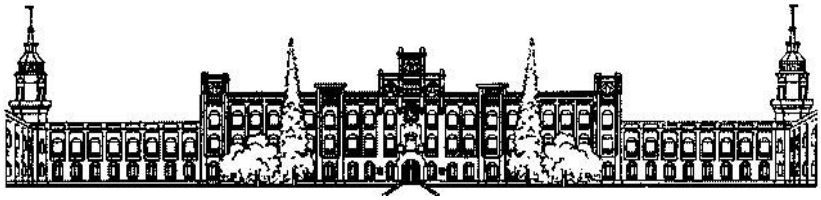

Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Лабораторна робота №9

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Взаємодія компонентів системи.»

Виконав:

студент групи ІА-32

Нагорний Максим

Перевірив:

Мягкий М. Ю.

Київ 2025

Зміст

Вступ

Тема: Вступ до паттернів проектування.

Мета: Вивчити види взаємодії додатків (Client-Server, Peer-to-Peer, Service oriented Architecture), та реалізувати в проєктованій системі одну із архітектур.

Завдання

• Ознайомитись з короткими теоретичними відомостями.

• Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

• Реалізувати функціонал для роботи в розподіленому оточенні відповідно до обраної теми.

• Реалізувати взаємодію розподілених частин:

– Для клієнт-серверних варіантів: реалізація клієнтської і серверної частини додатків, а також загальної частини (middleware); зв'язок клієнтської і серверної частин за допомогою WCF, TcpClient, .NET Remoting на розсуд виконавця.

– Для однорангових мереж: реалізація взаємодії клієнтських додатків за допомогою WCF Peer to peer channel.

– Для SOA додатків: реалізація сервісу, що надає послуги клієнтським застосуванням; викладання сервісу в хмару або підняття у вигляді Web Service на локальній машині; використання токенів для передачі даних про автентифікації, двостороннє шифрування.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє спроєктовану архітектуру. Навести фрагменти програмного коду, які є суттєвими для відображення реалізованої архітектури.

14. **Архіватор** (strategy, adapter, factory method, facade, visitor, p2p)

Архіватор повинен являти собою візуальний додаток з можливістю створення і редагування архівів різного типу (.tar.gz, .zip, .rar, .ace) – додавання/ видалення файлів / папок, редагування метаданих (по можливості), перевірка checksum архівів, тестування архівів на наявність пошкоджень, розбиття архівів на частини.

Теоретичні відомості

Архітектура Client–Server

Архітектура Client–Server є однією з найпоширеніших моделей організації взаємодії між програмними компонентами. У цій моделі система поділяється на два основні типи вузлів: клієнт та сервер. Клієнт відповідає за ініціювання запитів, взаємодію з користувачем та відображення інформації, тоді як сервер забезпечує обробку цих запитів, виконання бізнес-логіки, роботу з базою даних і повернення результату. Сервер є центральним елементом системи, який керує ресурсами та контролює доступ до них. Перевагою цієї моделі є чіткий поділ відповідальності, можливість масштабування сервера, централізований контроль даних та безпеки. Проте система залежить від доступності сервера: якщо він виходить з ладу, клієнти втрачають можливість працювати. Client–Server підходить для вебдодатків, банківських систем, інформаційних порталів та будь-яких сервісів, де потрібен централізований контроль даних.

Архітектура Peer-to-Peer (P2P)

Архітектура Peer-to-Peer передбачає рівноправність усіх вузлів мережі, де кожен учасник може бути і клієнтом, і сервером водночас. У такій системі відсутній центральний сервер, що означає децентралізовану обробку даних і виконання запитів. Кожен вузол має власну частину ресурсів, які він може надавати іншим, і водночас може отримувати ресурси від інших учасників мережі. P2P забезпечує високу стійкість до відмов, оскільки вихід одного з вузлів майже не впливає на загальну функціональність системи. Ця модель широко використовується у файлообмінних мережах, децентралізованих системах зберігання, блокчейні, криптовалютних мережах та в програмах, де важлива розподіленість і відсутність єдиного центру. Однак керувати такою мережею складніше через відсутність централізованого контролера, а питання безпеки та синхронізації даних потребують додаткових механізмів.

