МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ, КОМП'ЮТЕРНОЇ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ
КАФЕДРА ПРИКЛАДНОЇ ІНФОРМАТИКИ

ДОПУСТИТИ ДО ЗАХИСТУ

Завідувач кафедри

_____ Гамаюн В.П.
(підпис)

"____"_____ 2021р.

ДИПЛОМНИЙ ПРОЕКТ
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ "БАКАЛАВР"

Тема: Модуль візуалізації структур даних

Виконавець: _____ Фісун Дар'я Владиславівна
(підпис) (П.І.Б.)

Керівник: _____ Толстікова Олена Володимирівна
(підпис) (П.І.Б.)

Нормоконтролер: _____ Боровик Володимир Миколайович
(підпис) (П.І.Б.)

Київ 2021

ВСТУП

Щодня організації генерують нові дані. Як результат, кількість необроблених даних різко зросла. Користувачам важко візуалізовувати, досліджувати та використовувати цю величезну кількість даних. Здатність візуалізувати дані має вирішальне значення для наукових та ретроспективних досліджень. Виникла потреба у відображенні великих обсягів даних таким чином, щоб вони були легкодоступними та зрозумілими.

Сьогодні комп’ютери можна використовувати для обробки великих обсягів даних. Візуалізація даних стосується проектування, розробки та застосування згенерованого комп’ютером графічного представлення даних. Це забезпечує ефективне представлення даних що походять з різних джерел, дозволяючи особам, які приймають рішення, бачити аналітику у візуальній формі та полегшує осмислення даних, що допомагає виявляти закономірності, розуміти інформацію та сформувати думку.

Візуалізація є найбільш важливою частиною нашої когнітивної системи. Візуальне уявлення забезпечує найвищу «пропускну здатність» від комп'ютера до людини. Справді, людина отримує більше інформації через зір, ніж через усі інші почуття разом. Приблизно 20 мільярдів нейронів мозку, присвячених аналізу зорової інформації, забезпечують механізм пошуку шаблонів, який є фундаментальним компонентом більшої частини пізнавальної діяльності. Удосконалення когнітивних систем часто означає оптимізацію пошуку даних та полегшення бачення важливих закономірностей. Особа, яка працює з комп’ютерним засобом візуального мислення – це когнітивна система, де критичними компонентами є, з одного боку, людська зорова система, гнучкий шукач шаблонів у поєднанні з адаптивним когнітивним механізмом прийняття рішень, з іншого боку , обчислювальна потужність комп’ютера.

Візуалізація даних – сукупність підходів до застосування графічних принципів для представлення інформації, які допомагають бачити речі, які раніше не були очевидними. Навіть коли обсяги даних дуже великі, закономірності можна помітити легко і швидко.

Розуміння структур даних є справжньою проблемою для розробників в малих і великих компаніях, студентів, тощо. Швидкодія комп’ютерів

дозволяє пріоритезувати швидкість якості при сучасній розробці програмного забезпечення, що призвело до деградації обізнаності у внутрішній реалізації структур даних. Проте, в умовах обмежених обчислювальних ресурсів або при обробці величезних обсягів даних необізнаність у структурах даних одразу спливає назовні і стає вразливим місцем. Тому перетворення та узагальнення всіх даних в одну картину дуже корисно для розуміння та прийняття рішень, що залежать від даних.

Після коректності, друга найважливіша якість кожного алгоритму – його складність. Більш того, часто коректність сама по собі не має значення, нехтуючи складністю, тоді як можливо навпаки: дещо скомпрометувати точність обчислень, щоб отримати значно кращу складність. Складність алгоритму – це залежність кількості операцій, які будуть виконуватися, від розміру вхідних даних. Це має вирішальне значення для масштабованості комп'ютерної системи: може бути легко вирішити проблему програмування для певного набору входів, але як буде поводитися рішення, якщо введення подвоїться або збільшиться збільшився у десять або мільйон разів?

Яскравим прикладом наслідків неправильного вибору структури даних для алгоритму є видалення елементу з динамічного масиву (списку). Алгоритмічна складність вилучення елементу зі списку – O(n), через необхідність зсуву всіх наступних за видаленим елементів. Лінійне зростання часу виконання алгоритму не є допустимим при великій кількості даних, особливо, коли при правильному виборі відповідної структури даних час виконання залишається константою незалежно від кількості елементів.

РОЗДІЛ 1
ОРГАНІЗАЦІЯ СТРУКТУР ДАНИХ ТА ЇХ ВІДОБРАЖЕННЯ

Діаграма структури даних – графічний прийом. Він базується на типі нотації, що стосується класів – зокрема, класів сутностей та класів наборів, які їх пов'язують. Наприклад, окремі люди та автомобілі – це сутності. Взяті разом, вони складають два цілком різні класи сутностей. З іншого боку, всі автомобілі, що належать конкретній особі, складають набір сутностей, що підпорядковуються сутності власника.

1.1.Визначення понять діаграм структур даних

Чотири терміни: сутність, клас сутності, набір сутностей та клас набору є основними для розуміння діаграм структур даних. Термін сутність використовується для позначення конкретного об'єкта, що розглядається. Термін клас сутності представляє групу сутностей, подібних за атрибутами, що їх описують. Може існувати багато різних класів сутностей. Термін набір сутностей означає групування сутностей, які пов’язують групу сутностей одного класу сутності з однією сутністю іншого класу сутності підлеглим зв'язком. Поняття класу сутності і набору сутностей не залежать одне від одного і можуть розглядатися під прямим або ортогональним кутами (рис. 1.1).

Термін клас набору представляє групу наборів сутностей, подібних за атрибутами, що їх описують. Зокрема, це обмежується тими групами наборів, в яких існує однаковий підлеглий зв’язок сутність-до-сутності. Рисунок 1.2 розширює рисунок 1.1, щоб помістити ці чотири терміни у просторовий зв’язок.

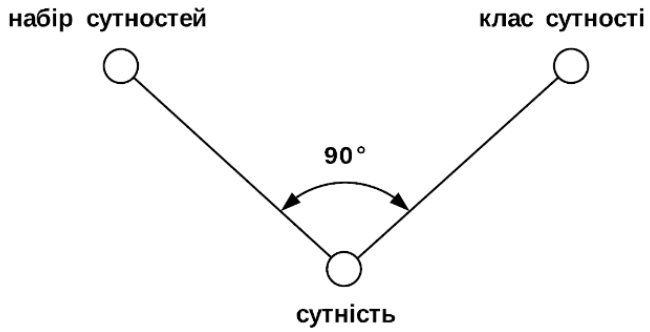

Рис. 1.1. Відношення між термінами набір сутностей і клас сутності

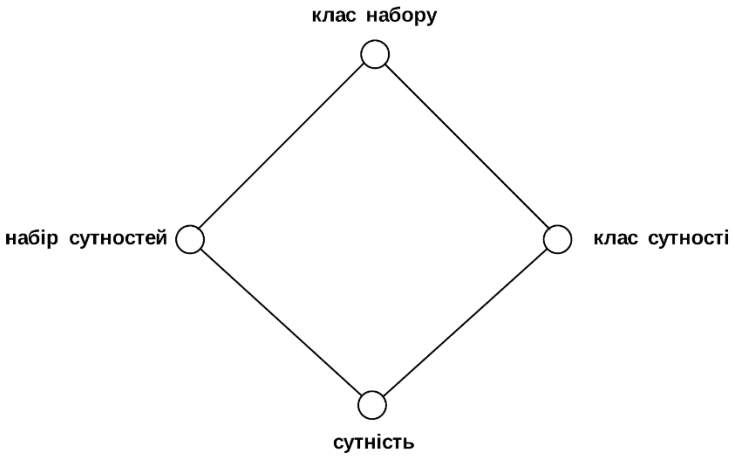

Рис. 1.2. Відношення між основними термінами діаграм структур даних

Може існувати багато різних класів набору. Наприклад, сутності, які можна розглядати в інформаційній системі управління – працівники та підрозділи. Усі співробітники компанії, розглянуті разом, склали б один клас сутності, тоді як усі підрозділи – інший. Хоча працівники та підрозділи можуть розглядатися незалежно одні від одних з певними цілями, зв’язок між групою працівників, які працюють в одному підрозділі, та цим підрозділом може бути дуже важливим. Оскільки підрозділ має набір працівників, які в даний час закріплені за ним, ці працівники можуть законно вважатися підлеглими сутностями цього підрозділу. Кожен підрозділ вважається власником набору, членами якого є його працівники. Набори співробітників, розглянуті спільно, складають набір класу.

Подібним чином, якщо співробітники, як клас сутності, розглядаються з чоловіком або дружиною та дітьми, які складають ще один клас сутності, тоді клас набору може бути встановлений на основі наборів з сутностями працівників, як власників, а їх чоловіка або дружини та дітей, як членів набору. Поняття власника і члена, у відношенні одного власника до багатьох членів, розглянуті на основі поняття класу, є основними для проектування діаграм структур даних.

1.2.Графічні позначення

Методика проектування діаграм структур даних використовує два основні графічні позначення: блок, що представляє клас сутності; і стрілку для представлення класу набору та призначення ролей власника / члена, встановлених цим класом набору. Стрілка вказує від класу сутності, що володіє наборами, до класу сутності, який є членом наборів.

На рисунку 1.3 показана сутність класу. Кількість сутностей, які складають клас сутності, не визначена. Єдиним висновком є оголошення класу сутності.

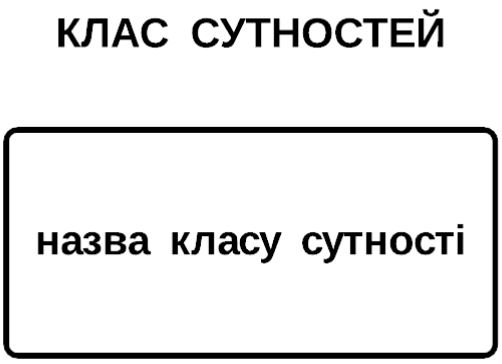

Рис. 1.3. Клас сутностей

На діаграмі (рис. 1.4) показані два класи сутностей: «підрозділ» і «працівник». При розгляді конкретної компанії, було б стільки сутностей підрозділів та співробітників, скільки мала б компанія.

