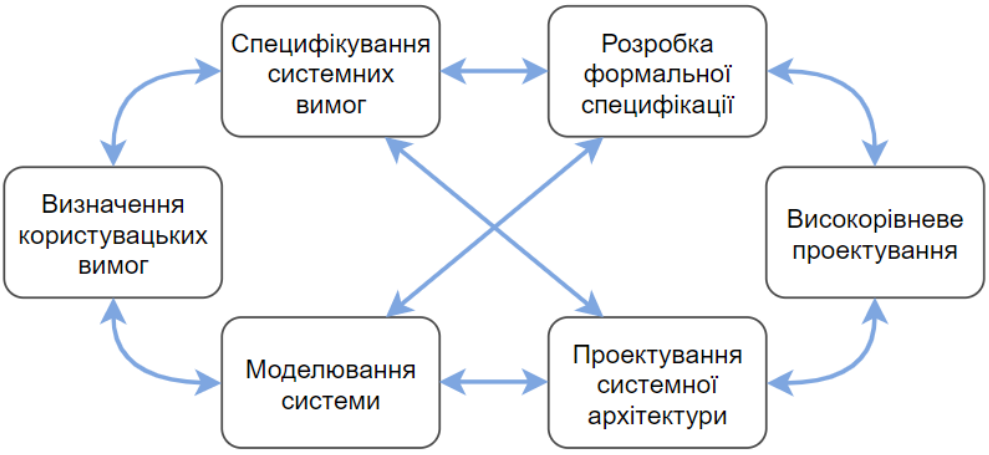

Рисунок 5.2 – Розробка формальної специфікації

Створення формальної специфікації вимагає детального аналізу системи, що дозволяє виявити помилки та невідповідності у специфікації неформальних вимог. Ця можливість виявлення помилок – найбільш важливий аргумент для використання формальної специфікації. Проблеми у вимогах, які залишаються невиявленими до останніх стадій процесу розробки ПЗ, звичайно вимагають більших витрат на виправлення.

Підходи до розробки формальної специфікації

Розробка та аналіз формальної специфікації вимагають додаткових витрат. На рисунку 5.3 показана вартість створення ПЗ при розробці формальної специфікації та без неї. При звичайному процесі розробки ПЗ вартість атестації системи становить близько 50% всієї вартості розробки, а вартість проектування та реалізації системи удвічі перевищує вартість розробки специфікації. При використанні формальної специфікації вартості розробки специфікації й реалізації системи порівнянні, а вартість атестації значно знижується, оскільки в процесі розробки формальної специфікації виявляються та усуваються недоробки у вимогах, тим самим виключається переробка системи на останніх стадіях її створення.

Існує два основні підходи до розробки формальної специфікації, які використовуються для написання деталізованих специфікацій нетривіальних програмних систем.

1. Алгебраїчний підхід, за яким система описується в термінах операцій та їх відносин.

2. Підхід орієнтований на моделювання, за якого модель системи будується з використанням математичних конструкцій, таких, як множини та послідовності, а системні операції визначаються тим, як вони змінюють стани системи.

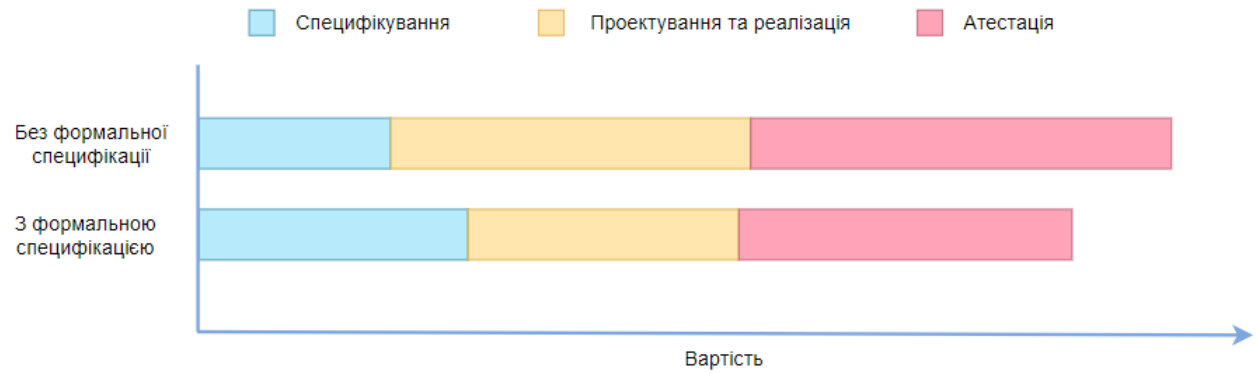

Рисунок 5.3 – Вартість розробки ПЗ з формальною специфікацією

Формальна специфікація

Формальна специфікація – завершений опис моделі системи та вимог до її поведінки в термінах того чи іншого формального методу.

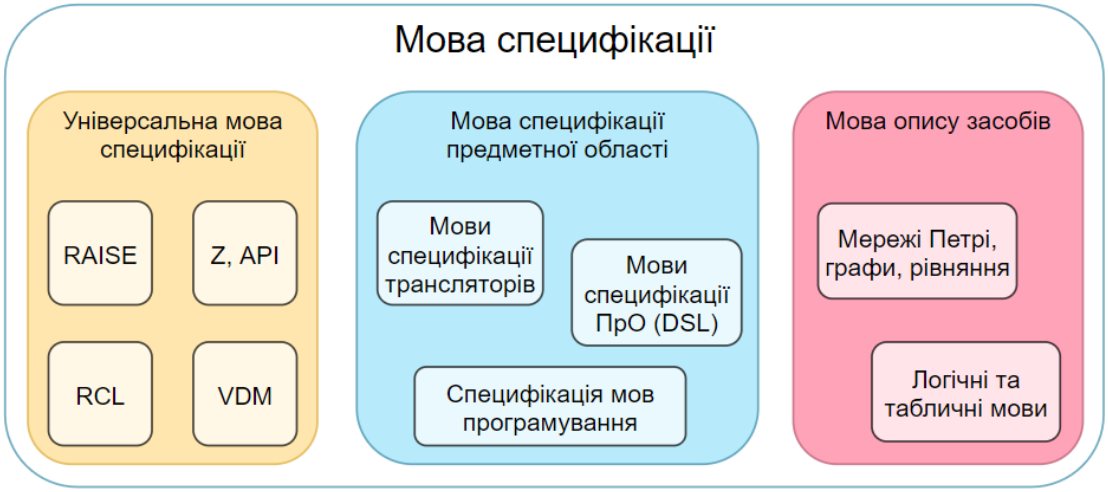

Рисунок 5.4 – Мови специфікацій

Формальний метод (метод формальної розробки ПЗ) – це набір методів та інструментальних засобів, що базуються на математичних засадах (моделювання, математична логіка, теорія множин, теорія скінченних автоматів, алгебра, тощо) та використовуються для формальної специфікації, верифікації, аналізу вимог до ПЗ та предметних областей. Такі інструментальні засоби іноді ще називають системою формальних міркувань. Як правило, окрім системи формальних міркувань формальні методи включають стандартизовані мови (мови специфікацій, формальні нотації). Приклади формальних методів: CSP, CCS, OBJ, VDM, B-метод, Z-метод, RAISE.