Service-Oriented Architecture (SOA)

Service-Oriented Architecture (SOA) — це архітектурний стиль, у якому система складається з набору окремих сервісів, що мають чітко визначені інтерфейси та спілкуються між собою через стандартизовані протоколи. Кожен сервіс є незалежним, виконує певну бізнес-функцію і може бути повторно використаний в інших системах. SOA спрямована на модульність, ізоляцію компонентів та простоту інтеграції. Сервіси можуть бути розміщені на різних серверах, розроблені різними

командами та реалізовані на різних мовах програмування, що забезпечує гнучкість у розробці. Взаємодія між сервісами зазвичай здійснюється через мережеві протоколи, такі як HTTP, SOAP, REST або за допомогою черг повідомлень. Основна перевага SOA — можливість швидко змінювати та розширювати систему, додаючи або замінюючи окремі сервіси без порушення роботи всієї архітектури. Ця модель широко застосовується в корпоративних рішеннях, хмарних сервісах, мікросервісній архітектурі, інтеграційних платформах та великих масштабованих системах.

Хід виконання

Для розробленої системи архівації було реалізовано підтримку однорангової комунікації p2p, що дозволить клієнтам, які взаємодіють з системою отримувати повідомлення про створення архівів в реальному часі. Основна причина вибору такого підходу - це розширення функціональності системи в мережі, будь-який користувач може отримати повідомлення, про те, що інший користувач створив архів. В нашій системі всі вузли рівні між собою.

Під час реалізації підтримки однорангової комунікації було виконано такі дії:

- Створено сервіс P2P, що відповідає за надсилання повідомлень в мережу, в якому визначено IP-адресу та порт для обміну повідомлень;
- Створено клас P2PClient, що відповідає за прослуховування порту та вивід повідомлень;

Переваги обраного шаблону:

- Рівноправність сіх вузлів;
- Швидкий обмін даними
- підтримка SOLID;
- Висока відмовостійкость;

На рис. 1 наведена модифікована діаграма класів із представленням використання шаблону в реалізації системи:

