Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

ВІЗУАЛЬНЕ ПРОГРАМУВАННЯ БІЗНЕС-ПРОЦЕСІВ ЗАСОБАМИ РЕАКТИВНОГО ПРОГРАМУВАННЯ

Текстова частина
магістерської роботи
за спеціальністю «Інженерія програмного забезпечення» 121

Керівник магістерської роботи
доц. Жежерун О.П.

Виконав студент
Проскурня Д.О.

Київ 2020

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри мультимедійних систем, к.ф-м.н.
_____В. В. Бублик
(підпис)
„____”_____2019 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на дипломну роботу

студенту 2 року МП ІПЗ факультету інформатики__
Проскурні Дмитру Олександровичу_____
Розробити Комп’ютерну систему для візуального програмування бізнес-процесів
Зміст ТЧ до магістерської роботи:

Зміст
Анотація
Вступ
1 Створення складних програмних систем засобами ООП
2 Огляд реактивної парадигми програмування
3 Програма для візуального програмування бізнес-процесів
Висновки
Список літератури
Додатки

Дата видачі „___” _____ 2019 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Тема: Візуальне програмування бізнес-процесів засобами реактивного програмування

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу.	01.11.2019	
2.	Огляд технічної літератури за темою роботи.	17.01.2020	
3.	Аналіз недоліків ООП та варіантів їх усунення	31.01.2020	
4.	Проектування комп'ютерної програми	14.02.2020	
5.	Розробка комп'ютерної програми	03.04.2020	
6.	Опис структури і принципів роботи комп'ютерної програми	17.04.2020	
7.	Створення слайдів для доповіді та написання доповіді.	20.04.2020	
8.	Оформлення текстової частини магістерської роботи	03.05.2020	
8.	Написання доповіді та попередній захист магістерської роботи.	30.05.2011	
10.	Остаточне оформлення пояснювальної записки та слайдів.	9.06.2020	
11.	Захист магістерської роботи	16.06.2020	

Студент Проскурня Д. О.

Керівник Жежерун О. П.

“_____”_____

Зміст

Анотація

Ця робота має на меті дослідити можливості моделювання бізнес-процесів засобами реактивного програмування. Інструментом для побудови розглядається бібліотека ReactiveX. В результаті роботи було розроблену систему для візуального моделювання процесів у вигляді орієнтованого графа, а також транслятор цієї конфігурації у JavaScript.

Ключові слова: **реактивне програмування**, **моделювання бізнес-процесів**, **візуальне програмування**

Вступ

Алан Кей ввів термін «об’єктно-орієнтоване програмування» у 1966 чи 1967 році. Великою ідеєю було використання невеликих інкапсульованих обчислювальних приладів, які би спілкувалися через передачу повідомлень, а не прямий обмін даними. Причиною цьому було бажання припинити розбивати програми на окремі «структури даних» та «процедури» [1].

За словами Алана Кей, основними складовими об’єктно-орієнтованого програмування мають бути:

- обмін повідомленнями;
- інкапсуляція;
- динамічне зв'язування [2].

Примітно, що Алан Кей, людина, яка запровадила та вивела назагал термін “об'єктно-орієнтоване програмування”, не вважав поліморфізм через успадкування та власне успадкування необхідними інгредієнтами ООП.

Те, що сьогодні називають об'єктно-орієнтованим програмуванням є зовсім не тим, що мав на думці його творець: “Я придумав термін “об'єктно-орієнтоване програмування”, і ось що я вам скажу: я не мав на увазі C++” [3].

Перший розділ розглядає недоліки ООП, які виникли через відхилення від оригінальної ідеї, запропонованої Аланом Кеєм.

Другий розділ пропонує альтернативні підходи до створення програмних систем, зміщуючи фокус із архітектури та блокуючого

виконання на дизайн системи навколо даних та використання push-стратегії розповсюдження змін.

У заключному розділі надається опис комп’ютерної системи для візуального програмування бізнес-процесів, яка була розроблена в процесі написання цієї роботи.

Постановка задачі: Розробити програмну систему для візуального програмування бізнес-процесів.

Сучасне ООП приділяє величезну увагу архітектурі системи. І це зрозуміло, адже неправильно спроектовану систему дуже складно підтримувати та покращувати. Програмна система за обєктно-орієнтованої парадигми - це ієрархія обєктів, які інкапсулюють у собі стан. А що може бути краще за ієрархічну організацію? Але проблема полягає у тому, що комунікація між цими об'єктами має відбуватися виходячи з тієї ж самої ієрархії, і об'єкти мають доступ лише до своїх синів. Тому зв'язок між компонентами з різних гілок ієрархії повинен встановлюватися через спільного батька, що іноді призводить до не зовсім інтуїтивного розташування логіки у системі.

Ця робота має на меті дослідити причини такого стану речей та варіанти покращення підходів до створення складних (розподілених) програмних систем.

Зокрема, пропонується використання push-стратегії розповсюдження даних замість загальноприйнятої pull-стратегії та досліджувати інші, крім текстового, способи програмування.

1 Недоліки ООП

1.1 Жорстка зв’язність даних та методів їх обробки

Об'єкти пов'язують функції і структури даних разом в неподільні сутності. Брайан Вілл вважає, що це фундаментальна помилка, тому що функції і структури даних - поняття, що відносяться до абсолютно різних категорій [4].