Формальна мова (мова специфікацій, формальна нотація) – це мова з точно визначеним синтаксисом та семантикою. Мови формальних специфікацій використовуються для написання специфікації, тобто для опису властивостей певної предметної області. Це, наприклад, мови Z, B, CLEAR, LARCH.

Формальна розробка – систематичне перетворення специфікації у виконуваний код з використанням певних формалізованих правил (певних формальних методів).

Методи формальної розробки базуються на використанні мови специфікацій та засобів формальних міркувань для побудови ПЗ. При цьому засоби формальних міркувань базуються на мовах формальних специфікацій, математичних засадах та використовуються для аналізу моделі системи та самої системи. Мови формальних специфікацій, в свою чергу, базуються на математичному апараті (наприклад, числення предикатів, алгебра, теорія скінченних автоматів) та використовуються для побудови моделі системи.

Існує низка підходів до методів формальної розробки:

- операційний підхід – розробка системи базується на описі моделі, яка має властивості розроблюваної системи;
- аксіоматичний підхід – розробка базується на аксіоматичному описі;
- алгебричний підхід – розробка базується на алгебричному описі;
- гібридний підхід об'єднує операційний підхід з аксіоматичним або кроки при формальній розробці об'єднані та супроводжуються аналізом, який базується на перевірці властивостей системи. Така перевірка здійснюється засобами формальних міркувань за допомогою двох основних підходів.

Автоматичне доведення теорем (*theorem proving*) – доведення теорем (логічних тверджень про властивості системи) за допомогою програмного забезпечення. В основі автоматичного доведення теорем лежить апарат математичної логіки.

Перевірка моделі (*model checking*) – процес перевірки, що визначає чи є побудована структура моделлю заданої предметної області.

Досвід використання формальних методів. Існує ряд прикладів використання формальних методів (ФМ), та зокрема методу Z для специфікації апаратних систем.

IBM використовувала метод Z для ре-специфікації великої (більше півмільйона рядків коду) системи обробки транзакцій CICS, в результаті чого покращила її підтримуваність. Крім того, для проекту CICS, незалежний аудит підтвердив економію коштів у розмірі 9% завдяки застосуванню специфікацій на Z, що зменшило кількість помилок та підвищило якість коду.

Британська компанія Plessey розробила специфікації для комп'ютерної архітектури надійної мультиобробки інформації Type Environment. Модуль роботи з числами з плаваючою комою для трансп'ютера був специфікований на Z, при цьому були виявлені помилки в реалізації.

Tektronix використала метод Z для специфікації функціональності осцилографу.

Dansk Datamatik Center вже багато років розробляє індустріальні компілятори, використовуючи формальні методи.

Прикладами використання формальних методів є Європейський стандарт для програмної інженерії Portable Common Tools Environment та специфікації програмно-інженерних середовищ на основі баз даних.

Компанія Rolls-Royce використовувала формальні методи для специфікації програми керування ядерним реактором.

Є ряд стандартів та класів систем, де застосування формальних методів для розробки регламентується та вимагається. Зокрема:

Міжнародна електротехнічна комісія (International Electrotechnical Commission, IEC – міжнародна організація з стандартизації в галузі електричних, електронних та суміжних технологій) вказує ряд формальних методів (CCS, CSP, HOL, LOTOS, OBJ, VDM, Z, Temporal Logic) для розробки критичних до безпеки систем.

Європейська космічна агенція рекомендує використання VDM або Z, доповнених описами на природних мовах, для специфікації вимог до критичних систем, а також доведення коректності та застосування формальних виведень перед тестуванням.

Міністерство оборони Великобританії поширює використання формальних методів у своїх стандартах та вимагає: використання формальної нотації в специфікації критичних до безпеки компонент, аналіз таких компонент з точки зору несуперечливості та повноти, все критичне до безпеки програмне забезпечення має бути верифіковане та валідоване (що включає формальне доведення та строге, але неформальне, доведення коректності, статичний та динамічний аналіз), програмні та електронні компоненти обладнання для оборонних цілей мають бути класифіковані відповідно до аналізу ризиків.

Наглядова рада з атомної енергетики Канади також вимагає застосування формальних методів для розробки програмного забезпечення безпекових систем атомних електростанцій.

Використання специфікацій програм на різних етапах життєвого циклу. Досвід показує, що використання математичних методів для специфікацій та моделювання програмного забезпечення може сприяти розв’язанню проблеми побудови надійного програмного забезпечення. Математичні методи, які-б ефективно використовувалися у виробництві програмного забезпечення, мають задовольняти деяким основним вимогам.

По-перше, нотації (записи), які застосовуватимуться в специфікаціях, мають бути стандартизовані, а отже виробництво багатьох проектів може здійснюватися групами розробників без непорозумінь. Це дає можливість використовувати вже розроблені специфікації.

По-друге, програмні засоби потребують автоматизації маніпулювання з формалізованим текстом. Обидві ці передумови вимагають від таких нотацій стабільності та ясності.

Традиційно виділяють наступні фази життєвого циклу програми:

- аналіз вимог;
- специфікація;
- проектування;
- реалізація;
- тестування та відлагодження;
- експлуатація та супроводження.

Специфікації можуть використовуватися на різних фазах життєвого циклу ПЗ, зокрема:

- при аналізі вимог специфікації можуть використовуватися для уточнення вимог, узгоджень їх із замовником, для побудови прототипів;
- при проектуванні – для контролю правильності проекту;
- при реалізації – для формулювання завдань розробникам та створення документації;
- при тестуванні – для перевірки виконання вимог;
- при супроводженні – для уточнення змін, підтримки узгодженості документації з системою, та інше.

Однією з важливих переваг використання специфікацій є збільшення глибини розуміння системи, що уточнюється. В процесі створення специфікації розробники мають більше можливостей для виявлення недоліків, непослідовностей, неоднозначностей та неповноти проекту. Специфікація є корисним засобом зв'язку між замовником та проектувальником, між проектувальником та розробником, а також між розробником та тестувальником.

Вона часто виступає як супровідна документація до програмного коду системи, але має високий рівень опису. Однією з важливих теоретичних причин використання формальних методів (ФМ) специфікацій програм є те, що самі програми є математичними об'єктами, так як вони мають формальну семантику та виражені на формальній мові, а отже, можуть бути оброблені за допомогою математичних засобів. Таким чином, виникає можливість доведення певних властивостей програм, зокрема властивості правильності, за допомогою математичних методів.