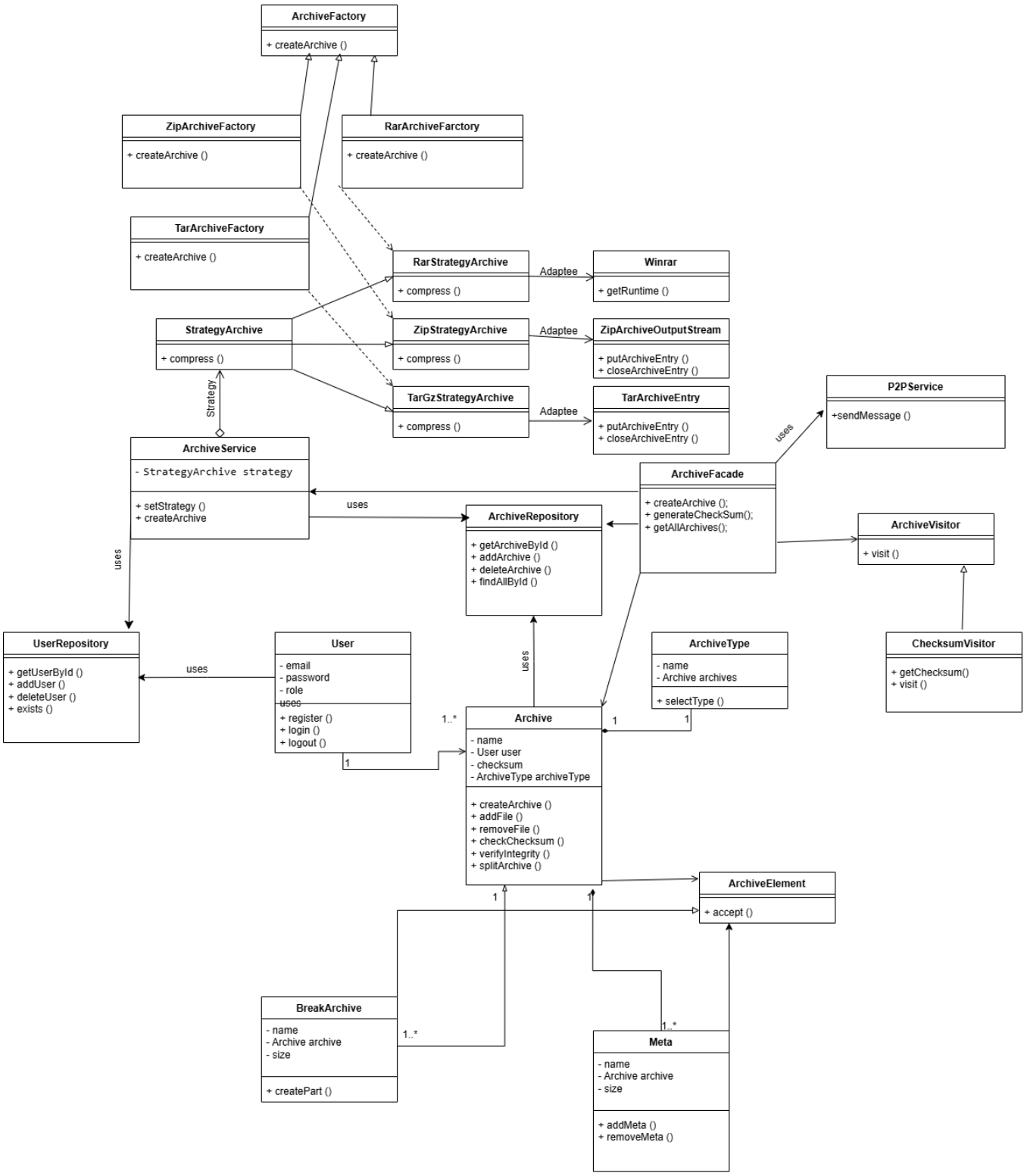

Рис. 1. Діаграма класів

Код програми

package com.example.demo.p2p;

import org.springframework.stereotype.Service;

import java.net.DatagramPacket;

import java.net.InetAddress;

```
import java.net.MulticastSocket;
@Service
public class P2PService {
    private static final String GROUP = "230.0.0.0";
    private static final int PORT = 4446;
    public void sendMessage(String text) {
        try {
            InetAddress group = InetAddress.getByName(GROUP);
            byte[] msg = text.getBytes();
            DatagramPacket packet = new DatagramPacket(msg, msg.length, group, PORT);
            MulticastSocket socket = new MulticastSocket();
            socket.send(packet);
            socket.close();
            System.out.println("P2P sent: " + text);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
package com.example.demo.p2p;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;
public class P2PClient {
    public static void main(String[] args) throws Exception {
        MulticastSocket socket = new MulticastSocket(4446);
        InetAddress group = InetAddress.getByName("230.0.0.0");
        socket.joinGroup(group);
        System.out.println("Start");
        while (true) {
            byte[] buf = new byte[256];
            DatagramPacket packet = new DatagramPacket(buf, buf.length);
            socket.receive(packet);
            String msg = new String(packet.getData(), 0, packet.getLength());
```

```
            System.out.println("Received P2P: " + msg);
        }
    }
}
package com.example.demo.facade;
import com.example.demo.model.Archive;
import com.example.demo.model.Meta;
import com.example.demo.model.User;
import com.example.demo.p2p.P2PService;
import com.example.demo.repository.ArchiveRepository;
import com.example.demo.repository.UserRepository;
import com.example.demo.service.ArchiveService;
import com.example.demo.visitor.ChecksumVisitor;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Component;
import org.springframework.web.multipart.MultipartFile;
import java.io.File;
import java.io.IOException;
import java.util.*;
@Component
public class ArchiveFacade {
    private final ArchiveService archiveService;
    private final ArchiveRepository archiveRepository;
    private final UserRepository userRepository;
    private final P2PService p2pService;
    public ArchiveFacade(
            ArchiveService archiveService,
            ArchiveRepository archiveRepository,
            UserRepository userRepository,
            P2PService p2pService
    ) {
        this.archiveService = archiveService;
        this.archiveRepository = archiveRepository;
```

```
        this.userRepository = userRepository;
        this.p2pService = p2pService;
    }
    public void createArchive(Archive archive, MultipartFile[] files) throws IOException {
        Authentication auth = SecurityContextHolder.getContext().getAuthentication();
        String email = auth.getName(); // email — логін
        User user = userRepository.findByEmail(email);
        archive.setUserId(user.getId());
        List<File> tempFiles = new ArrayList<>();
        Set<Meta> metaSet = new HashSet<>();
        for (MultipartFile multipart : files) {
            if (!multipart.isEmpty()) {
                File temp = File.createTempFile("upload_", "_" + multipart.getOriginalFilename());
                multipart.transferTo(temp);
                tempFiles.add(temp);
                Meta meta = new Meta();
                meta.setFileName(multipart.getOriginalFilename());
                meta.setSize(multipart.getSize());
                metaSet.add(meta);
            }
        }
        archive.setMeta(metaSet);
        ChecksumVisitor v = new ChecksumVisitor();
        archive.accept(v);
        archive.setChecksum(v.getChecksum());
        String outputDir = "C:/archives/";
        new File(outputDir).mkdirs();
        String outputPath = outputDir + archive.getName() + "." + archive.getArchiveTypeId();
        try {
            archiveService.createArchive(tempFiles, outputPath, archive.getArchiveTypeId());
        } finally {
            for (File f : tempFiles) f.delete();
        }
        archive.setBreakArchiveIds(Set.of("part1", "part2"));
```

```
        archiveRepository.save(archive);
        String message = "User " + archive.getUserId() + " created archive: " + archive.getName()
+ "." + archive.getArchiveTypeId();
        p2pService.sendMessage(message);
    }
    public List<Archive> getAllArchives() {
        return archiveRepository.findAll();
    }
}
```