Рис. 1.4. Два класи сутностей

Діаграма на рисунку 1.5 свідчить не тільки про те, що існують два класи сутностей, а і про зв’язок між ними за допомогою класу набору «призначення». Напрямок стрілки читається як: кожен працівник є членом набору працівників, що належать до певного підрозділу, і, далі, кожен підрозділ має такий набір працівників.

Рис. 1.5. Набір зв’язків сутностей

Діаграма структури даних має топологічний характер. Тільки визначення блоків, стрілок та їх назв мають значення. Їх розміри, положення та пропорції обираються для кращої читабельності.

Діаграма структури даних може містити будь-яку кількість блоків та стрілок, необхідних для встановлення конкретних досліджуваних

інформаційних структур. Будь-які два класи сутності можуть мати зв’язок клас сутності / підлеглий клас сутності за нулем, одним, двома або більше класами набору з однаковою або протилежною власністю.

1.3.Ієрархії

Термін ієрархія вживався досить неоднозначно у галузі інформаційних технологій. Діаграми структур даних дають можливість однозначного визначення. Можна сказати, що інформаційна ієрархія існує скрізь, де існує зв’язок набір-клас. Отже, інформаційна ієрархія існує, коли є два або більше рівні пов’язаних класів сутностей. Рисунок 1.6 інтегрує зв’язок підрозділу і працівника, зображеного на рисунку 1.5, зі згаданим раніше зв’язком працівника і підлеглого, для наведення прикладу трирівневої ієрархії.

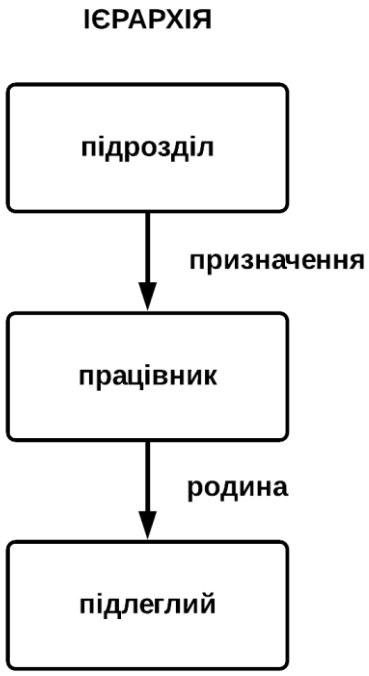

Рис. 1.6. Ієрархія

Багато фактичних структур можуть бути змодельовані як ієрархічні, мережеві або як дерева зв’язків. Коли елементи в ієрархії в реальному світі схожі на сутності і рівень їх звітності може змінюватися, мережа або дерево

зв’язків можуть виявитись більш задовільними при моделюванні ситуації, ніж ієрархія.

1.4.Мережі

Багато інформаційних моделей включають мережі інформації. Діаграми PERT або CPM є прикладами таких мереж. Інший приклад – система подвійного обліку рахунку «Т», в якій кожен запис впливає на дебетову сторону одного рахунку та кредитну сторону іншого.

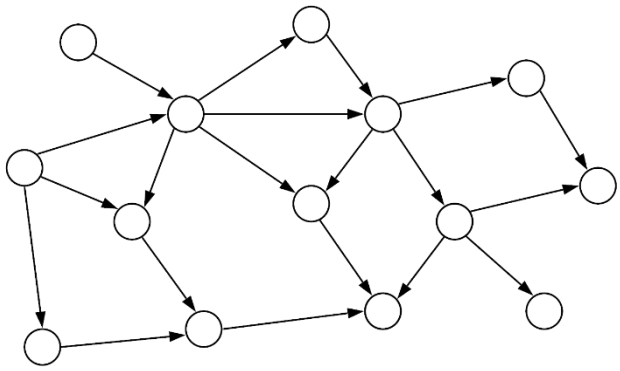

Рис. 1.7. Узагальнене зображення мережі з вузлами, з'єднаних один з одним як спрямований граф

Діаграму структури даних, яка ілюструє мережу, показано на рисунку 1.8.

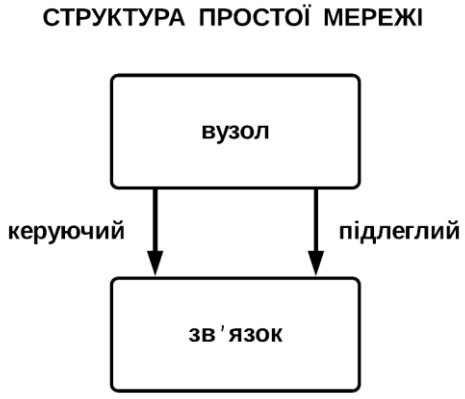

Рис. 1.8. Структура простої мережі

Два класи сутності «вузол» та «зв'язок» відносяться до вузлів і стрілок між вузлами на рисунку 1.7. Склавши таблицю відповідностей (таблиця 1.1), можна швидко визначити декілька різних моделей, орієнтованих на мережу.

Таблиця 1.1

Таблиця відповідностей

Застосування	Назва класу сутності		Назва класу набору	
	Вузол	Зв’язок	Керуючий	Підлеглий
PERT/CPM	Подія	Дія	Попередній	Вдалий
Загальна бухгалтерія	Аккаунт	Транзакція	Дебетовий	Кредитний
Списки деталей	Матеріальний предмет	Компонент	Виклик	Застосування
Генеалогічні схеми	Предмет	Зв’язок	Батько	Дитина
Структура підпрограм	Підпрограма	Виклик	Звернення	Повернення
Схеми організацій	Організація	Компонент	Підрозділ	Доповідь

Подібність діаграм PERT / CPM до мережі очевидна. Подібність загальної бухгалтерської моделі до мережі може бути менш очевидною, але що таке транзакції, якщо не спрямовані кількісні зв’язки між аккаунтами (вузлами); пробний баланс завжди повинен бути нульовим. Списки деталей виробництва складаються з матеріальних предметів, які складаються з матеріальних компонентів. Генеалогічні схеми подібні до списків деталей виробництва, за винятком того, що кожен предмет «виготовляється» лише з двох речей – батька і матері.

Взаємозв’язок набору підпрограм, що викликають одна на одну також є мережею, оскільки кожна підпрограма може викликати багато підпрограм або бути викликаною багатьма підпрограмами.

1.5.Дерева зв’язків

Організаційні схеми, як правило, є особливим випадком мережі – деревом зв’язків, в яких кожен вузол має один підлеглий зв'язок і багато керуючих. Рисунок 1.9 ілюструє дерево.

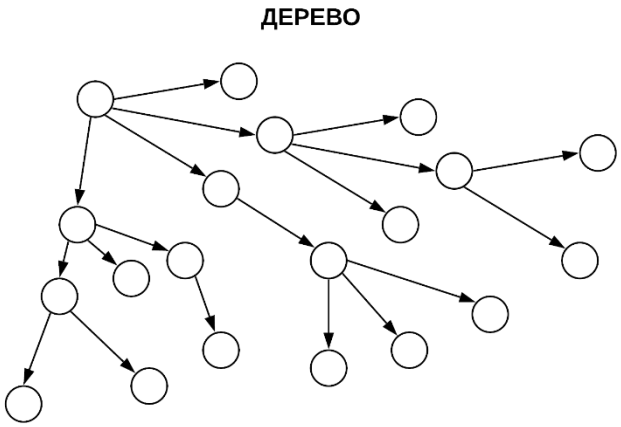

Рис. 1.9. Дерево

Діаграма структури даних на рисунку 1.8 однаково добре підходить для моделювання мережі або дерева. Діаграма структури даних на рисунку 10 більш спеціалізована на моделі дерева, а не мережі.

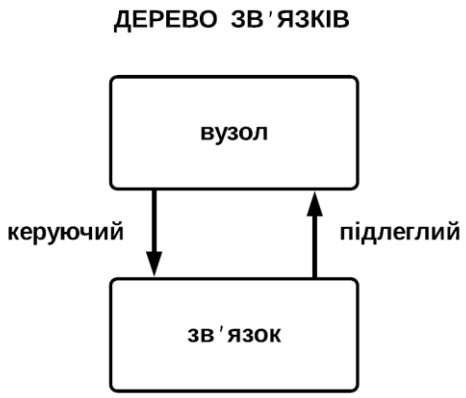

Рис. 1.10. Дерево зв’язків

На діаграмі дерева зв’язок власника та члена (стрілка) «підлеглого» класу набору йде в іншу сторону. Можливість цього основана на факті, того

що дерево допускає лише один зв’язок або їх відсутність на підлеглій стороні вузла. Моделювання дерева за допомогою діаграми структури даних допускає два варіанти: 1) одна сутність зв’язку володіє набором вузлів сутностей з одного члена (рис. 1.10); 2) сутність з одного вузла володіє набором сутностей зв’язків з одного члена (рис. 1.8). Тому спрямованість зв’язків сутностей і підлеглих сутностей є дещо довільною. Діаграму структури даних на рисунку 1.10, спростивши її структуру, можна привести до вигляду діаграми на рисунку 1.11, яка все ще є деревом.

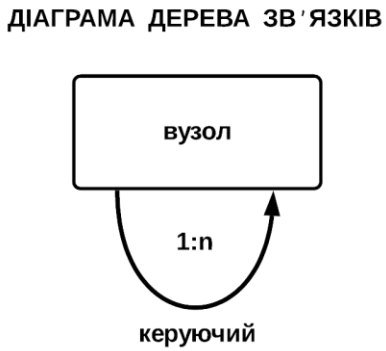

Рис. 1.11 Діаграма дерева зв’язків

«Підлеглий» набір обмежився визначенням дерева, як співвідношення один до одного сутностей «вузол» та «зв’язок». Це був факт, який дозволив змінити спрямованість залежного зв’язку сутності і підлеглої сутності на рисунку 1.10 і далі підтримуючи злиття класу сутності «вузол» з класом сутності «зв’язок». Діаграма просто ілюструє, що кожен «вузол» має «керуючий» зв'язок з іншими вузлами, які, в свою чергу, обмежені одним зв’язком, з їх «підлеглою» стороною.