Останнім часом серед практичних методів проектування програмних систем (ПС) найбільшого розповсюдження набуло об'єктно-орієнтоване проектування (ООП). Визначилась технологічна база розробки різних видів програмного забезпечення методами ООП, зокрема, мова UML (Unified Modeling Language) та її інструментальна підтримка (Rational Rose, MS Visio, Visual Paradigm та інші). ООП стало базисом генеруючого (породжуючого) програмування, яке включає в себе і інші методи програмування. Зазначені методи підтримують розробку ПС на усіх етапах життєвого циклу.

Хід роботи

1) Використовуючи раніше розроблені матеріали, розробити специфікацію системи відповідно до власного варіанту.
2) Оформити звіт (специфікація системи, висновки).

Перелік літератури

1. Макконнелл С. Профессиональная разработка программного обеспечения. Санкт-Петербург, 2007.
2. Константайн. Разработка программного обеспечения. СПб.: Питер, 2004 – 592с.

ПРАКТИЧНЕ ЗАНЯТТЯ №6

АРХІТЕКТУРНЕ ПРОЕКТУВАННЯ. СТРУКТУРУВАННЯ СИСТЕМИ. БАГАТОПРОЦЕСОРНА АРХІТЕКТУРА

Мета заняття: ознайомитися з архітектурним проектуванням. Спроектувати одну із систем. Ознайомитися зі структурами систем. Створити структуру однієї із систем.

Теоретична частина

Архітектурне проектування

Великі системи завжди можна розбити на підсистеми, що надають зв'язані набори сервісів. *Архітектурним проектуванням* називають перший етап процесу проектування, на якому визначаються підсистеми, а також структура керування та взаємодії підсистем. Метою архітектурного проектування є опис *архітектури програмного забезпечення.*

Процес архітектурного проектування

Існують різні підходи до процесу архітектурного проектування, які залежать від професійного досвіду, а також майстерності та інтуїції розробників. І все-таки можна виділити кілька етапів, загальних для всіх процесів архітектурного проектування.

1. *Структурування системи.* Програмна система структурується у вигляді сукупності відносно незалежних підсистем. Також визначаються взаємодії між підсистемами.

2. *Моделювання керування.* Розробляється базова модель керування взаєминами між частинами системи.

3. *Модульна декомпозиція.* Кожна певна на першому етапі підсистема розбивається на окремі модулі. Тут визначаються типи модулів і типи їхніх взаємозв'язків.

Як правило, ці етапи перемежовуються та накладаються один на одного. Етапи повторюються для детальнішого пророблення архітектури доти, поки архітектурний проект не буде задовольняти системним вимогам.

Чітких розходжень між підсистемами та модулями немає, але будуть корисними наступні визначення.

1. *Підсистема* – це система (тобто задовольняє "класичному" визначенню "система"), операції (методи) якої не залежать від сервісів, що надаються іншими підсистемами. Підсистеми складаються з модулів і мають певні інтерфейси, за допомогою яких взаємодіють з іншими підсистемами.

2. *Модуль* – це зазвичай компонент системи, що надає один або кілька сервісів для інших модулів. Модуль може використовувати сервіси, підтримувані іншими модулями. Як правило, модуль ніколи не розглядається як

незалежна система. Модулі зазвичай складаються з ряду інших, простіших компонентів.

Архітектурні моделі

Результатом процесу архітектурного проектування є документ, що відображає архітектуру системи. Він складається з набору графічних схем подань моделей системи з відповідним описом. В описі повинне бути зазначене, з яких підсистем складається система і з яких модулів складається кожна підсистема. Графічні схеми моделей системи дозволяють глянути на архітектуру з різних сторін. Як правило, розробляється чотири архітектурні моделі.

1. Статична структурна модель, у якій представлені підсистеми або компоненти, розроблювальні надалі незалежно.
2. Динамічна модель процесів, у якій представлена організація процесів під час роботи системи.
3. Інтерфейсна модель, що визначає сервіси, надавані кожною підсистемою через загальний інтерфейс.
4. Моделі відносин, у яких показані взаємини між частинами системи, наприклад потік даних між підсистемами.

Ряд дослідників при описі архітектури систем пропонують використовувати спеціальні мови опису архітектур. У них основними архітектурними елементами є компоненти та коннектори (об'єднуючі ланки); ці мови також пропонують принципи й правила побудови архітектур. Однак, як і інші спеціалізовані мови, вони мають один недолік, а саме: всі вони зрозумілі тільки фахівцям, що їх освоїли, і майже не використовуються на практиці. Фактично використання мов опису архітектур тільки ускладнює аналіз систем. Тому вважається, що для опису архітектур краще використовувати неформальні моделі й системи нотації, подібні пропонованої, наприклад уніфікований мова моделювання UML.

Вплив архітектури на характеристики системи

Архітектура системи впливає на продуктивність, надійність, зручність супроводу та інші характеристики. Тому моделі архітектури, обрані для даної системи, можуть залежати від нефункціональних системних вимог.

1. *Продуктивність.* Якщо критичною вимогою є продуктивність системи, варто розробити таку архітектуру, щоб за всі критичні операції відповідало якнайменше підсистем з максимально малою взаємодію між ними. Щоб зменшити взаємодію між компонентами, краще використовувати крупномодульні компоненти, а не дрібні структурні елементи.
2. *Захищеність.* У цьому випадку архітектура повинна мати багаторівневу структуру, у якій найбільш критичні системні елементи захищені на внутрішніх рівнях, перевірка безпеки цих рівнів здійснюється на більш високому рівні.
3. *Безпека.* У цьому випадку архітектуру варто спроектувати так, щоб за всі операції, що впливають на безпеку системи, відповідало якнайменше

підсистем. Такий підхід дозволяє знизити вартість розробки та вирішує проблему перевірки надійності.

4. *Надійність.* У цьому випадку варто розробити архітектуру із включенням надлишкових компонентів, щоб можна було заміняти та обновляти їх, не перериваючи роботу системи.

5. *Зручність супроводу.* У цьому випадку архітектуру системи варто проектувати на рівні дрібних структурних компонентів, які можна легко змінювати. Програми, що створюють дані, повинні бути відділені від програм, що використовують ці дані. Варто також уникати структур спільного використання даних.

Очевидно, що деякі з перерахованих архітектур суперечать одна одній. Наприклад, для того щоб підвищити продуктивність, необхідно використовувати крупномодульні компоненти, у той же час супровід системи набагато спрощується, якщо вона складається із дрібних структурних компонентів. Якщо необхідно врахувати обидві вимоги, варто шукати компромісне рішення. Раніше вже було сказано, що один зі способів рішення подібних проблем міститься у застосуванні різних архітектурних моделей для різних частин системи.