Висновок

Під час виконання лабораторної роботи було реалізовано взаємодію розподілених частин для одноранговых мереж для теми «архіватор», під час реалізації було розроблено сервіс P2P, що відповідає за надсилання повідомлень в мережу, в якому визначено IP-адресу та порт для обміну повідомлень та створено клас P2PClient, що відповідає за прослуховування порту та вивід повідомлень;

Питання до лабораторної роботи

1. Що таке клієнт-серверна архітектура?

Клієнт-серверна архітектура — це модель взаємодії програмних компонентів, у якій система поділяється на дві частини: клієнт, що ініціює запити та взаємодіє з користувачем, і сервер, що обробляє ці запити, виконує бізнес-логіку, працює з даними та повертає результати. Клієнт не зберігає і не обробляє складні дані самостійно, а лише викликає серверні функції, тоді як сервер зосереджує у собі ресурси, логіку та механізми керування. Такий підхід забезпечує централізоване управління даними, спрощує підтримку та дозволяє масштабувати систему за рахунок збільшення потужності сервера або його реплікації.

2. Розкажіть про сервіс-орієнтовану архітектуру.

Сервіс-орієнтована архітектура (SOA) — це підхід до побудови програмних систем, у якому функціональність застосунку розділена на окремі сервіси, що є автономними, незалежними та повторно використовуваними компонентами. Кожен сервіс реалізує певну бізнес-функцію і має чітко визначений інтерфейс для взаємодії з іншими сервісами. SOA передбачає слабке зв’язування компонентів, стандартизовані протоколи взаємодії та можливість інтегрувати сервіси, розроблені різними командами, мовами чи навіть розміщені на різних платформах. Такий підхід значно

підвищує гнучкість, масштабованість і здатність системи до оновлення без порушення її загальної роботи.

3. Якими принципами керується SOA?

Сервіс-орієнтована архітектура ґрунтується на принципах автономності, повторного використання та слабкого зв’язування. Кожен сервіс повинен бути незалежним і самостійно керувати своїм станом, щоб його можна було змінювати або замінювати без впливу на інші частини системи. Сервіси не залежать від конкретної реалізації один одного, а взаємодіють через стандартизовані контракти, що описують структуру запитів і відповідей. Важливим принципом є повторне використання сервісів, який дозволяє уникати дублювання логіки всередині системи. Також SOA передбачає можливість оркестрування сервісів — тобто побудови складних бізнес-процесів шляхом послідовного виклику окремих сервісів.

4. Як між собою взаємодіють сервіси в SOA?