Функції виконують дії. У них є вхідні і вихідні значення. Вхідні і вихідні значення - це структури даних, які змінюються функціями. У багатьох мовах програмування функції - це послідовності імперативів. Для розуміння суті функції необхідно зрозуміти порядок виконання цих імперативів (в лінивих функціональних і в логічних мовах порядок може значення не мати).

Структури даних просто існують. Вони не виконують ніяких дій. Вони виключно декларативні. Суть структури даних в рази простіша за суть функції. Функції можна представляти чорними ящиками, що трансформують дані із типу аргументів у тип результату.

Такий підхід протирічить принципу єдиного обов’язку – одному з головних принципів об’єктно-орієнтованого програмування. Насправді головною ідеєю цього принципу є не те, що клас має мати лише один обов’язок, а те, що не має бути більше однієї причини вносити зміни у клас [5]. Але якщо зберігати і дані, і логіку їх обробки в одному і тому ж класі, ми завжди маємо принаймні дві потенційні причини для змін: впровадження іншого формату даних чи нового алгоритму їх обробки.

Для прикладу можна розглянути React - декларативну, ефективну і гнучку JavaScript-бібліотеку, призначену для створення користувацьких інтерфейсів. Вона дозволяє компонувати складні інтерфейси з невеликих окремих частин коду - “компонентів” [6].

Компоненти є обєктно-орієнтованим аспектом React у тому сенсі, що вони мають інкапсульований стан та інтерфейс для взаємодії з батьківськими компонентами.

Але React також має багато рис функціональної парадигми. Стан React-застосунку розтікається з кореневого компонента по всій ієрархії. Тобто ієрархію компонентів можна представити як функцію, результатом якої є користувацький інтерфейс, а аргументом - дані-стан. До речі, прості компоненти, які використовуються лише для відображення даних а тому не не мають внутрішнього стану і не взаємодіють з батьківськими рекомендується визначати саме як функції, а не класи.

Попри таку просту і логічну організацію структури, React-розробники зіткнулися з проблемою взаємозвязку між компонентами, які є віддаленими візуально, але мають відображати одні й ті ж дані та бути інтерактивними. Ця проблема полягає в тому, що встановити такий звязок можна лише через спільний батьківський компонент, який може бути досить віддаленим у ієрархічній структурі. Нові бізнес-вимоги часто змушували переносити стан та код його обробки вище по ієрархії, поміщаючи його в компоненти, в яких його розташування могло бути абсолютно не логічним.

У 2015 році Ден Абрамов і Ендрю Кларк представили Redux - відкриту JavaScript бібліотеку, призначена для управління станом програми. Redux дає змогу відокремити дані від компонентів. Тепер

компоненти отримують дані для відображення зі єдиного джерела - сховища Redux. Ці дані призначені лише для читання, а зміни відбуваються централізовано через шину подій [7].

1.2 Блокуючий виклик методів

Оригінальна ідея Алана Кея про об’єктно-орієнтоване програмування передбачає комунікацію між об’єктами шляхом передачі повідомлень. Мова програмування Smalltalk, розробку якої він очолював, саме так і працює. Сучасні ж об’єктно-орієнтовані мови пішли іншим шляхом: замість передачі повідомлення об’єкту вони викликають метод об’єкта. На перший погляд може здатися, що принципової різниці між цими двома підходами немає, що це лише семантика. Проте перевага передачі повідомлень насправді розкрилася коли перестав працювати закон Мура - з запровадженням багатопоточності. Якщо повідомлення можна складати у спеціальну чергу і так по одному і обробляти, незалежно від того, з якого саме потоку виконання прийшло це повідомлення, то виклик методів одного й того ж об’єкту може відбуватися одночасно різними потоками. Одночасна зміна стану різними потоками по факту руйнує інкапсуляцію, яка є одним із «китів» об’єктно-орієнтованого програмування.

Модель акторів – це математична модель одночасних обчислень, яка з’явилася 3 1973 році. У її основі лежить концепція акторів, які обмінюються асинхронними повідомленнями, завдяки чому досягається паралелізм [8]. Завдяки тому, що у Smalltalk комунікація

відбувається саме через обмін повідомленнями, а не виклик методів, її легко можна застосовувати у контексті моделі акторів, додавши додаткову абстракцію – черги повідомлень.

Мовам, які використовують блокуючий виклик методів довелося піти іншим шляхом – постійно синхронізувати дані, які можуть одночасно використовуватися різними потоками. Цей підхід дійсно вирішує проблему, але оскільки виконання коду є блокуючим, то потокам доводиться чекати, якщо інший потік уже працює у блоці синхронізованого коду. Крім того, потреба синхронізації створює додаткові накладні витрати.

1.3 Неоднозначність декомпозиції домену на класи

Створення програмної системи, як правило, починається з ідентифікації сутностей, які присутні в предметній області та визначення їхньої поведінки. Наприклад, після того, як було встановлено, що предметна область потребує сутність типу Message та визначено поля відповідного класу, потрібно вирішити, що саме ми хочемо робити з цим класом та імплементувати ці поведінки як методи класу.