На першому етапі процесу проектування архітектури система розбивається на кілька взаємодіючих підсистем. На самому абстрактному рівні архітектуру системи можна зобразити графічно за допомогою блок-схеми, у якій окремі підсистеми представлені окремими блоками. Якщо підсистему також можна розбити на кілька частин, на діаграмі ці частини зображуються прямокутниками усередині більших блоків. Потоки даних і/або потоки керування між підсистемами позначається стрілками. Така блок-схема дає загальне уявлення про структуру системи.

Приклад структурної моделі архітектури

На рисунку 6.1 представлена структурна модель архітектури для системи керування автоматичним упакуванням різних типів об'єктів. Вона складається з декількох частин. Підсистема спостереження вивчає об'єкти на конвеєрі, визначає тип об'єкта та вибирає для нього відповідний тип упакування. Потім об'єкти знімаються з конвеєра, упаковуються та переміщуються на інший конвеєр.

Бэсс (Bass) вважає, що подібні блок-схеми є марними поданнями системної архітектури, оскільки з них не можна нічого довідатися ані про природу взаємин між компонентами системи, ані про їхні властивості. З точки зору розробника програмного забезпечення, це абсолютно вірно. Однак такі моделі виявляються ефективними на етапі попереднього проектування системи. Ця модель не обтяжена подробицями, з її допомогою зручно представити структуру системи. У структурній моделі визначені всі основні підсистеми, які можна розробляти незалежно від інших підсистем, отже, керівник проекту може розподілити розробку цих підсистем між різними виконавцями. Звичайно,

для передзмісту архітектури використовуються не тільки блок-схеми, однак подібне подання системи не менш корисне, ніж інші архітектурні моделі.

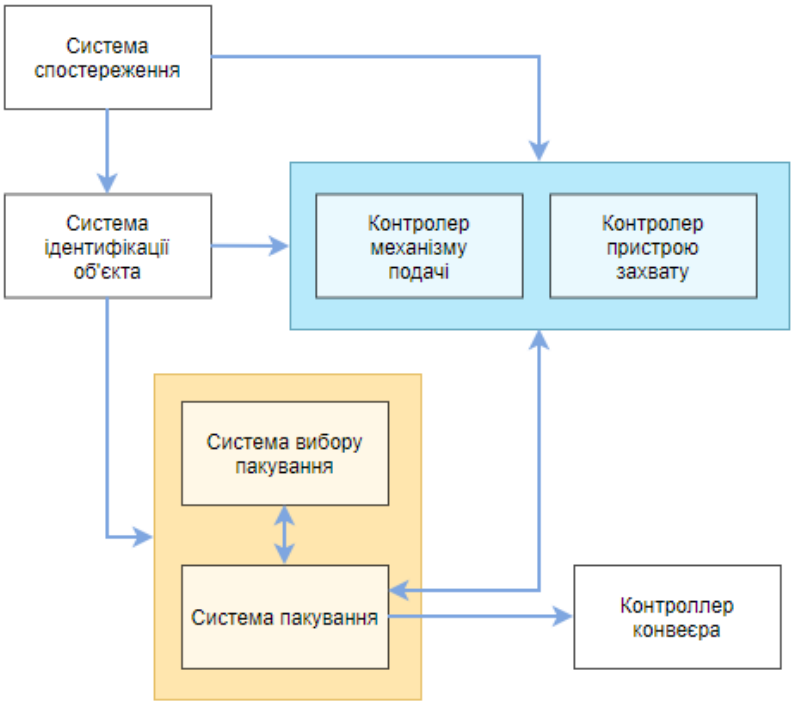

Рисунок 6.1 – Структурна схема системи автоматичного упакування

Багатопроцесорна система

Найпростішою розподіленою системою є багатопроцесорна система. Вона складається із безлічі різних процесів, які можуть (але не обов'язково) виконуватися на різних процесорах. Дана модель часто використовується у великих системах реального часу. Ці системи збирають інформацію, приймають на її основі рішення та відправляють сигнали виконавчому механізмові, що змінює системне оточення. У принципі всі процеси, пов'язані зі збиранням інформації, прийняттям рішень і керуванням виконавчим механізмом, можуть виконуватися на одному процесорі під управлінням планувальника завдань. Використання декількох процесорів підвищує продуктивність системи і її здатність до відновлення. Розподіл процесів між процесорами може перевизначатися (властиво критичним системам) або ж перебувати під управлінням диспетчера процесів.

На рисунку 6.2 показаний приклад системи такого типу. Це спрощена модель системи керування транспортним потоком. Група розподілених датчиків збирає інформацію про величину потоку. Зібрані дані перед відправленням у диспетчерську обробляються на місці. На підставі отриманої інформації оператори приймають рішення та управляють світлофорами. У цьому прикладі для керування датчиками, диспетчерською та світлофорами є окремі логічні процеси. Це можуть бути як окремі процеси, так і група процесів. В нашому прикладі вони виконуються на різних процесорах.

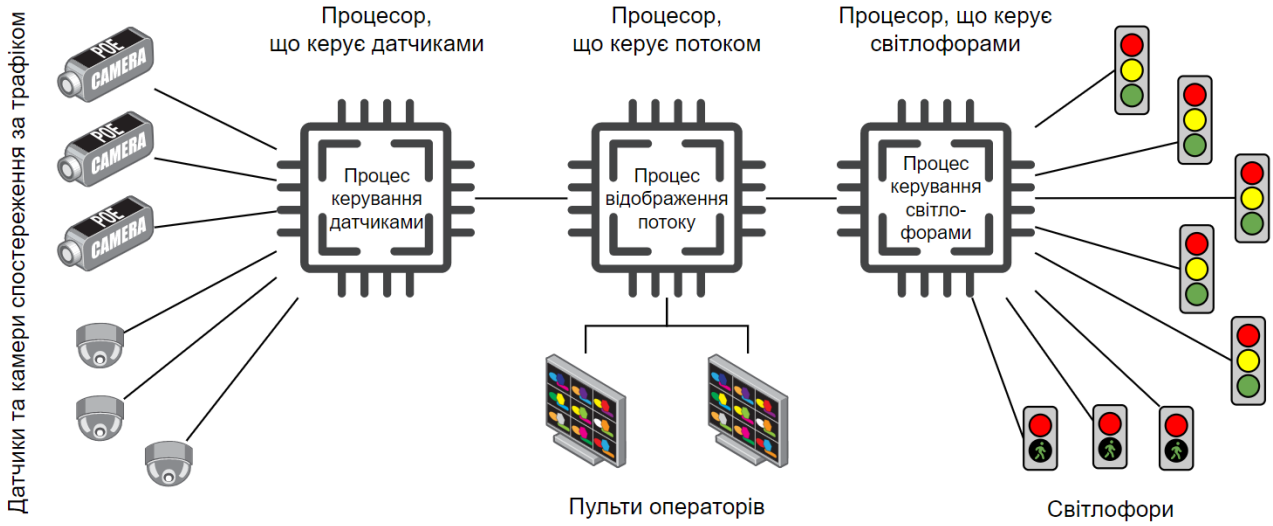

Рисунок 6.2 – Приклад багатопроцесорної системи

Системи, що одночасно виконують безліч процесів, не обов'язково є розподіленими. Якщо в системі більше одного процесора, реалізація розподілу процесів не є складною задачею. Однак при створенні багатопроцесорних програмних систем не обов'язково відштовхуватися тільки від розподілених систем. При проектуванні систем такого типу, по суті, використовується той же підхід, що і при проектуванні систем реального часу.