Діаграми структур даних на рисунках 1.10 та 1.11 створюють ситуацію з куркою та яйцем. Сутність члена не може існувати поки немає набору, до якого її можна віднести. На рисунках 1.10 і 1.11 сутність «вузол» є одночасно власником і членом. Отже, одна така сутність не може існувати незалежно, якщо вона не є своїм власником.

Висновки до розділу

Були визначені діаграми структур даних, які складаються з двох елементів: блоків, які представляють класи сутностей, і стрілок, які представляють класи набору. На прикладах було проілюстровано їх застосування, описано практичне використання при проектуванні та вивченні механізованих інформаційних систем.

Ці фундаментальні структури дозволяють моделювати реляційні або не реляційні структури даних будь-якої складності.

Вище зазначені діаграми структури даних, як вони є, були взяті за стандарт у 60-х роках для уніфікації роботи з даними будь-якою мовою програмування. Внутрішня реалізація цих структур може відрізнятись або бути повністю відсутньою, проте контракт, який реалізують мови програмування, однаковий.

Було прийнято рішення взяти за основу реалізації модуля структуру цих діаграм даних та їх відносин, що дає можливість нативної візуалізації цих блоків.

РОЗДІЛ 2
ВИЗНАЧЕННЯ ВИМОГ ДЛЯ РЕАЛІЗАЦІЇ МОДУЛЯ ВІЗУАЛІЗАЦІЇ ДАНИХ

2.1.Структура функціональних компонентів комп’ютера

Основними функціональними одиницями цифрового комп’ютера є центральний процесор (CPU), система пам’яті, вхідний блок для отримання даних із зовнішнього світу та вихідний блок для надсилання даних, результатів обчислень та команд назад у зовнішній світ. Процесор містить арифметико-логічний пристрій (ALU) та блок управління. Арифметико-логічний пристрій використовується для виконання арифметичних та логічних операцій з даними. Блок управління отримує інструкції програми з пам'яті та видає сигнали управління для інтерпретації та виконання інструкцій, використовуючи доступні апаратні компоненти. Процесор також містить принаймні один, але зазвичай кілька пристроїв пам'яті, які називаються регістрами, якими програміст може маніпулювати як безпосередньо, так і неявно, щоб утримувати дані, поки вони зберігаються в процесорі. Існують також деякі константи та приховані регістри (SPM), які можуть бути використані лише конструктором машини або мікропрограмістом для інтерпретації набору видимих інструкцій програміста на апаратних компонентах.

Коли інструкції програми та дані операндів зберігаються в одній пам'яті, як показано на рисунку 2.1, структура називається архітектурою фон Неймана. Коли інструкції програми зберігаються в пам'яті, окремій від пам'яті, в якій зберігаються операнди даних, як на рисунку 2.2, структура називається архітектурою Гарварда.

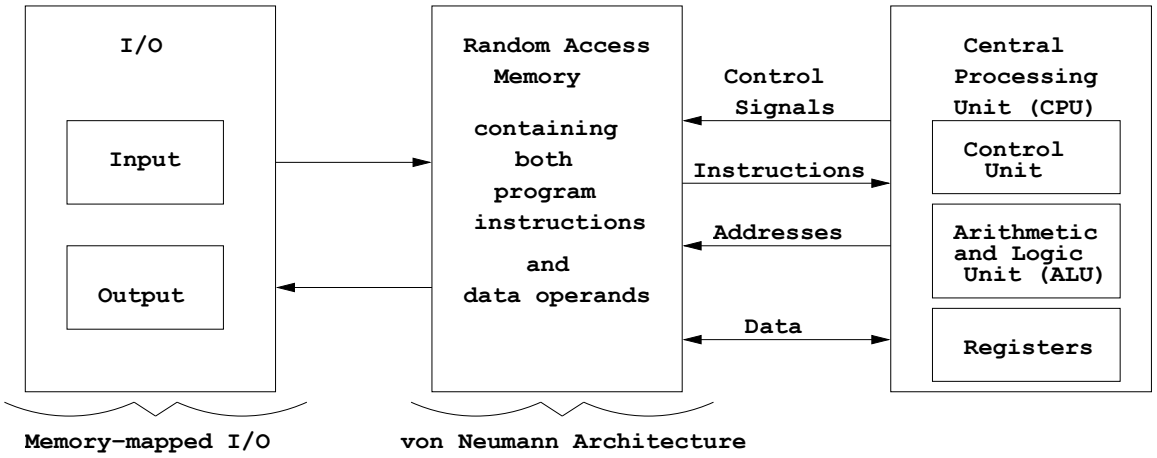

Рис. 2.1. Основні функціональні одиниці, організовані в архітектурі фон Неймана з вводом / виводом через пам'ять

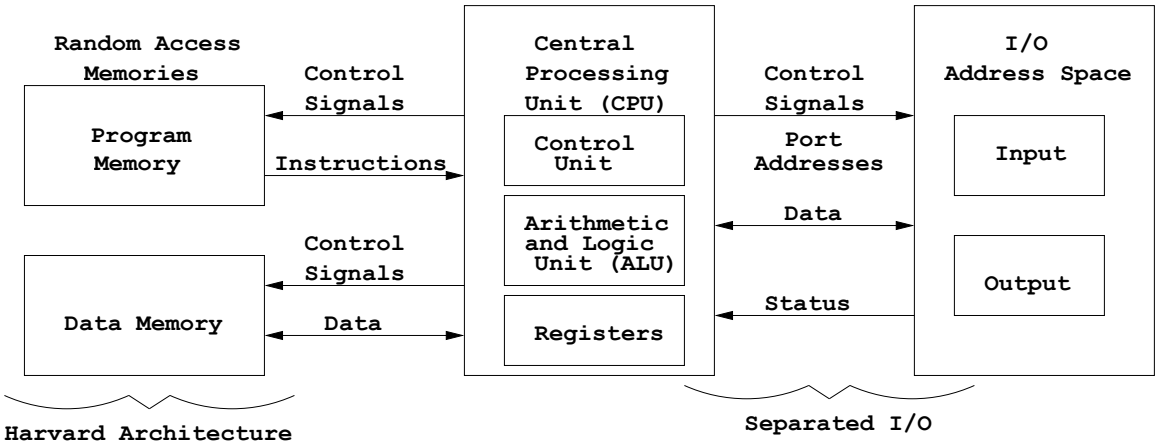

Рис. 2.2. Основні функціональні одиниці, організовані як архітектура Гарварда з відокремленим вводом / виводом

На цих двох малюнках відображено два способи обробки вводу / виводу, а саме – ввід / вивід через пам'ять, та відокремлений ввід / вивід. Ці дві схеми обробки вхідних і вихідних даних на двох малюнках можуть бути змінені місцями, не без зміни підпису. У пристроях з вводом / виводом через пам'ять, регістри є частиною (і, отже, споживають частину) простору адреси пам'яті. Регістри вводу / виводу даних пристроїв, статусу та командні маніпулюються шляхом отримання та виконання процесором інструкцій завантаження, зберігання та переміщення. В архітектурі з відокремленим вводом / виводом, регістри вводу / виводу даних пристроїв, статусу та командні утворюють власний адресний простір, окремо від простору

адресної пам'яті, і процесор повинен мати інструкції вводу / виводу, які (при їх виконанні) змушують реагувати порти пристрою вводу / виводу.

2.2.Регістри і пам’ять

Регістр – пристрій для запису та запам'ятовування деякої закодованої величини. Стрілка на шкалі на 10 поділів в одометрі записує і запам'ятовує рядок десяткових цифр кінцевої довжини, що відображає кількість кілометрів, яку проїхав автомобіль, де переведення на кількість пройдених кілометрів досягається зміною відношень між шестернями, що підраховують оберти трансмісії вихідного валу. У більшості таких автомобілів, коли одометр реєструє 99999,9 кілометрів, а стрілка на шкалі повертається додатково на 0,1 кілометра, одометр показує 00000,0 кілометрів. Іншими словами, регістри кінцевої довжини забезпечують модульну арифметику, де в цьому випадку кратні 100000,0 кілометрам еквівалентні нулю кілометрів, тому що немає шкали, щоб утримувати 1, що записує лівий кінець регістра. Те саме стосується n-бітових регістрів, що використовуються в цифрових комп'ютерах; один n-бітовий регістр може представляти рівно 2^n величин, кодованих двійковими числами в діапазоні $\{0, ..., 2^n - 1\}$, незалежно від того, що представляють ці числа.

Пам'ять можна розглядати як сукупність n-бітових регістрів, кожен з яких має унікальну k-бітову двійкову адресу, подібну до лінійного масиву поштових скриньок. Кожна поштова скринька позначена номером, який є абсолютною адресою цієї скриньки. Вміст номера поштової скриньки є n-бітовим номером, написаним на конверті всередині поштової скриньки. Мешканці квартир зазвичай пишуть своє ім’я на дверцятах поштової скриньки. Це є прикладом прив’язки символьної адреси, представленої іменем особи, до абсолютної адреси, відомої як номер поштової скриньки. При складанні комп'ютерної програми для зберігання в пам'яті кожна символьна (змінна) назва або мітка є символьною адресою, яка повинна бути прив'язана до абсолютної адреси пам'яті. Ця прив'язка адреси зазвичай обробляється програмою, яка називається асемблер, або іншою програмою – редактором посилань.

Бажано мати пам'ять, побудовану таким чином, щоб вона займала однакову кількість часу для отримання доступу до будь-якої випадково обраної адреси. Ця пам’ять називається оперативною, на відміну від пам'яті з послідовним доступом, розміщеної на магнітній стрічці. Очевидно, що читання стрічки займає більше часу для пошуку даних, розташованих в кінці стрічки, ніж на початку. Пристрої з послідовний доступом, такі як стрічки, пристрої з напіввипадковим доступом, такі як магнітні диски та інші пристрої подібним доступом трактуються швидше як пристрої вводу / виводу, ніж пам'ять.