Насправді ж розробники дуже часто стикаються з ситуацією, коли потрібна поведінка, яка по факту не належить жодній з наявних сутностей. Але оскільки вона має десь бути, доводиться придумувати для різні класи на кшталт Manager, Helper, Doer…

По-перше, це створює ще додаткові залежності у системі. А по-друге, згідно з ідею обєктно-орієнтованого програмування, класи

мають моделювати сутності реального, предметного світу, в якому не існує ефемерних контейнерів для нестандарних дій.

І якщо детально розглянути сучасний типовий мікросервіс, побудований за допомогою фреймворка Spring, то ми побачимо там не так і багато об’єктно-орієнтованого програмування.

Якщо Repository ще співвідноситься зі сховищем реального світу, то Controller і Service, які також повсюдно використовуються, є уявними сутностями. Repository і Service насправді прийшли з предметно-орієнтованого проектування. Вони вважаються будівельними блоками системи за цього підходу.

Більшість сутностей предметної моделі по факту є просто контейнерами для даних: вони мають лише поля, методи читання та запису. Дані отримуються з бази даних, траснформуються і передаються далі. Або ж дані приходять у Контролер та проходять через низку операцій: валідацію, трансформацію та запис у базу даних. Вони не можуть зберегти себе у базу даних, адже для цього потрібно було б впроваджувати у них залежності.

Повертаючись до Контролерів, Сервісів, Репозиторіїв та різних Адаптерів, хочеться зазначити, що вони є загальноприйнятими рівнями абстракцій системи. Але ж цих рівнів може бути і інша кількість. Реактивне програмування надає створює неявні абстракції на кожному кроці, а тому дуже легко піддається рефакторингу.

2 Вирішення проблем об’єктно-орієнтованого програмування

2.1 Система навколо даних

Якщо ми будемо в першу чергу думати про дані і створювати архітектуру програми виходячи з моделі даних, то це дасть нам безліч переваг.

Модель даних завжди є першим кроком у створенні будь-якої системи, об’єктно-орієнтована парадигма програмування вимагає кілька рівнів абстракції над цими даними. Це пояснюється необхідністю контролю доступу до даних та відділення бізнес логіки від технічних особливостей реалізації системи.

2.1.1 Паралельність

В наші дні неможливо позбутися того факту, що нам необхідно працювати з декількома ядрами. Ті, хто намагався розпаралелити ООП-код, дуже добре знають, наскільки це складно зробити ефективне рішення і уникнути помилок. Часто доводиться додавати безліч примітивів синхронізації, щоб уникнути одночасного доступу до даних з декількох потоків, і зазвичай багато потоків довгий час простоюють, чекаючи завершення роботи інших потоків. В результаті підвищення продуктивності виявляється досить посереднім [9].

Якщо ми застосуємо підхід з фокусом на дані, то паралелізація стає набагато простіше: у нас є вхідні дані, невелика функція їх обробки і вихідні дані. Це можна легко розділити на кілька потоків з мінімальною синхронізацією між ними.

Саме так працює Інтерфейс передачі повідомлень (MPI) – специфікація інтерфейсу для обміну повідомленнями між процесами. Обчислення розпаралелені на декілька потоків, які синхронізуються між собою лише тоді, коли це необхідно [10]. Сучасні мови програмування високого рівня мають вбудовану підтримку багатопоточності (Scala, Java 8+, Go), яка працює аналогічним чином. Вбудовану багатопоточність має також і Erlang, перший реліз якої відбувся у 1987 році, проте Erlang використовує зовсім іншу модель взаємодії [11].

2.1.2 Використання кешу

Крім використання багатоядерності одним з ключових способів досягнення високої продуктивності на сучасному обладнанні з глибокими конвеєрами команд і повільними системами пам'яті з декількома рівнями кешу є реалізація зручного для кешування доступу до даних. Data-oriented design дозволяє дуже ефективно використовувати кеш команд, тому що в ньому постійно виконується один і той же код. Крім того, якщо ми розташуємо дані великими суміжними блоками, то зможемо обробляти дані послідовно, домігшись майже ідеального використання кешу даних і відмінною продуктивністю.

2.1.3 Можливість оптимізацій

Коли ми думаємо про об'єкти або функції, то зазвичай зациклюємось на оптимізації на рівні функції або навіть алгоритму: намагаємося змінити порядок виклику функцій, міняємо метод сортування або навіть переписуємо частину коду на C на мові асемблера.

Подібні оптимізації звичайно корисні, але якщо спочатку задуматися про дані, то ми можемо відійти назад і створити більш масштабні і важливі оптимізації. Пам'ятаючи про потік даних, ми можемо приймати високорівневі, більш усвідомлені рішення на підставі способів перетворення і застосування даних. Подібні оптимізації в більш традиційних ООП-методиках можуть бути надзвичайно складними і витратними за часом.

2.1.4 Модульність

Всі перераховані вище переваги підходу побудови системи навколо даних були пов'язані з продуктивністю: використання кеша, оптимізації і паралелізація. Звичайно ж, продуктивність дуже важлива. Але часто виникає конфлікт між техніками, що підвищують продуктивність, і техніками, які сприяють читанню коду і простоті розробки. Наприклад, якщо переписати частину коду на мові асемблера, то ми підвищимо продуктивність, але це зазвичай призводить до зниження читабельності і ускладнення підтримки коду.