Клієнт-серверна архітектура

В архітектурі клієнт-сервер програмний додаток моделюється як набір сервісів, які надаються серверами, та безліч клієнтів, що використовують ці сервіси. Клієнти повинні знати про доступні (наявні) сервери, хоча можуть і не мати уявлення про існування інших клієнтів. Як видно з рисунку 6.3, клієнти та сервери представляють різні процеси.

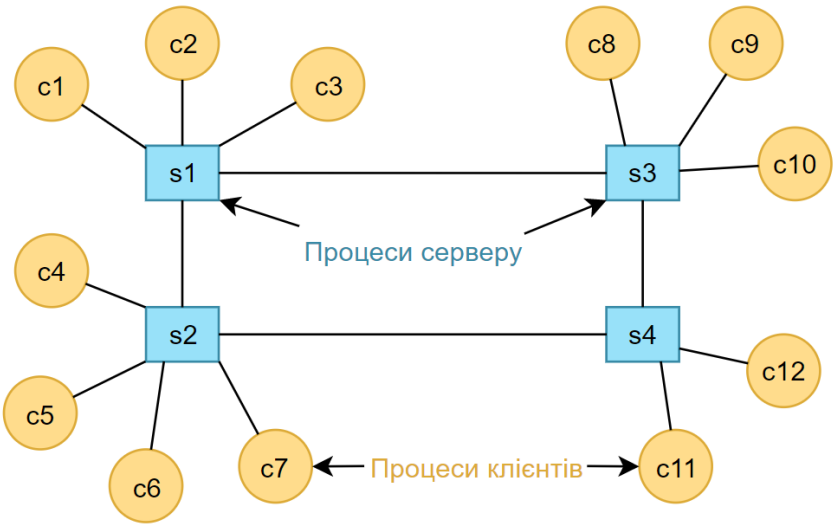

Рисунок 6.3 – Система клієнт-сервер

У системі між процесами та процесорами не обов'язково має дотримуватися відношення "один до одного". На рисунку 6.4 зображена фізична архітектура системи, яка складається із п’яти клієнтських машин та двох серверів. На них запускаються клієнтські серверні процеси, зображені на рисунку 6.3. У загальному випадку, говорячи про клієнтів та сервери, мають на увазі скоріше логічні процеси, ніж фізичні машини, на яких ці процеси виконуються.

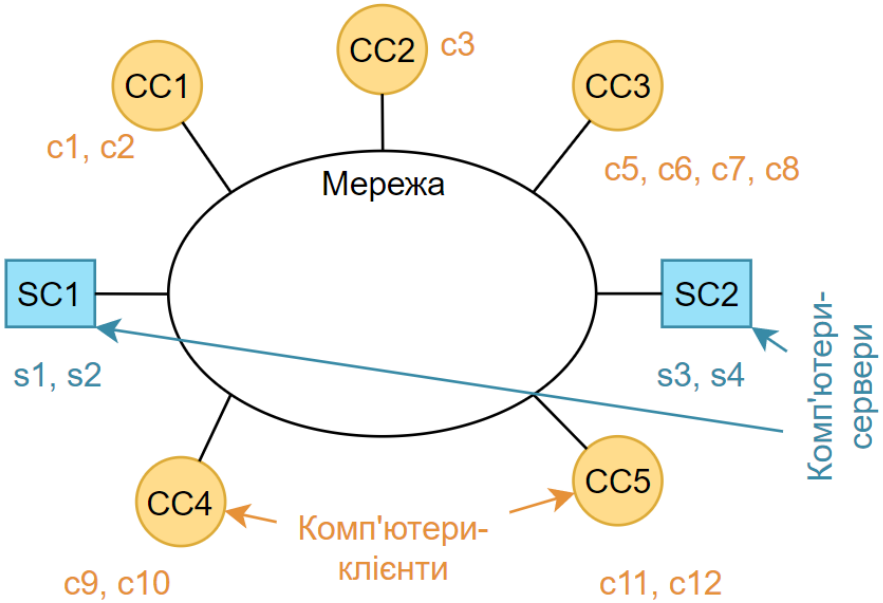

Рисунок 6.4 – Машини у мережі клієнт-сервер

Архітектура системи клієнт-сервер повинна відбивати логічну структуру програмного додатка. На рисунку 6.5 пропонується ще один погляд на програмний додаток, структурований у вигляді трьох рівнів. Рівень подання забезпечує інформацію для користувачів та взаємодію з ними. Рівень виконання додатка реалізує логіку роботи додатка. На рівні керування даними виконуються всі операції з базами даних. У централізованих системах між цими рівнями немає чіткого розподілу. Однак при проектуванні розподілених систем необхідно розділяти ці рівні, щоб потім розташувати кожний рівень на різних комп'ютерах.

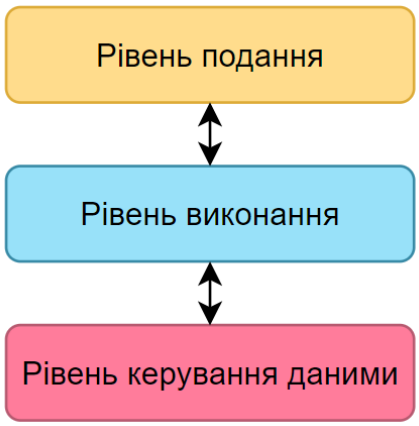

Рисунок 6.5 – Рівні програмного додатку

Найпростіша клієнт-серверна архітектура – дворівнева: додаток складається із сервера (або множини ідентичних серверів) та групи клієнтів.

Існує два види такої архітектури (рисунок 6.6).