2.3.JVM як середовище виконання

Кожного разу при запуску програми, написаної однією з мов програмування JVM, програма фактично запускається в межах екземпляра JVM, і кожна окрема програма, матиме власний екземпляр. Ці мови використовують інтерпретовану форму вихідного коду, що має назву байт-код. Але як інструкції, написані мовами програмування JVM, транслюються в інструкції, зрозумілі ОС?

Специфікація JVM визначає абстрактну внутрішню архітектуру для цього процесу. Файли класу (скомпільовані файли мають розширення .class і називаються файлами класів) завантажуються в JVM, де вони виконуються механізмом виконання. При виконанні байт-коду, JVM взаємодіє з ОС за допомогою власних методів, і реалізації тих власних методів, які пов’язують певну реалізацію JVM з певною платформою (рис. 2.3).

На додаток до попередніх компонентів, JVM також потребує пам'ять для зберігання тимчасових даних, пов'язаних з виконанням коду, таких як локальні змінні, який метод виконується, і так далі. Ці дані зберігаються в областях даних середовища виконання JVM.

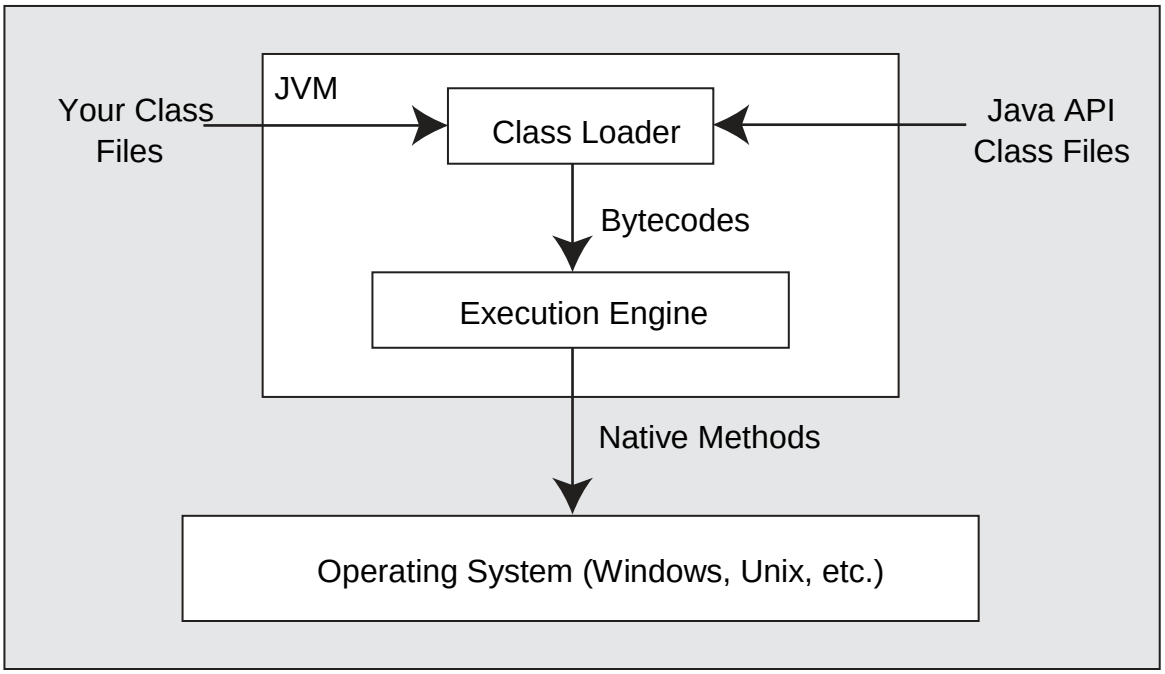

Рис. 2.3. Роль JVM

Хоча окремі реалізації можуть дещо відрізнятися від платформи до платформи, кожна JVM повинна забезпечити компоненти виконання, показані на рисунку 2.4.

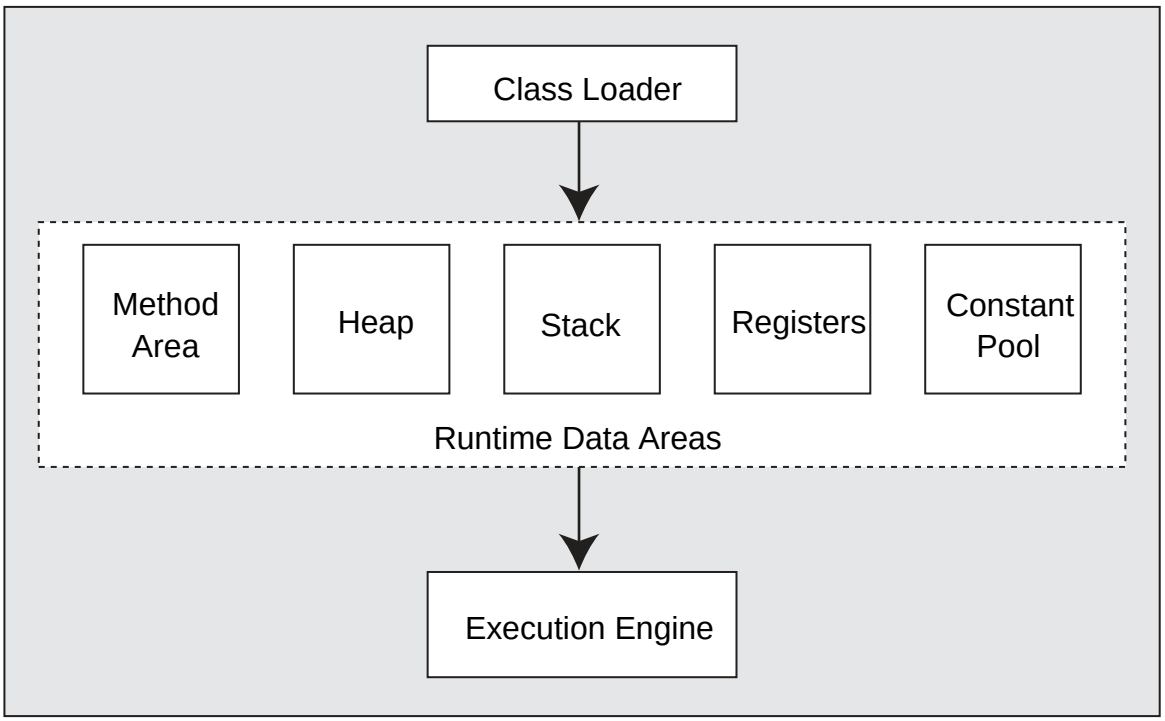

Рис. 2.4. Область виконання даних

2.3.1. Купа

Купа – це область вільної пам'яті, яка часто використовується для динамічного або тимчасового розподілу пам'яті. Купа – область даних виконання, яка забезпечує пам’ять для об’єктів класу та масиву. Коли створюються об'єкти класу або масиву, потрібна їм пам'ять виділяється з купи, яка створюється при запуску JVM. Пам'ять купи відновлюється, коли посилання на об'єкт або масив більше не існують, за допомогою автоматичної системи управління сховищем, відомої як збирання сміття.

Специфікація JVM не визначає деталі реалізації купи; це залишається за творчістю окремих реалізацій JVM. Розмір купи може бути постійним, або може збільшуватись та зменшуватися за потреби. Програмісту може бути дозволено вказувати початковий розмір купи.

2.3.2. Стек

Стековий кадр зберігає стан викликів методів. Стековий кадр зберігає дані, часткові результати та включає середовище виконання методу, будь-які локальні змінні, що використовуються для виклику методу та стек операндів методу. Стек операндів зберігає параметри і повертає значення для більшості інструкцій байт-коду. Середовище виконання містить вказівники на різні аспекти виклику методу.

Кадри - це компоненти, що складають стек JVM. Вони зберігають часткові результати та дані, і повертають значення для методів. Вони також виконують динамічне зв'язування та видають винятки під час виконання. Кадр створюється, коли метод викликається,та знищується, коли виконання методу завершується з будь-якої причини. Кадр складається з масиву локальних змінних, стеку операндів та посилання на пул констант виконання класу поточного методу.

Коли JVM запускає код, лише один кадр, що відповідає виконуваному в даний час методу є активним. Він називається поточний кадр. Метод, який він представляє, є поточним методом, а клас, що включає цей метод, є поточним класом. Коли потік викликає метод (кожен потік має власний стек),

JVM створює новий кадр, який стає поточним кадром, і штовхає його в стек для цього потоку.

Як і у випадку з купою, специфікація JVM залишає реалізацію фреймів стеку до конкретної реалізації JVM. Стеки можуть бути як фіксованого розміру, так і розширюватись або скорочуватись за потреби. Програмісту може бути надано контроль над початковим розміром стека та його максимальними та мінімальними розмірами.

2.3.3. Регістри

Регістри, які підтримуються JVM, подібні до регістрів в інших комп'ютерних системах. Вони відображають поточний стан машини і оновлюються під час виконання байт-коду. Первинний регістр – це програмний лічильник (лічильник команд), який вказує адресу інструкції JVM, яка виконується в поточний момент. Якщо метод, який виконується в поточний момент, є нативним (написаний не мовою JVM), значення лічильнику команд не визначено. Інші регістри в JVM включають вказівник на середовище виконання поточного методу, вказівник на першу локальну змінну поточного виконуваного методу та вказівник на верх стека операндів.

2.4.Порівняння мов програмування JVM Scala та Java

Java є об’єктно-орієнтованою імперативною мовою програмування, тоді як Scala припускає, що жодна з парадигм не підходить для вирішення всіх можливих проблем на практиці, тому поєднує функціональний та імперативний стилі, щоб розробники не мали робити ексклюзивний вибір.