На щастя, побудова системи навколо даних і покращує продуктивність, і спрощує розробку. Якщо писати код спеціально для перетворення даних, то у результаті виходять маленькі функції з дуже малою кількістю залежностей від інших частин програмної системи. Кодова база залишається дуже «плоскою», з безліччю функцій-«листя», які не мають великих залежностей. Такий рівень модульності і відсутність залежностей сильно спрощує розуміння, заміну і оновлення коду.

2.1.5 Тестування

Остання важлива перевага побудови системи навколо даних - це простота тестування. Написання юніт-тестів для перевірки взаємодії

об'єктів - нетривіальне завдання. Необхідно створювати макети і тестувати кожен з елементів окремо. Це дуже монотонне та трудозатратне завдання. До того ж, у разі масштабного рефакторингу може виникнути необхідність також і переписувати юніт-тести. З іншого боку, працюючи безпосередньо з даними, писати юніт-тести легко і просто: створюємо якісь вхідні дані, викликаємо перетворюючу їх функцію, і перевіряємо, чи відповідають вихідні дані очікуваним. І на цьому все. Насправді це величезна перевага, яка дозволяє спрощувати тестування коду, тим самим підвищуючи надійність системи

2.2 Реактивне програмування.

Реактивне програмування — це парадигма програмування, побудована на потоках даних і розповсюдженні змін [12].

Розробнику системи корисно розуміти, як відбувається рух даних, які існують взаємозв’язки між її частинами. Проте об’єктно-орієнтована парадигма приховує цю інформацію. Сучасні середовища розробки програмного забезпечення намагаються вирішити цю проблему, надаючи інформацію про місця, де викликається той чи інший метод та де використовується той чи інший клас.

Рішенням цієї проблеми може бути використання push-стратегії замість традиційної для об’єктно-орієнтованої парадигми pull-стратегії. Pull-стратегія не є чимось новим, по факту це повсюдне використання патерну програмування “Спостерігач”, який і становить

основу реактивної парадигми програмування, допомагаючи будувати асинхронні потоки подій [13].

Шини подій або типові події натискань на кнопки - це все реальні приклади таких асинхронних потоків, які ви можете слухати і виконувати деякі побічні дії. Реактивне програмування дозволяє створювати потоки будь-чого, не тільки події натискань. Потоки легковагі і використовуються повсюдно: змінні, поля для вводу даних користувачем, властивості, кеш, структури даних і багато-багато іншого. Наприклад, стрічка Твітера може бути потоком даних нарівні з низкою подій користувальницького інтерфейсу. Тобто можна слухати потік і реагувати на події в ньому.

Більш того, реактивне програмування пропонує широкий набір інструментів і функцій для роботи з потоками, які дозволяють практично без обмежень створювати, фільтрувати, об’єднувати і поширювати ці потоки. Потоки також можуть бути використані як вхідні параметри інших потоків для створення потоків вищого рівня будь-якої складності.

Потік - це послідовність, що складається з подій, відсортованих за часом. У ньому може бути три типи повідомлень: значення (дані деякого типу), помилки і сигнал про завершення потоку.

Підписник потоку отримує ці події асинхронно, визначаючи окремий метод обробки кожного типу подій (або ж не визначаючи, і тоді подія буде просто ігноруватися). Це узагальнення патерну програмування “Спостерігач”: метод обробки звичайних повідомлень, так само як і методи обробки помилок та завершення потоку є спостерігачами.

Бібліотеки для реактивного програмування існують практично в усіх сучасних мовах програмування. Наприклад, біблотека ReactiveX має імплементації на Java, JavaScript, C#, C#(Unity), Scala, Clojure, C++, Lua, Ruby, Python, Go, Groovy, JRuby, Kotlin, Swift, PHP, Elixir, Dart. Але є і інші реалізації реактивного програмування. Наприклад, WebFlux від Spring розділяє потоки, які можуть містити єдиний елемент (Mono) від потоків, які містять послідовність елементів (Flux) [14].

2.2.1 Короткий огляд операторів реактивного програмування.

Справжню потужність реактивного підходу розкривають оператори. Оператори — це набір загальних функцій для роботи з асинхронними потоками. Набір операторів, які пропонує та чи інша імплементація реактивного програмування здебільшого аналогічний, і відрізняються вони лише термінологією. Оператори поділяються на такі категорії:

- оператори створення;
- оператори трансформації;
- оператори фільтрації;
- оператори об'єднання потоків;
- оператори обробки виняткових ситуацій;
- утилітні оператори;
- умовні та булеві оператори;
- математичні та агрегаційні оператори;

- оператори зворотного контролю навантаження;

- оператори конфігурації підписки [15].

Як було зазначено раніше, реактивне програмування застосовує push-стратегію. Однією з її переваг є виконання обчислень лише після підписки, а не заздалегідь, як це зазвичай відбувається за pull-стратегії. Це дозволяє, наприклад, пришвидшити первинне відображення веб-сторінки, адже виконуватися буде лише той код, який потрібен (лише ті потоки, на які підписалися спостерігачі). Застосування реактивних драйверів баз даних може пришвидшити отримання даних, адже можна отримувати елементи по-одному, як тільки вони стають доступними і застосовувати до них конвеєрну обробку. У мікросервсіному середовищі такий підхід може в рази пришвидшити виконання операцій над даними.