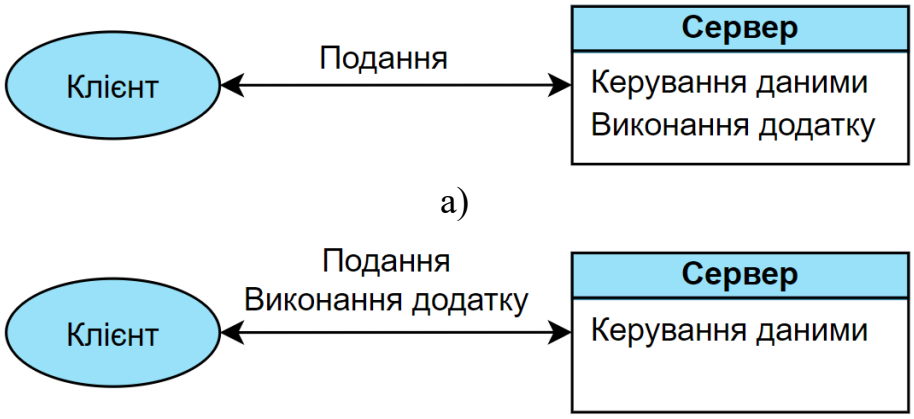

Рисунок 6.6 – Моделі дворівневої архітектури:
а – тонкий клієнт; б – товстий клієнт

1. *Модель тонкого клієнта.* У цій моделі (рисунок 6.6 а) вся робота додатку та керування даними виконуються на сервері. На клієнтській машині запускається тільки ПЗ рівня подання.

2. *Модель товстого клієнта.* У цій моделі (рисунок 6.6 б) сервер тільки керує даними. На клієнтській машині реалізована робота додатку та взаємодія з користувачем системи.

Тонкий клієнт дворівневої архітектури – найпростіший спосіб перенесення існуючих централізованих систем в архітектуру клієнт-сервер. Користувальницький інтерфейс у цих системах розміщується на персональному комп'ютері, а сам програмний додаток виконує функції сервера, тобто виконує всі процеси додатка та управляє даними. Модель тонкого клієнта можна також реалізувати там, де клієнти є звичайними мережними пристроями, а не персональними комп'ютерами або робочими станціями. Мережні пристрої запускають Internet-браузер та користувальницький інтерфейс, реалізований всередині системи.

Головний недолік моделі тонкого клієнта – більша завантаженість сервера та мережі. Всі обчислення виконуються на сервері, а це може привести до значного мережевого трафіка між клієнтом та сервером. У сучасних комп'ютерах досить обчислювальної потужності, проте вона практично не використовується в моделі тонкого клієнта банку.

Навпаки, модель товстого клієнта використовує обчислювальну потужність локальних машин. Рівень виконання додатку і рівень подання розміщуються на клієнтському комп'ютері. Сервер тут, по суті, є сервером транзакцій, що керує всіма транзакціями баз даних. Прикладом архітектури

такого типу можуть служити системи банкоматів, у яких банкомат є клієнтом, а сервер – центральним комп'ютером, що обслуговує базу даних по розрахунках з клієнтами.

На рисунку 6.7 представлена мережна система банкоматів. Банкомати пов'язані з базою даних розрахунків не напряму, а через монітор телеобробки. Цей монітор є проміжною ланкою, що взаємодіє з віддаленими клієнтами та організовує запити клієнтів у послідовність транзакцій для роботи з базою даних. Використання послідовних транзакцій при виникненні збоїв дозволяє системі відновитися без втрати даних.

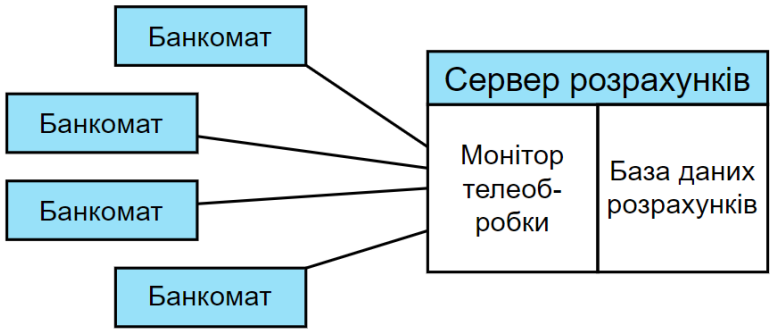

Рисунок 6.7 – Клієнт-серверна модель мережі банкоматів

Поява мови Java та аплетів дозволили розробляти клієнт-серверні моделі, які перебувають десь посередині між моделями тонкого та товстого клієнта. Частину програм, що становлять додаток, можна завантажувати на клієнтській машині як аплети Java, і тим самим розвантажити сервер. Інтерфейс користувача будується за допомогою Web-браузера, що запускає аплети Java. Однак Web-браузери від різних виробників і навіть різні версії Web-браузерів від одного виробника не завжди виконуються однаково. Більш ранні версії браузерів на старих машинах не завжди можуть запустити аплети Java. Отже, такий підхід можна використовувати тільки тоді, коли є впевненість, що у всіх користувачів системи встановлені браузери, сумісні з Java.

У дворівневій моделі клієнт-серверу істотною проблемою є розташування на двох комп'ютерних системах трьох логічних рівнів – подання, виконання додатка та керування даними. Тому в даній моделі часто виникають або проблеми з масштабованістю та продуктивністю, якщо обрано модель тонкого клієнта, або проблеми, пов'язані з керуванням системою, якщо використовується модель товстого клієнта. Щоб уникнути цих проблем, необхідно застосувати альтернативний підхід – трирівневу модель архітектури клієнт-сервер (рисунок 6.8). У цій архітектурі рівням подання, виконання додатка й керування даними відповідають окремі процеси.

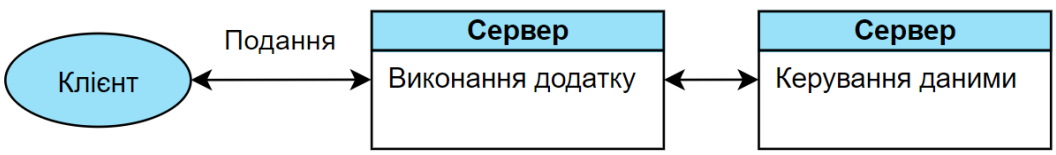

Рисунок 6.8 – Трирівнева архітектура клієнт-сервер

Архітектура ПЗ, побудована за трирівневою моделі клієнт-сервер, не вимагає, щоб три комп'ютерні системи були об'єднані у мережу. На одному комп'ютері-сервері можна запустити і виконання додатку, і керування даними як окремі логічні сервери. У той же час, якщо вимоги до системи зростуть, можна буде відносно просто розділити виконання додатка та керування даними та виконувати їх на різних процесорах.

Банківську систему, що використовує Internet-сервіси, можна реалізувати за допомогою трирівневої архітектури клієнт-сервер. База даних розрахунків (зазвичай розташована на головному комп'ютері) надає сервіси керування даними, Web-сервер підтримує сервіси додатка, наприклад засобу переказу грошей, генерацію звітів, оплату рахунків та ін. А машина користувача з Internet-браузером є клієнтом. Як показано на рисунку 6.9, ця система є масштабованою, тому що до неї відносно просто додати нові Web-Сервери при збільшенні кількості клієнтів.

Використання трирівневої архітектури в цьому прикладі дозволило оптимізувати передачу даних між Web-сервером і сервером бази даних. Взаємодію між цими системами не обов'язково будувати на стандартах Internet, можна використовувати більш швидкі комунікаційні протоколи низького рівня. Звичайно, інформацію від бази даних обробляє ефективне проміжне ПЗ, що підтримує запити до бази даних мовою структурованих запитів SQL.