Завершення проекту у Scala вимагає більше зусиль, ніж у Java. Рисунок 2.5 ілюструє зведену статистику. Медіана зусиль становить 56 годин для Scala та 43 години для Java (різниця в 13 годин). Середні зусилля складають 72 години для Scala та 52 години для Java, що означає, що в середньому завершення проекту Scala займає на 20 годин (38%) більше.

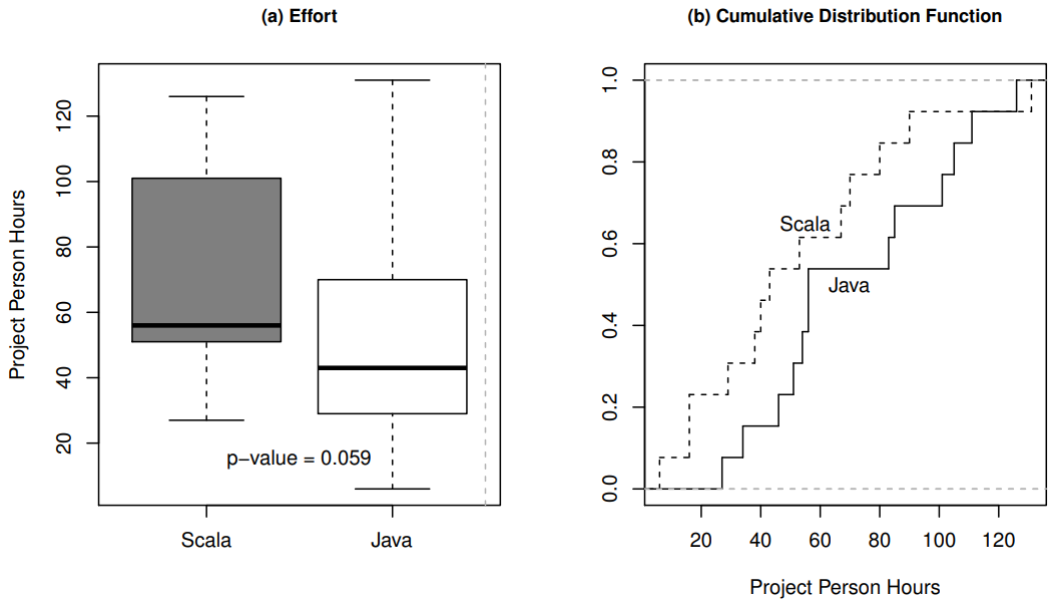

Рис. 2.5. Зведена статистика зусиль

Рисунок 2.6 показує, скільки часу витрачається на імплементацію, при роботі переважно з послідовним або паралельним кодом. Найсуттєвіша різниця обумовлена зусиллями, витраченими на тестування та налагодження: медіана для Scala становить 20 годин (середнє значення 23 години), медіана для Java – 10 годин (в середньому 14).

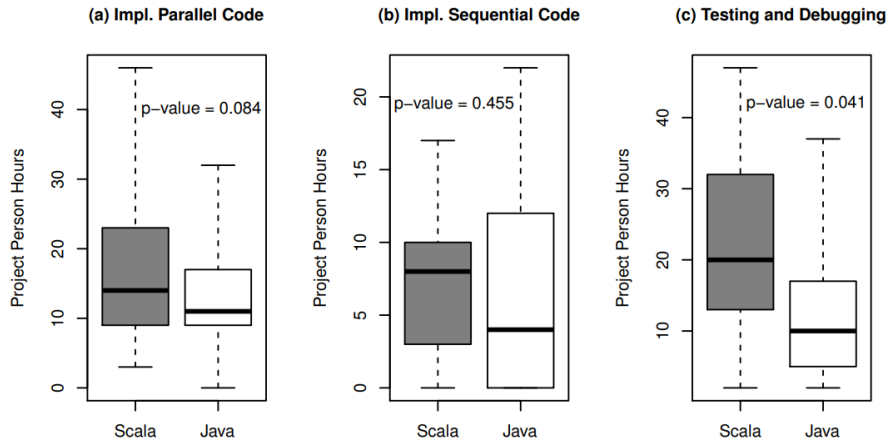

Рис. 2.6. Зусилля, витрачені на всі паралельні проекти, поділені на реалізацію послідовного коду, паралельного коду, тестування та налагодження

Рисунок 2.7 підсумовує кількість рядків коду (LOC) програм Scala та Java, а також кількість символів, за винятком коментарів та порожніх рядків. Програма Scala в середньому займає 536 ліній коду, а Java – 632, але довжина рядків Scala коротша. Спарений тест суми рангу Уілкоксона для рішення кожного суб'єкта показує підтримку (p = 0,078) більшої компактності коду Scala.

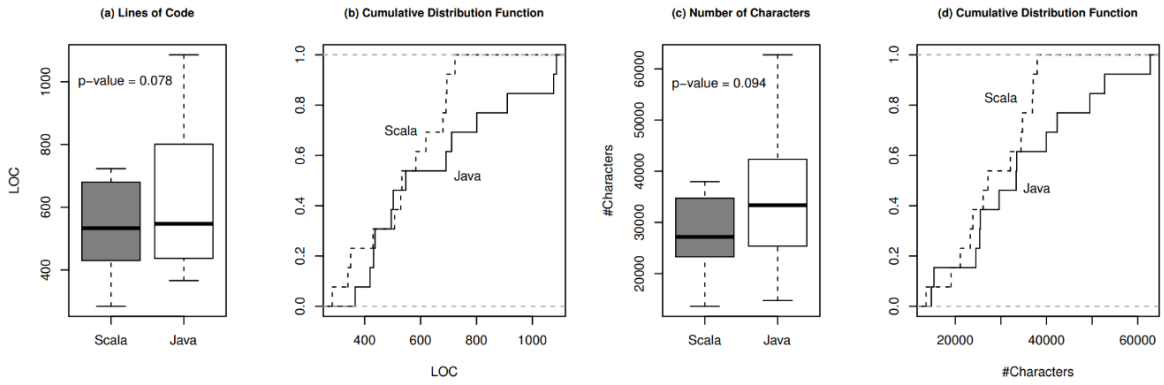

Рис. 2.7. Аналіз компактності коду Scala та Java

Порівнюючи продуктивність, вимірювання на рисунку 2.8 показують, що програми Scala, що виконуються в одному потоці, зазвичай швидші, ніж програми Java (середня різниця складає 82 с. на 4-ядерному процесорі та 190 с. на 32-ядерному процесорі). Реалізація стека в Java (java.util.stack) є основною причиною виникнення проблем із продуктивністю, яких не існує в Scala.

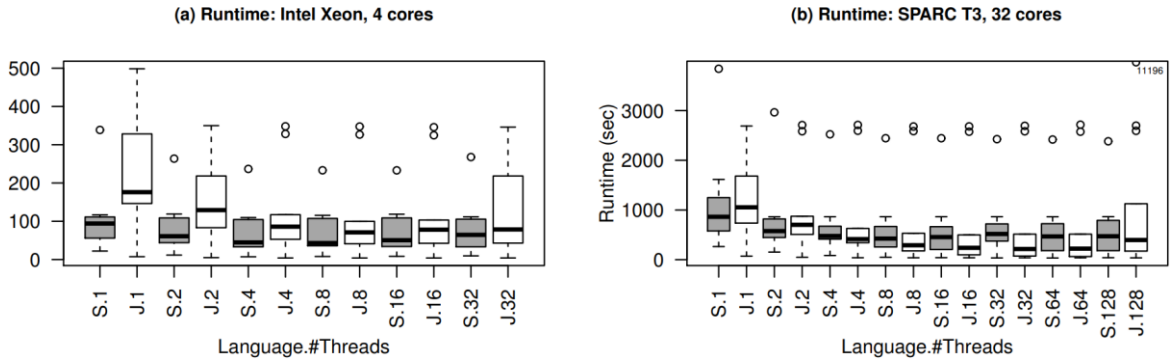

Рис 2.8. Огляд продуктивності всіх програм Scala та Java для різної кількості потоків

Найкращі 3 результати на 4-ядерному процесорі дало використання 47%, 55% і 38% функціонального стилю у програмах. На 32-ядерному процесорі найкращі 3 результати дало використання 47%, 55% і 64% функціонального стилю. Використання 2% функціонального стилю (тобто 98% імперативного), у програмі Scala, показав найгірший результат на обох процесорах. Ці емпіричні результати суперечать поширеній думці, що функціональний стиль програмування шкодить продуктивності.

Висновки до розділу

У розділі було розглянуто структуру комп'ютерної архітектури, принцип роботи регістрів та пам'яті, JVM, як середовище виконання та порівняні мови програмування, для подальшого вибору найбільш підходящої для реалізації проекту.

Багатоядерне обладнання вже використовується всюди, і розробка програмного забезпечення повинна підлаштовуватись під розвиток апаратного забезпечення. Мови з багатьма парадигмами, такі як Scala обіцяють полегшити проблеми паралельного програмування, з якими стикаються розробники, поєднавши функціональний та імперативний стилі програмування. Результати порівняння Scala та Java показують, що код Scala справді є більш компактним, ніж код Java. Продуктивність програми Scala також на одному рівні з Java. Результати порівняння також спрощують поширену думку про шкоду для продуктивності від використання функціонального стилю програмування. Найкращі результат дає написання близько половини програми у функціональному стилі, а другої половини в імперативному. Однак результати також свідчать про необхідність поєднання функціонального стилю та імперативного стилю, оскільки найкращий результат не досягався виключно завдяки використанню функціонального стилю. Щодо зусиль, огляд мов спростовує твердження про те, що програми Scala розробляються швидше: у порівнянні з Java, Scala вимагає більше зусиль, і особливо більше часу для тестування та налагодження.

Підсумовуючи виконану роботу над дослідженням переваг мов JVM Scala та Java, було прийнято рішення використати Scala для розробки проекту.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ МОДУЛЯ ВІЗУАЛІЗАЦІЇ СТРУКТУР ДАНИХ

Існує ряд речей, необхідних для розробки мовою Scala. Перш за все потрібен доступ до компілятора Scala. Компілятор має назву scalac, і може використовуватись з командного рядка для компіляції файлів безпосередньо, або через IDE, яке компілює код автоматично. Його також можна використовувати за допомогою різних інструментів збірки, таких як SBT, який буде описаний у цьому розділі.