Потоки подій, якими оперує реактивне програмування, мають багато спільних рис із бізнес-процесами: вони є протяжними в часі, складаються з визначеного набору типів подій, які можуть виникати на конкретному етапі та мають певну логіку виконання.

2.2.2 Модель даних та модель процесів

Побудова будь-якої програмної системи починається з моделі даних. І це не дивно, адже дані - найдорожча цінність у сучасному світі. Модель даних - це концептуальне подання об'єктів даних та зв’язків між об'єктами у базі даних. Основна увага зосереджується на тому, як організовані об'єкти даних.

Модель даних, в першу чергу, потрібна для побудови бази даних. База даних - ключовий елемент будь-якої програмної системи. Вона є зліпком стану всієї системи і засобом збереження даних у часі.

Моделювання процесів - інший вектор побудови інформаційної системи. Модель бізнес-процесів допомагає зрозуміти, що саме і як відбувається у системі – описує «алгоритм», для забезпечення роботи якого будується архітектура системи.

2.2.3 ReactiveX як засіб моделювання бізнес-процесів

Природно, що підхід до моделювання процесів має бути іншим, ніж до побудови статичного графа об’єктів. Для цієї задачі чудово підходить push-стратегія розповсюдження змін. ReactiveX об’єднує імплементації реактивних розширень для багатьох популярних мов програмування.

Засоби цієї бібліотеки і пропонується використовувати для моделювання бізнес-процесів, оскільки вона має вбудовану підтримку асинхронності, а отже протяжності в часі. Багатий вибір операторів робить можливою будь-яку конфігурацію та взаємодію з іншими процесами.

Застосування ReactiveX надає низку переваг:

- єдиний інтерфейс для синхронного і асинхронного виконання коду;
- можливість отримання вектору даних, а не лише одиничних значень;
- ініціативність замість виконання наказу;
- вбудована підтримка протяжних у часі процесів;
- дані незалежні від методів їх обробки;

- безпечний рефакторинг;
- підтримка паралельного виконання.

Схематичний приклад конфігурації бізнес-процесу представлений у додатку А.

3 Visual Reactor

Більшість людей найкраще сприймають інформацію візуально. Ця теза справджується і для сфери розробки програмного забезпечення. Не дарма існують ER діаграми, UML діаграми, BPMN діаграми та багато-багато інших. Розглянемо детальніше три згадані типи діаграм.

ER (“Сутність-зв’язок”) діаграма є візуальним представленням структури бази даних. Вона є невід’ємною частиною розробки будь-якого складного застосунку. Як підказує її назва, складовими частинами цієї моделі є сутності та зв’язки між ними.

UML діаграма класів зображує високорівневу модель зв’язків між компонентами системи. Причому ці зв’язки можуть бути досить різноманітними (асоціація, агрегація, композиція, наслідування, імплементація та залежність). Діаграма класів є притаманною суто об’єктно-орієнтованому програмуванню.

Найцікавішою є BPMN: Модель та нотація бізнес-процесів є стандартом для моделювання бізнес-процесів що надає графічну нотацію для визначення бізнес-процесу у вигляді "Діаграми бізнес-процесу" (Business Process Diagram, BPD). Така діаграма ґрунтується на представлені бізнес-процесу у вигляді блок-схеми, що семантично схожа на діаграму діяльності.

Моделювання з використанням BPMN виконується у вигляді діаграм, що складаються з різних елементів. Розрізняють чотири категорії елементів:

- Об’єкти потоку керування: дії, події та логічні оператори
- Поєднуючі елементи: потік керування, потік повідомлень та асоціації
- Ролі: пули та доріжки
- Артефакти: дані, групи та текстові анотації

Отже, підсумуємо сказане: ER діаграма - це схема бази даних, BPMN - це схема процесів, які відбуваються у системі, а діаграма класів - це схема

архітектури програмного застосунку. І тут виникає питання: хіба програмний застосунок створюється не для автоматизації процесів? Навіщо будувати архітектуру, коли основне завдання - це моделювання процесів? Питання, звісно риторичне, адже без гарно продуманої архітектури неможливо створити складну систему.

Проте, можливо, все ж таки є спосіб це зробити. Реактивне програмування оперує потоками даних. Ці потоки мають багато спільного з процесами згаданої моделі і нотації бізнес процесів: вони є протяжними в часі, складаються з визначеного набору типів подій, які можуть виникати на конкретному етапі та мають певну логіку виконання.

Тому метою цієї роботи є дослідження доцільності використання засобів реактивного програмування для моделювання бізнес процесів.

3.1 Опис системи Visual Reactor

Visual Reactor - це візуальний конфігуратор реактивних потоків. Взаємодія між модулями за реактивної парадигми відбувається шляхом підписки на потоки подій. А кількість різноманітних підписок у великій системі може бути величезною, і може бути складно зрозуміти, як саме відбувається рух даних у системі.

Цю проблему частково вирішують різноманітні діаграми поведінки. Проте підтримання таких діаграм у актуальному стані потребує додаткових зусиль від розробників. Visual Reactor побудований навколо власне взаємодії між частинами програмної системи. Підписка на джерело даних відбувається кількома кліками мишки замість написання бойлерплейт-коду. Всі підписки поточного контексту є видимими, що забезпечує близьке до миттєвого розуміння руху даних у системі.