У деяких випадках трирівневу модель клієнт-сервера можна перевести в багаторівневу, додавши до системи додаткові сервери. Багаторівневі системи можна використовувати і там, де додаткам необхідно мати доступ до інформації, що перебуває в різних базах даних. У цьому випадку об'єднуючий сервер розташовується між сервером, на якому виконується додаток, і серверами баз даних. Об'єднуючий сервер збирає розподілені дані та представляє їх у додатку в такий спосіб, ніби вони перебувають в одній базі даних.

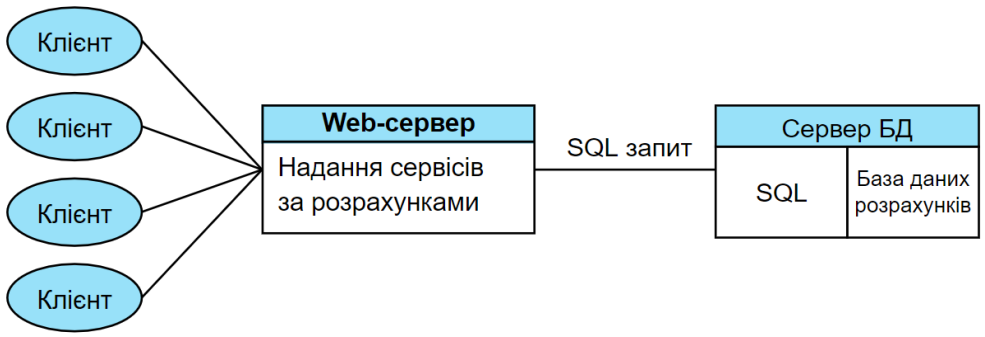

Рисунок 6.9 – Розподілена архітектура банківської системи з використанням Internet-сервісів

Розробники архітектур клієнт-сервер, вибираючи найбільш доцільну, повинні враховувати ряд факторів. У таблиці 6.1 перераховані різні випадки застосування архітектури клієнт-сервер.

Таблиця 6.1 – Застосування архітектур клієнт-сервер

Архітектура	Додатки
Дворівнева архітектура тонкого клієнта	Наслідувані системи, у яких недоцільно розділяти виконання додатка та керування даними. Додатки з інтенсивними обчисленнями, наприклад компілятори, але з незначним обсягом керування даними. Додатки, у яких обробляються великі масиви даних (запити), але з невеликим обсягом обчислень у самому додатку.
Дворівнева архітектура товстого клієнта	Додатки, де користувачеві потрібна інтенсивна обробка даних (наприклад, візуалізація даних або більші обсяги обчислення). Додатки з відносно постійним набором функцій на стороні користувача, застосовуваних у середовищі з добре налагодженим системним керуванням. Великі додатки з 100+ клієнтів.
Трирівнева та багаторівнева архітектури клієнт-сервер	Додатки, у яких часто змінюються і дані, і методи обробки. Додатки, в яких виконується інтеграція даних з багатьох джерел.

Архітектура розподілених об’єктів

У моделі клієнт-серверної розподіленої системи між клієнтами та серверами існують розбіжності. Клієнт запитує сервіси тільки в сервера, але не в інших клієнтів; сервери можуть функціонувати як клієнти та запитувати сервіси в інших серверів, але не в клієнтів; клієнти повинні знати про сервіси, надавані певними серверами, і про те, як взаємодіють ці сервери. Така модель добре підходить до багатьох типів додатків, але в той же час обмежує розробників системи, які змушені вирішувати, де надавати сервіси. Вони також повинні забезпечити підтримку масштабованості та розробити засоби включення клієнтів у систему на розподілених серверах.

Більш загальний підхід, що застосовується у проектуванні розподілених систем, є стирання розходжень між клієнтом та сервером і проектування архітектури системи як архітектури розподілених об’єктів. У цій архітектурі (рисунок 6.10) основними компонентами системи є об’єкти, що надають набір сервісів через свої інтерфейси. Інші об’єкти викликають ці сервіси, не роблячи розходжень між клієнтом (користувачем сервісу) і сервером (постачальником сервісу).

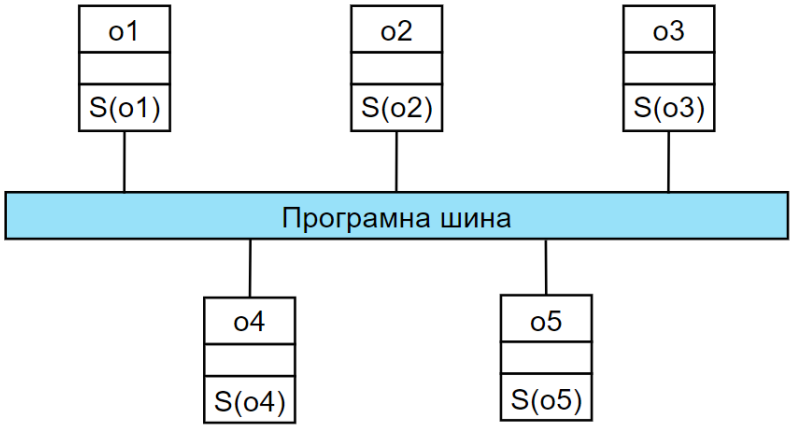

Рисунок 6.10 – Архітектура розподіленої системи

Об’єкти можуть розташовуватися на різних комп’ютерах у мережі та взаємодіяти за допомогою проміжного ПЗ. За аналогією із системною шиною, яка дозволяє підключати різні пристрої та підтримувати взаємодію між апаратними засобами, проміжне ПЗ можна розглядати як шину програмного забезпечення. Вона надає набір сервісів, що дозволяє об’єктам взаємодіяти один з одним, додавати або видаляти їх із системи. Проміжне ПЗ називають брокером запитів до об’єктів. Його завдання – забезпечувати інтерфейс між об’єктами.

Основні переваги моделі архітектури розподілених об’єктів:

- Розробники системи можуть не поспішати із прийняттям рішень щодо того, де і як будуть надаватися сервіси. Об’єкти, що надають сервіси, можуть виконуватися в будь-якому місці (вузлі) мережі. Отже, розходження між моделями товстого і тонкого клієнтів стають несуттєвими, тому що немає необхідності заздалегідь планувати розміщення об’єктів для виконання додатка.
- Системна архітектура досить відкрита, що дозволяє при необхідності додавати в систему нові ресурси. Стандарти програмної шини постійно вдосконалюються, що дозволяє об’єктам, написаним на різних мовах програмування, взаємодіяти та надавати сервіси один одному.
- Гнучкість і масштабованість системи. Для того, щоб упоратися із системним навантаженнями, можна створювати екземпляри системи з однаковими сервісами, які будуть надаватися різними об’єктами або різними екземплярами (копіями) об’єктів. При збільшенні навантаження в систему можна додати нові об’єкти, не перериваючи при цьому роботу інших.
- Існує можливість динамічно переконфігурувати систему посередництва об’єктів, що мігрують у мережі по запитах. Об’єкти, що надають сервіси, можуть мігрувати на той же процесор, що й об’єкти, які запитують сервіси, тим самим підвищуючи продуктивність системи.