Для розробки, компіляції, тестування та запуску програм Scala також необхідне середовище Scala. Оскільки Scala є мовою JVM, це означає, що також необхідне середовище Java. Теоретично, можна встановити кожен із цих компонентів самостійно та використовувати будь-який редактор. Однак найпростіший спосіб розпочати роботу із Scala – встановити одну з Scala IDE.

IntelliJ IDE забезпечує повну підтримку JVM мов програмування, у тому числі Scala. Для підтримки Scala в Community Edition необхідно додатково встановити Scala плагін.

3.1.Створення, реалізація і використання проекту

Проект та середовище Scala, яке буде використовуватися з проектом, вказується при створенні нового проекту в меню File → New → Projects. Ця дія призведе до відкриття діалогового вікна, що проведе через необхідні для створення нового Scala проекту кроки, як показано на рисунку нижче.

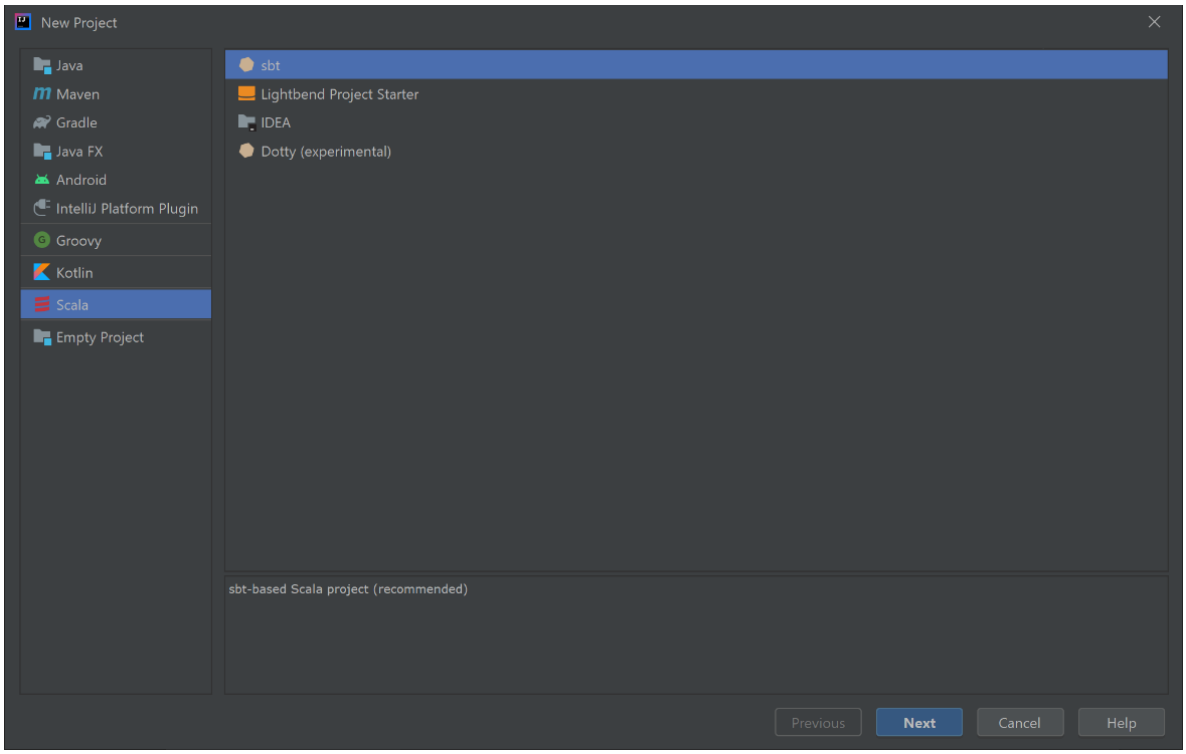

Рис. 3.1. Створення нового Scala проекту на основі SBT

Структура вже готового проекту складається з модуля, який містить в собі пакети. Приховані пакети .bsp та .idea містять файли конфігурацій для локальної машини. Вони генеруються автоматично при завантаженні проекту з системи контроля версіями Git або при створенні нового проекту.

В пакеті core знаходиться безпосередньо реалізація бібліотеки. В demo – демо-проект, розроблений як приклад для ознайомлення з використанням бібліотеки. В images зберігається результат виконання програми, куди поміщаються згенеровані зображення.

target – каталог збірки Maven. Maven це програмний засіб управління проектами. Весь згенерований під час компіляції проекту код буде розміщений у цій теці.

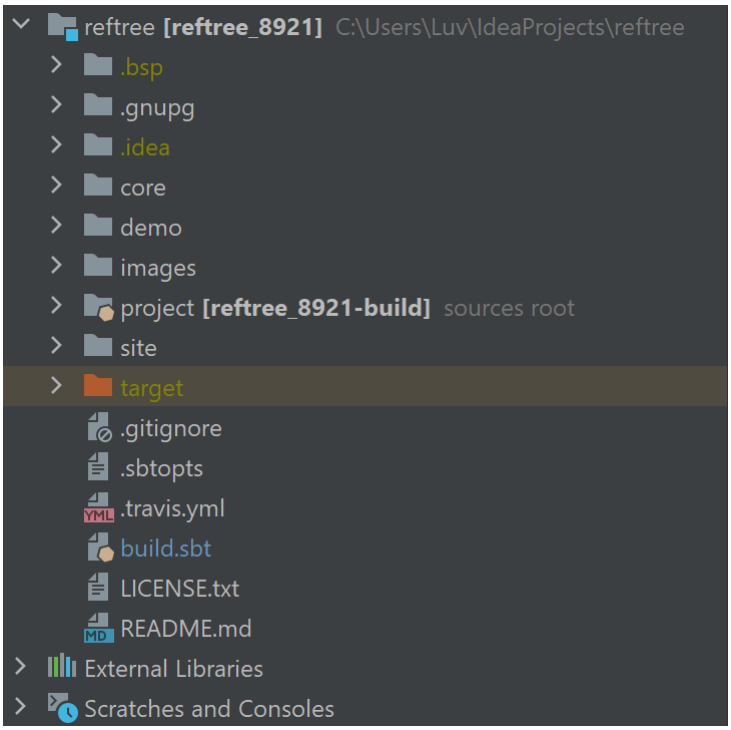

Рис. 3.2. Структура проекту

Нижче будуть описані принципи роботи і використання бібліотеки.

3.1.1. Дерева

Бібліотека відображає діаграми на основі простого подання даних, яке називається RefTree. По суті, RefTree позначає або об'єкт (AnyRef) з кількістю полів, або примітиви (AnyVal).

Щоб візуалізувати значення типу A, знадобиться неявний екземпляр ToRefTree [A]. Для багатьох колекцій Scala, а також case класів не потрібна додаткова робота, оскільки ці екземпляри легко доступні або генеруються під час виконання програми.

Можна налаштувати автоматично згенеровані екземпляри таким чином:

```
import reftree.core.ToRefTree

case class Tree(size: Int, value: Int, children: List[Tree])

implicit val treeDerivationConfig = (ToRefTree.DerivationConfig[Tree]
.rename("MyTree")                                  // display as “MyTree”
.tweakField("size", _.withName("s"))               // label the field “s”,
instead of “size”
.tweakField("value", _.withTreeHighlight(true))    // highlight the value
.tweakField("children", _.withoutName))            // do not label the
“children” field
```

```
implicitly[ToRefTree[Tree]] // auto-derivation will use the configuration
above
```

Для більш специфічний реалізацій, необхідно наслідувати вручну:

```
import reftree.core._

implicit def treeInstance: ToRefTree[Tree] = ToRefTree[Tree] { tree =>
  RefTree.Ref(tree, Seq(
  // display the size as a hexadecimal number (why not?)
  RefTree.Val(tree.size).withHint(RefTree.Val.Hex).toField.withName("s"),
  // highlight the value
  tree.value.refTree.withHighlight(true).toField.withName("value"),
  // do not label the children
  tree.children.refTree.toField
  )).rename("MyTree") // change the name displayed for the class
}
```

3.1.2. Візуалізатори

Для візуалізації діаграм та анімацій призначений клас Renderer:

```
case class Renderer(
  renderingOptions: RenderingOptions = RenderingOptions(),
  animationOptions: AnimationOptions = AnimationOptions(),
  directory: Path = Paths.get("."),
  format: String = "png"
)
```

Приклад використання класу для JVM:

```
import reftree.render._
import reftree.diagram._
import java.nio.file.Paths

val renderer = Renderer(
renderingOptions = RenderingOptions(density = 75),
directory = Paths.get(ImagePath, "guide")
)
```

Приклад використання класу для Scala.js:

```
import reftree.render._
import reftree.diagram._

val renderer = Renderer(
renderingOptions = RenderingOptions(density = 75)
)
```

Також можна передавати параметр format як рядок до конструктора класу Render, щоб вказати необхідний формат зображення. Формат за

замовчуванням, як видно з коду класу, PNG. Замість нього може бути переданий будь-який формат, який підтримується dot -T.