За задумом, Visual Reactor має максимально спростити встановлення взаємозв'язків між елементами системи, адже взаємозв'язки - це бізнес-логіка, з якою найкраще обізнані бізнес-аналітики. Якщо вони матимуть змогу поєднувати компоненти системи, програмістам не потрібно буде витрачати час на розуміння всіх тонкощів бізнес-процесів, і вони зможуть зосередитися на тому, що вони роблять найкраще - написанні коду.

Виконання різноманітних трансформацій даних, реалізація алгоритмів, складні обрахунки - все це потребує написання програмного коду (або навчання нейронних мереж). Такий підхід потенційно має стимулювати розробку через тестування (test driven development), тобто в результаті утворюється три незалежні команди, які працюють над системою: бізнес-аналітики, тестувальники та розробники.

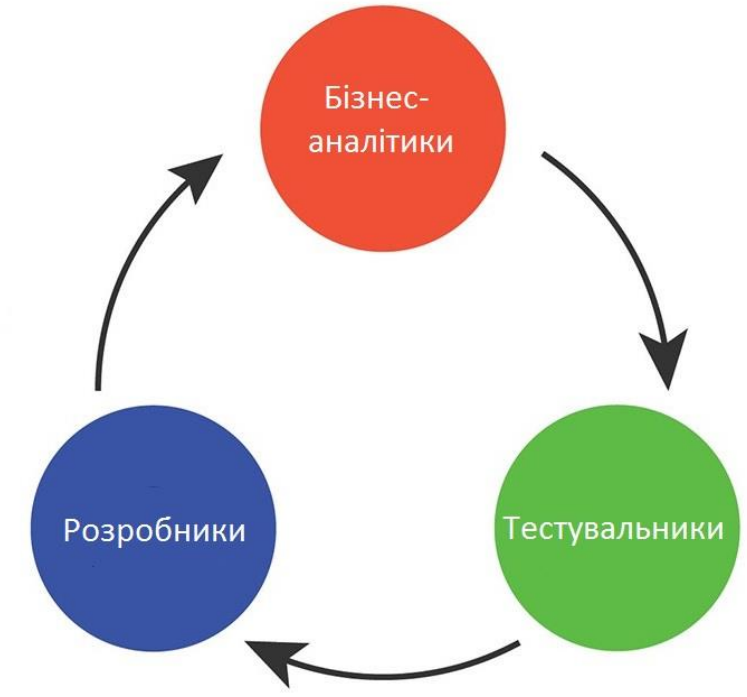


Рисунок 1. Взаємодія між командами бізнес-аналітиків, тестувальників та розробників за розробки через тестування.

Отже, завдяки візуалізації та конфігурації руху даних у системі замість традиційного написання коду для високорівнева побудова програмної системи не потребує спеціальних знань. Модель та нотація бізнес-процесів

(BPMN) фактично є стандартом для побудови бізнес-процесів. Робляться спроби автоматичної генерації коду з таких моделей. Потоки подій, сконфігуровані у Visual Reactor мають багато спільних рис із бізнес-процесами: вони є протяжними у часі, складаються із конкретних наперед визначених подій, можуть різним чином поєднуватися, розгалужуватися та взаємодіяти між собою. В подальшому планується дослідити можливість створення чіткого алгоритму для перетворення BPMN-моделей на конфігурації Visual Reactor. Це дозволить користуватися знайомими інструментами для створення початкової конфігурації системи.

Бібліотеки для реактивного програмування існують практично в усіх сучасних мовах програмування. Наприклад, бібліотека ReactiveX має імплементації на Java, JavaScript, C#, C#(Unity), Scala, Clojure, C++, Lua, Ruby, Python, Go, Groovy, JRuby, Kotlin, Swift, PHP, Elixir, Dart. Але є і інші реалізації реактивного програмування. Наприклад, WebFlux від Spring розрізняє потоки, які можуть містити єдиний елемент (Mono) від потоків, які містять послідовність елементів (Flux).

Тому конфігурація Visual Reactor є незалежною від мови програмування. Для запуску конфігурації іншою мовою програмування потрібно лише забезпечити сумісність аргументів реактивних операторів. І якщо трансляція примітивних типів даних не викликає складнощів, то функції зворотнього виклику, які широко використовуються багатьма операторами потребують фактичних реалізацій цільовою мовою.

Аналогічну проблему було вирішено у .NET фреймворку: він має вбудовану підтримку таких мов програмування, як C#, F#, J#, VB.NET, JScript.NET та C++/CLI. Підтримка великої кількості інших мов, в тому числі і Java (IKVM), стала можливою завдяки стороннім розробникам. Тому в подальшому планується дослідити можливість використання Common Intermediate Language як цільової.

Питання строгої типізації завжди було дискусійним. З одного боку, типізація є додатковим рівнем стабільності системи, а у тандемі з сучасними інтегрованими середовищами розробки вона значно полегшує життя розробникам завдяки контекстним підказкам. Аргументом проти жорсткої типізації є гнучкість та свобода. Проте зважаючи на стрімкий ріст популярності мови TypeScript, яка є типізованою надбудовою над мовою JavaScript, можна зробити висновок, що типізація все-таки - гарна річ. TypeScript використовує структурну типізацію, а не номінативну. Номінативна система типів вимагає явного оголошення приналежності об'єкта до певного типу, тоді як за структурної системи типу інтерфейс визначається самою структурою об’єкта - його полями та методами. Структурна типізація є більш гнучкою, але класичні об’єктно-орієнтовані мови програмування використовують номінативну систему, оскільки вона забезпечує захист від (випадкового) неправильного використання об’єктів. У цьому є сенс, адже абсолютно різні класи, які ніяк не пов’язані між собою, можуть мати поля чи методи з однаковими назвами та типами. Через це більшість мов не підтримують також і множинне наслідування.