В процесі проектування систем архітектуру розподілених об’єктів можна використовувати подвійно.

1. У вигляді логічної моделі, що дозволяє розробникам структурувати та спланувати систему. У цьому випадку функціональність додатку описується тільки в термінах і комбінаціях сервісів. Потім розробляються способи надання сервісів за допомогою декількох розподілених об’єктів. На цьому рівні, як правило, проектують крупномодульні об’єкти, які надають сервіси, що відбивають специфіку конкретної області додатка. Наприклад, до програми обліку роздрібної торгівлі можна включити об’єкти, які б вели облік стану запасів, відслідковували взаємодію із клієнтами, класифікували товари та ін.

2. Як гнучкий підхід до реалізації систем клієнт-сервер. У цьому випадку логічна модель системи – це модель клієнт-сервер, у якій клієнти та сервери реалізовані як розподілені об’єкти, що взаємодіють за допомогою програмної шини. За такого підходу легко замінити систему, наприклад, дворівневу на багаторівневу. У цьому випадку ні сервер, ні клієнт не можуть бути реалізовані в одному об’єкті, однак можуть складатися з безлічі невеликих об’єктів, кожний з яких надає певний сервіс.

Прикладом системи, якій підходить архітектура розподілених об'єктів, може служити система обробки даних, що зберігаються в різних базах даних (рисунок 6.11). У цьому прикладі будь-яку базу даних можна представити як об’єкт з інтерфейсом, що надає доступ до даних "тільки читання". Кожний з об’єктів-інтеграторів може займатися певними типами залежностей між даними, збираючи інформацію з баз даних, щоб спробувати простежити ці залежності.

Об’єкти-візуалізатори взаємодіють із об’єктами-інтеграторами для представлення даних у графічному виді або для складання звітів за проаналізованими даними.

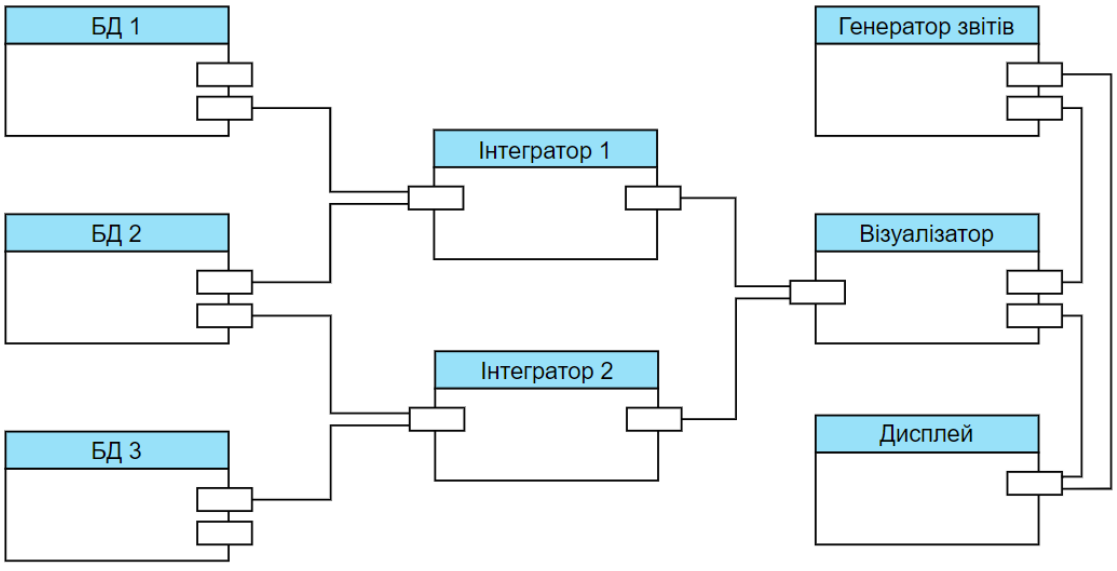

Рисунок 6.11 – Приклад розподіленої системи обробки даних

Для такого типу додатків архітектура розподілених об'єктів підходить більше, ніж архітектура клієнт-сервер, з трьох причин.

1. В цих системах (на відміну, наприклад, від системи банкоматів) немає єдиного постачальника сервісу, на якому були б зосереджені всі сервіси керування даними.

2. Можна збільшувати кількість доступних баз даних, не перериваючи роботу системи, оскільки кожна база даних являє собою просто об'єкт. Ці об'єкти підтримують спрощений інтерфейс, що управляє доступом до даних. Доступні бази даних можна розмістити на різних машинах.

3. За допомогою додавання нових об'єктів-інтеграторів можна відслідковувати нові типи залежностей між даними.

Головним недоліком архітектур розподілених об'єктів є те, що їх складніше проектувати, ніж системи клієнт/сервер. Виявляється, що системи клієнт/сервер надають більш природний підхід до створення розподілених систем.

Хід роботи

1) Спроектувати архітектуру системи згідно варіанту.
2) Оформити звіт (за допомогою UML зобразити архітектуру системи, висновки).

Варіанти завдань

1) Інформаційна система ВНЗ
2) Інформаційна система торгової організації
3) Інформаційна система медицинських організацій міста
4) Інформаційна система автопідприємства міста
5) Інформаційна система проектної організації
6) Інформаційна система авіабудівельного підприємства
7) Інформаційна система військового округу
8) Інформаційна система будівельної організації
9) Інформаційна система бібліотечного фонду міста
10) Інформаційна система спортивних організацій міста
11) Інформаційна система автомобілебудівельного підприємства
12) Інформаційна система готельного підприємства
13) Інформаційна система магазину автозапчастин
14) Інформаційна система аптеки
15) Інформаційна система туристичного клубу.

Перелік літератури

1. Брауде Э. Технология разработки программного обеспечения. – СПб. Питер, 2004. – 655 с.

Додаток А

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЧЕРНІГІВСЬКА ПОЛІТЕХНІКА»

ЗВІТ

про виконання програми проектно-технологічної практики

здобувач вищої освіти _____
(прізвище, ім'я, по батькові)

групи _____

напрям підготовки (спеціальність) _____

спеціалізація _____

кваліфікаційний рівень _____

база практики _____
(повна назва)

Керівник практики
від бази практики

_____
(посада, прізвище, ініціали)

Керівник практики від кафедри

_____
(посада, прізвище, ініціали)