3.1.3. Діаграми

Візуалізація діаграм імплементована за допомогою явного класу DiagramRenderSyntax. Ключове слово implicit робить основний конструктор класу доступним для неявних перетворень, коли клас доступний в контексті.

```
implicit class DiagramRenderSyntax(diagram: Diagram) {
  def render(
    name: String,
    tweak: RenderingOptions ⇒ RenderingOptions = identity
  ) = self
    .tweakRendering(tweak)
    .render(name, diagram)
}
```

Діаграми можна створити та об’єднати у більші діаграми за допомогою відповідного API:

```
// no caption
Diagram(Queue(1)).render("caption-none")
```

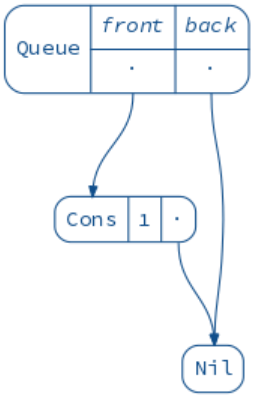

Рис. 3.3. Діаграма черги з одним елементом

```
// automatically set caption to "Queue(1) :+ 2"
Diagram.sourceCodeCaption(Queue(1) :+ 2).render("caption-source")
```

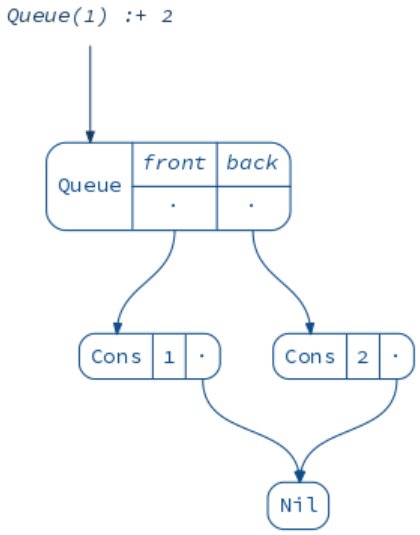

Рис. 3.4. Діаграма черги з двох елементів (додання другого елементу)

```
// use toString to get the caption, i.e. "Queue(1, 2)"
Diagram.toStringCaption(Queue(1) :+ 2).render("caption-tostring")
```

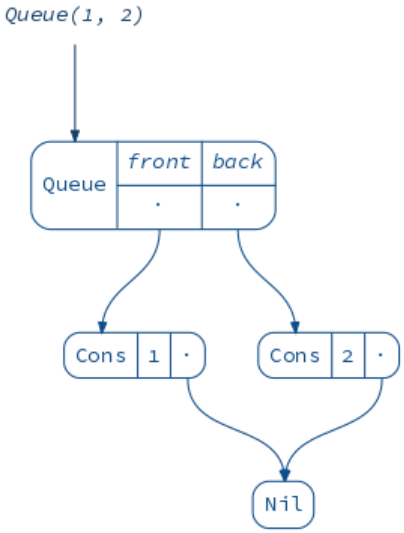

Рис. 3.5. Діаграма черги з двох елементів

```
// merge two diagrams, set captions manually
(Diagram(Queue(1)).withCaption("one") +
Diagram(Queue(2)).withCaption("two")).render("one-two")
```

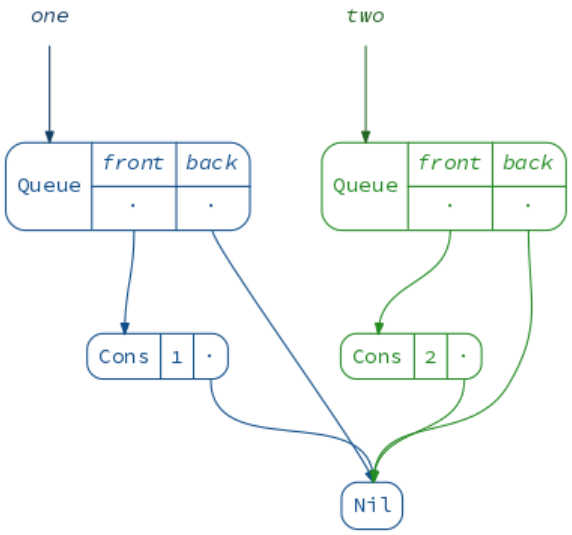

Рис. 3.6. Об’єднані діаграми

```
// isolate each diagram in its own namespace (graph nodes will not be shared
across them)
(Diagram(Queue(1)).toNamespace("one") +
Diagram(Queue(2)).toNamespace("two")).render("namespaced")
```

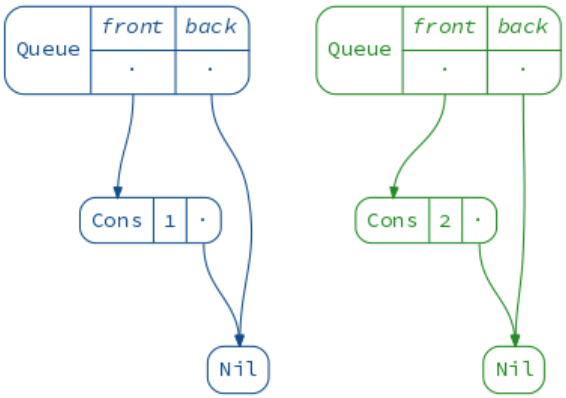

Рис. 3.7. Розділені діаграми

3.1.4. Анімації

Анімація – це послідовність діаграм, яка може бути відтворена в анімованому форматі GIF. Найпростіший спосіб створення анімації – використання builder API:

```
(Animation
.startWith(Queue(1))
.iterateWithIndex(2)((queue, i) ⇒ queue :+ (i + 1))
.build()
.render("animation-simple"))
```

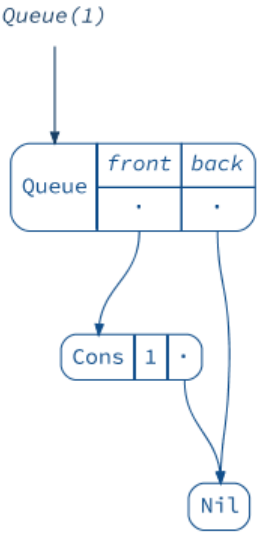

Рис. 3.8. Послідовне додання елементів до черги

Також можна налаштувати спосіб створення діаграми для кожного окремого кадру.

Варто звернути увагу, що бібліотека реалізована таким чином, що за замовчуванням вона намагатиметься зменшити кількість середнього переміщення всіх вузлів дерев через кадри анімації. Іноді замість цього потрібно «закріпити» корінь структури даних, щоб змусити його залишатися нерухомим, поки все інше рухається. Це досягається за допомогою методу withAnchor:

```
(Animation
.startWith(Queue(1))
.iterateWithIndex(2)((queue, i) ⇒ queue :+ (i + 1))
.build(Diagram(_).withAnchor("queue").withCaption("This node is anchored!"))
.render("animation-anchored"))
```

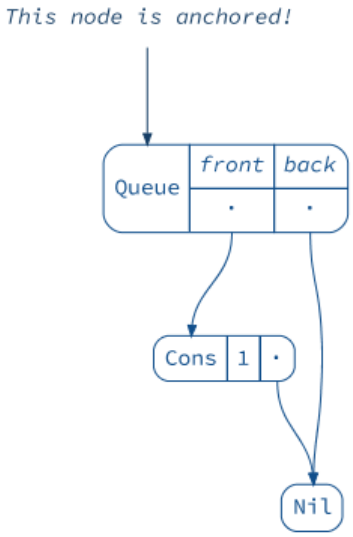

Рис. 3.9. Діаграма з закріпленим блоком

Нарешті, анімацію можна комбінувати послідовно або паралельно, наприклад таким чином:

```
val queue1 = (Animation
.startWith(Queue(1))
.iterateWithIndex(2)((queue, i) ⇒ queue :+ (i + 1))
.build()
.toNamespace("one"))

val queue2 = (Animation
.startWith(Queue(10))
.iterateWithIndex(2)((queue, i) ⇒ queue :+ (10 * (i + 1)))
.build()
.toNamespace("two"))

(queue1 + queue2).render("animation-parallel")
```

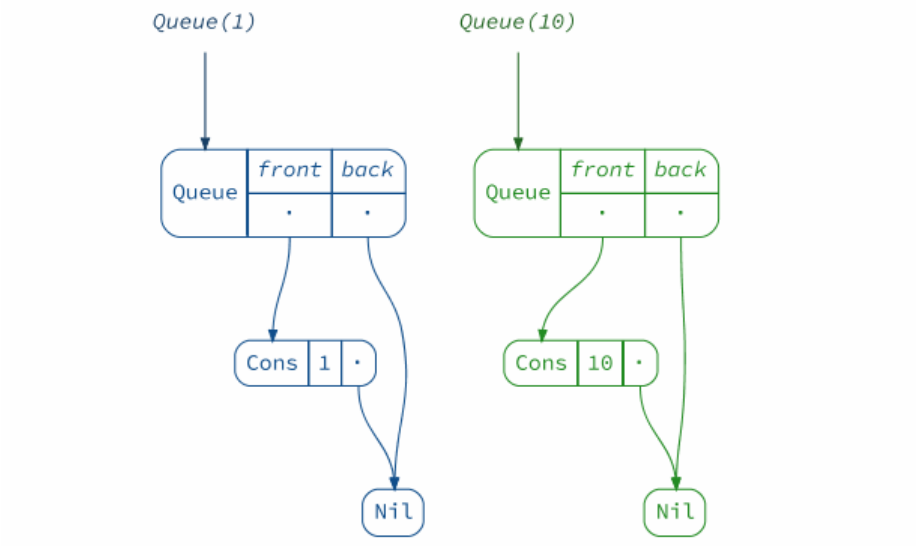

Рис. 3.10. Паралельне відображення

3.2.Інструмент збірки SBT

Для збірки і запуску проекту використовується інструмент збірки SBT.

Simple Build Tool (SBT) написаний на мові Scala і може використовуватися для компіляції як Java так і Scala коду. SBT є не просто інструментом збірки. Він також забезпечує базовий фреймворк для середовища розробки. Вбудована консоль Scala, механізм безперервної компіляції та швидкий сервер компіляції роблять його необхідним інструментом у будь-якому наборі інструментів Scala розробника.

Крім того, що SBT є інструментом збірки, який обирається для більшості проектів Scala, SBT пропонує деякі цікаві функції, крім тих, що надаються звичайним інструментом збірки. Для початку, SBT можна налаштувати за допомогою простого DSL на основі Scala та розширено для використання повноцінної конфігурації зі Scala, як того вимагає проект.

SBT використовує покрокову рекомпіляцію, щоб зменшити час компіляції Scala коду. Покроковий компілятор буде компілювати лише те, що потрібно перекомпілювати після внесення змін. Він також підтримує роботу JVM компілятора і працює разом зі скомпільованим проектом код, щоб запобігти часу завантаження.