У SOA сервіси взаємодіють між собою через стандартизовані протоколи, які використовують описаний контракт. Найчастіше це HTTP, SOAP або REST-повідомлення, які передаються через мережу. Сервери не викликають внутрішні методи один одного напряму, а спілкуються через чітко визначені зовнішні інтерфейси. Це дозволяє розміщувати сервіси на різних машинах, забезпечувати незалежність їх реалізації та масштабувати кожний компонент окремо. Взаємодія здійснюється у вигляді запитів та відповідей, що робить систему легко інтегрованою з іншими платформами та сервісами.

5. Як розробники взнають про існуючі сервіси і як робити до них запити?

Розробники отримують інформацію про сервіси через їхні специфікації або контракти, які містять опис інтерфейсів, форматів даних, методів виклику та параметрів. Такий контракт може бути представлений у вигляді документації, файлів опису (наприклад, WSDL для SOAP або OpenAPI/Swagger для REST). Після вивчення специфікації розробники можуть виконувати запити до сервісу через HTTP-запити, інструменти тестування API або допоміжні SDK. Наявність чітко визначеного контракту дозволяє викликати сервіс навіть без знання його внутрішньої реалізації.

6. У чому полягають переваги та недоліки клієнт-серверної моделі?

Переваги клієнт-серверної моделі полягають у централізованому управлінні ресурсами, чіткій структурі системи та простоті обслуговування. Сервер зберігає дані та виконує складну логіку, що забезпечує узгодженість інформації та безпеку. Система легко масштабується шляхом модернізації сервера чи розподілу навантаження. Недоліки полягають у можливій перевантаженості або відмові сервера, що робить клієнтів залежними від

його доступності. Також клієнт-серверна модель може створювати “вузьке місце” у випадку великих навантажень, оскільки сервер часто є центральною точкою взаємодії.

7. У чому полягають переваги та недоліки однорангової моделі взаємодії?

Переваги однорангової моделі полягають у повній децентралізації, високій відмовостійкості та можливості рівноправної взаємодії між учасниками мережі. Вихід з ладу одного вузла не порушує загальну роботу системи, а ресурси розподіляються між усіма учасниками. Така модель добре підходить для файлообмінних систем, блокчейн-мереж, чатів і розподілених систем зберігання. Її недоліки полягають у складності забезпечення безпеки, контролю доступу, синхронізації даних і рівномірного розподілу навантаження. Через відсутність централізованого управління координація між вузлами може бути складною і вимагати додаткових протоколів.

8. Що таке мікро-сервісна архітектура?

Мікросервісна архітектура — це підхід, у якому система будується з невеликих незалежних сервісів, кожен з яких виконує одну чітко визначену функцію та може розгортатися, масштабуватися та оновлюватися окремо від інших. Кожен мікросервіс має власну логіку, власні дані, власний інтерфейс для взаємодії з іншими сервісами, і є повністю автономним. Такий підхід дозволяє ефективно масштабувати окремі частини системи, швидко впроваджувати зміни, використовувати різні технології для різних сервісів та забезпечувати високу стабільність завдяки ізоляції компонентів.

9. Які протоколи використовуються для обміну даними в мікросервісній архітектурі?

У мікросервісній архітектурі найчастіше використовуються легковагові протоколи HTTP, REST, gRPC або WebSocket, а також системи передавання повідомлень на основі черг, такі як RabbitMQ, Kafka чи MQTT. Протокол часто обирається відповідно до вимог до швидкодії, затримок, надійності та способу обміну даними. REST використовується для класичних запитів-відповідей, gRPC — для високопродуктивного бінарного зв’язку, а Message Queue — для асинхронної взаємодії між сервісами.

10. Чи можна назвати підхід сервіс-орієнтованою архітектурою, коли ми в проєкті між шаром веб-контролерів та шаром доступу до даних реалізуємо шар бізнес-логіки у вигляді сервісів?

Такий підхід не є сервіс-орієнтованою архітектурою у повному розумінні цього терміна. У цьому випадку ми створюємо внутрішній шар сервісів, але вони не є автономними компонентами, не мають власних контрактів, не розподіляються між серверами і не взаємодіють через стандартизовані мережеві протоколи. Це лише структурна організація коду,

яка покращує логічний поділ системи, але не відповідає принципам SOA. Справжня SOA передбачає незалежні сервіси, здатні працювати окремо один від одного, тоді як у внутрішньому шарі застосунку сервіси залишаються частинами єдиного монолітного процесу.