Реактивні потоки працюють з даними, а не об’єктами. Тому використання об’єктів у розумінні класичних об’єктно-орієнтованих мов програмування не має сенсу, адже дані і логіка їх обробки є сегрегованими. В ідеалі, контекст виконання у операторів реактивного програмування має містити лише дані, отримані з попереднього кроку, а глобальні змінні відсутні взагалі.

Тому саме структурна система типів краще поєднується з реактивною парадигмою: додатковий захист, який пропонує номінативна система типів, просто не потрібен. А от гнучкість структурної системи типів є надзвичайно корисною. Завдяки ній можна не визначати явні типи для кожного реактивного оператора, оскільки компілятор може це робити неявно.

Тут також можна згадати, що мова програмування C++ також використовує структурну систему типів для шаблонів. Шаблони C++ є потужним інструментом для узагальненого програмування, вміле користування якими усуває необхідність писати бойлерплейт-код. TypeScript також має аналогічну потужну систему параметризації, яку дуже зручно використовувати для створення узагальнених функцій, класів та інтерфейсів.

Конфігурація Visual Reactor складається лише з двох типів сутностей: реакторів та підписок.

Реактор - це найменша складова частина системи. Реактори, подібно до об'єктів об'єктно-орієнтованої парадигми мають інкапсульований стан, який вони можуть виставляти назовні у вигляді Observable. Подібно агентам, реактори реагують на зміни даних у середовищі системи завдяки підписці на потоки даних, які їх цікавлять. підписки можуть бути як статичними, тобто існувати протягом всієї роботи програми, так і динамічними, тобто реактори можуть підписуватися і відписуватися від джерел даних самостійно виходячи зі своєї логіки роботи.

Конфігурація окремого реактора містить список вхідних типів даних, вихідних типів даних та інформацію про батьківський реактор.

Підписки уособлюють собою зв'язки між реакторами. Кожна підписка має інформацію про джерело даних та підписника, а також метадані самої підписки (як то налаштування кешування, буферизації, фільтрації тощо). В процесі розробки системи було зроблено припущення, що відображати як реактори потрібно лише трансформаційні оператори, інші ж оператори мають бути конфігурацією реакторів та бути відображені візуально через використання різних кольорів з'єднань та реакторів, різних типів ліній та фігур тощо. Тобто досвідчений розробник Visual Reactor повинен розуміти, як саме працює та чи інша підписка лише поглянувши на схему, без необхідності вивчати її конфігурацію.

3.2 Потенційні можливості системи

Обробка виняткових ситуацій є важливим аспектом у створенні високоякісних програмних систем. Класичне блокуюче виконання програмного коду сильно обмежене у можливостях через необхідність негайного повернення значення. Реактивне програмування не має такого обмеження, а тому має величезну перевагу у цій сфері. Варіанти дій при виникненні виняткової ситуації практично необмежені: від банального “прийняти помилку” до звернення до іншого мікросервісу для уточнення даних.

Попри свою назву, виняткові ситуації не є чимось надзвичайним і присутні у будь-якій програмній системі, тим паче якщо вона працює з мережею, адже незважаючи на те, наскільки скрупульозно розробники пишуть програмний код та наскільки якісна валідація даних у системі, уникнути мережевих проблем неможливо. Робота з винятковими ситуаціями має бути максимально аналогічною до звичайного конфігурування потоків подій та природно поєднуватися з ними. Потоки обробки помилок планується винести в окремий шар, відображення якого можна вмикати і вимикати за бажанням.

Програмування постійно еволюціонує, рівень мов програмування поступово зростає і рядові розробники не мають уявлення, як працює асемблер.

Проводячи аналогію з архітектурою будівель, різноманітні діаграми програмної системи - це різні типи планів будівлі. Розробники за цими планами за допомогою тієї чи іншої мови програмування будують програмну систему, як будівельники цеглина за цеглиною зводять будинок. Тоді Visual Reactor - це 3D принтер, який

будує систему одразу з конфігурації, прокладаючи усі необхідні комунікації.

На мою думку, цей проект має величезний потенціал, адже він поєднує в собі найкращі риси моделі акторів, агентного програмування, аспектного програмування, реактивну, об'єктно-орієнтовану та функціональну парадигми програмування.

Розглянемо декілька потенційних можливостей системи Visual Reactor:

- автоматичне горизонтальне масштабування.

  Оскільки найменшою складовою частиною Visual Reactor є оператор, який по факту є функтором від асинхронного Observable, ця модель чудово поєднується з безсерверними обчисленнями. А однією з основних переваг безсерверної архітектури є автоматичне горизонтальне масштабування.