Команди SBT можна запускати в активному режимі виконання. Тобто, при виборі користувачем, конкретні завдання будуть виконуватися щоразу, коли користувач вносить зміни до будь-якого вихідного коду. Спочатку це було призначено для безперервної компіляції, але зараз розширено для інших завдань. Також є можливість запуску кількох команд у цьому режимі.

SBT забезпечує підтримку тестування за допомогою ScalaCheck, специфікацій та ScalaTest. Крім того, тести JUnit також можна запускати за допомогою плагіна (junit-interface). SBT дозволяє користувачеві запускати тести вибірково або всі одразу. Це також дозволяє запускати лише ті тести, які раніше провалились, були виключені або мають залежність від перекомпільованого коду.

SBT може запустити Scala REPL з проектом, завантаженим у classpath. Можна викликати методи, визначені в коді проекту в рамках REPL.

SBT може бути налаштований на підпроекти. Це забезпечує кращі засоби для досягнення модульності. Залежні модулі можуть бути згруповані за допомогою однієї збірки та бажані модулі можуть бути повністю незалежними від інших.

SBT дозволяє включати зовнішні проекти, використовуючи їх шлях або URL-адресу. Це означає, що також можна включити Git репозиторії, як залежність для проекту. Це набагато спрощує роботу розробника, оскільки він / вона може легко мати проект, який залежить від іншого проекту.

SBT виконує завдання паралельно. Паралелізм завдань – це запуск одного або декількох самостійних завдання одночасно. Паралельність завдань забезпечує ефективність і масштабованість використання ресурсних систем. Це дійсно ефективно для паралельного запуску тестів.

SBT підтримує включення бібліотек як некерованих або керованих залежностей; тобто бібліотеки можна просто включити, додавши відповідний JAR у lib директорію або вказавши залежності у визначенні збірки. SBT використовує Apache Ivy для реалізації керованих залежностей.

У SBT властивості проекту, такі як залежності бібліотек, версія Scala і так далі, які необхідні для успішної збірки, оголошуються у визначенні збірки.

Визначення збірки build може бути файлом .sbt або .scala, або їх комбінацією. Файл .sbt повинен знаходитися в базовій директорії і, як правило, має назву build.sbt. Файл .scala повинен знаходитись у підкаталозі проекту базового каталогу.

Визначення збірки містить ключі та перетворення, які застосовуються до асоційованих значень. SBT аналізує проект і створює незмінну мапу, що описує збірку. Якщо визначення збірки знайдено, SBT використовує ці значення для ключів.

```
❯ sbt
sbt thinks that server is already booting because of this exception:
java.io.IOException: org.scalasbt.ipcsocket.NativeErrorException: [95] Operation not supported
Create a new server? y/n (default y)
y
[info] welcome to sbt 1.5.0 (Ubuntu Java 11.0.11)
[info] loading settings for project reftree-build from plugins.sbt ...
[info] loading project definition from /mnt/c/Users/Luv/Desktop/reftree/project
[info] loading settings for project root from build.sbt ...
[info] set current project to root (in build file:/mnt/c/Users/Luv/Desktop/reftree/)
[info] sbt server started at local:///home/q/.sbt/1.0/server/4698baf3d6e969678d41/sock
[info] started sbt server
```

Рис. 3.11. Збірка проекту за допомогою SBT

```
sbt:root> demoJVM / run
[warn] multiple main classes detected: run 'show discoveredMainClasses' to see the list

Multiple main classes detected. Select one to run:
 [1] reftree.demo.All
 [2] reftree.demo.FingerTrees
 [3] reftree.demo.Queues
 [4] reftree.demo.Teaser
 [5] reftree.demo.Zippers

Enter number: 1
[info] running reftree.demo.All
```

Рис. 3.12. Вибір main класу виконання

Модуль візуалізації є бібліотекою, яка призначена для використання іншими користувачами в своїх програмах, тому вона не має класу main. Клас main є точкою входу в програму. Оскільки він відсутній, при спробі запуску проекту SBT сповіщає про помилку.

```
sbt:root> run
[error] java.lang.RuntimeException: No main class detected.
[error]         at scala.sys.package$.error(package.scala:30)
[error] stack trace is suppressed; run last Compile / bgRun for the full output
[error] (Compile / bgRun) No main class detected.
```

Рис. 3.13. Помилка спроби запуску програми

3.3.Залежності та початок роботи з бібліотекою

Для початку використання бібліотеки, необхідно встановити Graphviz.

Graphviz – це програмне забезпечення для візуалізації графіків з відкритим кодом. Візуалізація графіків – це спосіб представлення структурної інформації як діаграм абстрактних графіків та мереж.

Програми візуалізації Graphviz беруть описи графіків мовою простого тексту і складають схеми в корисних файлових форматах, таких як зображення та SVG для веб-сторінок, а також PDF або Postscript для включення в інші документи. Graphviz має безліч корисних функцій для конкретних діаграм, таких як параметри кольорів, шрифтів, макетів табличних вузлів, стилів ліній, гіперпосилань та спеціальних фігур.

Graphviz безпосередньо покладається на dot. dot малює спрямовані графіки як ієрархії. Він читає графічні текстові файли та робить рисунки у графічних форматах, таких як GIF, PNG, SVG, PDF або PostScript.

dot приймає введення мовою DOT. Ця мова описує три основні типи об’єктів: графіки, вузли та ребра.

dot малює графіки у чотири основні фази. Процедура макетування, яка використовується dot, залежить від ациклічності графіку. Таким чином, перший крок – зламати будь-які цикли, що відбуваються на вхідному графіку, змінюючи внутрішній напрямок певних циклічних ребр. Наступним кроком призначають вузли для дискретних рангів чи рівнів. На кресленні зверху вниз ранги визначають координати Y. Краї, що охоплюють більше ніж один ранг, розбиваються на ланцюжки «віртуальних» вузлів. Третій крок упорядковує вузли в рангах, щоб уникнути перетину. Четвертий крок встановлює X координати вузлів, щоб ребра були короткими, а кінцевий крок спрямовує ребра сплайнів.

Перед використанням безпосередньо у своєму проекті, ознайомитись з бібліотекою можна за допомогою демо-проекту, розробленого як приклад використання та роботи бібліотеки.

Для включення бібліотеки у власний проект, необхідно додати такі рядки у build.sbt:

```
libraryDependencies ++= "reftree" % "latest-version"
```

Висновки до розділу

У розділі було розглянуто реалізацію та використання розробленої бібліотеки, призначеної для візуалізації структур даних. Бібліотека дає можливість користувачам покращити документацію їх проектів, використовувати згенеровані графіки на демонстраціях та презентаціях проектів, а також для кращого розуміння роботи різних структур мови Scala, та будь-де, де можуть бути необхідні структури даних Scala.

ВИСНОВКИ

Розуміння числових даних big data викликає плутанину у більшості людей, що стало великою мотивацією думати про нові рішення, які дозволяють людям найкраще зрозуміти та дослідити дані. Розумний підхід до представлення даних полягає у їх візуалізації.

Візуалізація даних – це процес подання даних у графічному або наочному вигляді в чіткій та зрозумілій формі. Вона виникла як потужний та широко застосовуваний інструмент для аналізу та інтерпретації великої кількості складних даних. Візуалізація стала швидким та простим засобом передачі концепцій в універсальному форматі. Вона передає складні ідеї з чіткістю, точністю та ефективністю. Ці переваги дозволили візуалізації даних бути корисною у використанні в багатьох галузях досліджень, безпосередньо у комп’ютерному програмуванні.

З виконаної роботи видно, що існує багато пробілів в області представлення даних. Проте, починаючи невеликими кроками заповнювати ці пробіли та усувати проблеми, можна побачити що візуалізація даних дозволяє отримати можливість інакше поглянути на структури даних та роботу з ними.

В ході дипломного проектування були визначені діаграми структур даних, які складаються з двох елементів: блоків, які представляють класи сутностей, і стрілок, які представляють класи набору. На прикладах було проілюстровано їх застосування, описано практичне використання при проектуванні та вивченні механізованих інформаційних систем.

На основі цих даних було прийнято рішення взяти за основу реалізації модуля структуру цих діаграм даних та їх відносин, що дає можливість нативної візуалізації цих блоків.

В межах проведеної роботи були розглянуті структура функціональних компонентів комп’ютера, роботи регістрів і пам’яті. Було обране середовище виконання JVM, так як воно є універсальним для всіх платформ. При детальному порівнянні мов, які працюють у цьому середовищі – Scala та Java, було обрано Scala, так як Scala працює на стандартній платформі Java та сумісна з усіма Java бібліотеками, при цьому структури мови Scala роблять її кращою при масштабуванні, а її код більш компактним. Було також

спростовано поширену думку про шкоду для продуктивності функціонального підходу до програмування.

В третьому розділі була описана реалізація Scala бібліотеки, представлені частини її вихідного коду, наведені приклади використання за допомогою відповідного API та розглянута можливість включення бібліотеки користувачами у власні проекти. Завдяки функціональному підходу Scala для реалізації map-функцій були використані монади та моноїди, що дозволяє з легкістю відокремити бізнес-логіку модуля та при бажанні зробити реалізацію цього модуля будь-якою мовою, що підтримує функції вищого порядку та замикання.

Одні з основних сценаріїв використання розробленої бібліотеки, які варто зазначити:

- візуалізація структур даних в реальному часі дозволяє явно і чітко показати відмінності і особливості реалізації різних структур даних (масивів, сетів, пов’язаних списків, двосторонніх пов’язаних списків, черг і т. д.);
- профілювання даних в реальному часі дозволяє швидко визначити вразливі місця алгоритму, що виконується;
- доповнення документації наочними діаграмами (кращі та бажані практики поточного проекту) для поглиблення та прискорення розуміння структур даних і структури проекту розробниками;
- відображення відмінностей та особливостей структур даних та реалізованих на їх основі алгоритмів, під час сесій парного програмування або code review.