  Проте навіть за звичайного мікросервісного підходу завдяки гомогенній структурі з єдиним інтерфейсом звязку та інкапсульовану вкладеність будь-який модуль можна розбити на складові частини, які будуть стануть комунікувати через мережу замість прямого звязку в межах одного контейнера.
- розумна обробка виняткових ситуацій

  Оскільки комунікація у системі відбувається шляхом передачі серіалізованих повідомлень, поточний стан системи є результатом поступової обробки всіх повідомлень, які надійшли у систему через підписки. Це відкриває широкі можливості зокрема і у обробці виняткових ситуацій. Подію, що спричинила виняткову ситуацію, завжди можна залогувати разом із поточним станом компонента. Цю інформацію також можна

надіслати спеціальному обробнику помилок, який вирішить, як саме потрібно вчинити в конкретній ситуації. Причому виняткова ситуація може бути перехоплена і ніколи не дійти до кінцевого користувача, незважаючи на те, що її вирішення є асинхронним.

- вся хмара - локально

  Оскільки система складається лише з потоків подій та функцій зворотного виклику, її запуск обмежений лише кількістю операційної пам’яті, необхідної для утримання системного коду. Тому кожен розробник може локально розгорнути всю систему
- інтеграція з хмарними сервісами

  робота

Висновки

Програмування - це спосіб передачі інструкцій комп’ютеру. Програмування почалося з експериментів, щоб зрозуміти для чого взагалі можна використовувати комп’ютери, а сьогодні є величезною індустрією, яка розвивається колосальними темпами. З ростом обчислювальних потужностей фокус розвитку програмування все більше спрямовується на зручність та швидкість розробки. Зрозуміло, що при цьому дещо страждає ефективність роботи програмних систем, але у світі, де час - це гроші, люди готові нею жертвувати.

Сучасні мови програмування вже не є чистими представниками якоїсь однієї парадигми, вони в тій чи іншій мірі поєднують у собі риси багатьох парадигм, адже по суті парадигма не має значення, дійсно важливими є швидкість та зручність розробки, необхідний фаховий рівень розробників для ефективного програмування та простота внесення змін до початкової концепції у процесі розробки чи навіть у готовий продукт.

На одній з конференцій я почув фразу “Страждати повинні машини, а не люди”. Мається на увазі, що всю тяжку монотонну роботу, яка не потребує

Я переконаний, що прийде час, коли програмування буде настільки ж поширеним та необхідним, як і вміння читати та писати. І я вірю, що воно буде відбуватися у абсолютно іншому форматі. Але програмування сьогодні - це дійсно нетривіальна справа.

Програмний код сучасний програмних систем розпорошений по великій кількості файлів, і ознайомлення з незнайомою кодовою базою потребує значних зусиль.

Система для візуального конфігурування потоків подій, запропонована в цій роботі, має на меті усунути необхідність проектувати складні, переповнені абстракціями архітектури програмних систем. Натомість вона пропонує оперувати напряму бізнес-процесами, застосовуючи засоби реактивного програмування для створення єдиного інтерфейсу зв'язку між компонентами як в межах одного мікросервісу, так і між розподіленими мікросервісами.

Концепт візуального конфігурування асинхронних реактивних потоків було перевірено та на практиці доведено можливість такого підходу.

Поточна реалізація системи вимагає низки доопрацювань. До наступного релізу необхідно:

- підключити сервіс для статичної типізації;
- підтримати зручну роботу з вкладеними операторами;
- забезпечити обробку помилок.

Наразі компілятор існує лише для TypeScript/JavaScript, планується поступова підтримка інших мов програмування. Це дозволить мати всю систему в одному проекті і безшовно з'єднувати мікросервіси.

Компілятор та середовище розробки Visual Reactor не є окремими проектами, а тому роботи по їх покращенню можуть йти паралельно. Крім того, беручи модель реактор-підписка за проміжний інтерфейс, можливе створення альтернативних програм для створення таких конфігурацій чи більш оптимальних та ефективних компіляторів.

Для полегшення роботи з потоками потрібно також зробити підказки для користувачів щодо того, як саме буде відбуватися рух

даних. Завдяки цим підказкам систему можна буде використовувати для розвитку навичок реактивного програмування.

Я не має сумнівів, що одного дня ми зможемо спілкуватися з роботами, як це бачить Майкл Крайтон, сценарист фільму «Край „Дикий Захід“». Але для цього програмування однозначно має стати набагато простішим та ефективнішим заняттям.

Список використаної літератури

1. Alan Kay On Messaging [Електронний ресурс] / Alan Key // Електронний лист — 1998 — Режим доступу: https://wiki.c2.com/?AlanKayOnMessaging.
2. Alan Kays Definition Of Object Oriented [Електронний ресурс] — Режим доступу: https://wiki.c2.com/?AlanKaysDefinitionOfObjectOriented
3. The Forgotten History of OOP [Електронний ресурс] / Eric Elliott // Онлайн публікація — 2018 — Режим доступу: https://medium.com/javascript-scene/the-forgotten-history-of-oop-88d71b9b2d9f
4. Object-Oriented Programming: A Disaster Story / Brian Will // Онлайн публікація — 2015 — Режим доступу: https://medium.com/@brianwill/object-oriented-programming-a-personal-disaster-1b044c2383ab
5. Robert C. Martin [Електронний ресурс] / The Single Responsibility Principle // Онлайн публікація — 2014 — Режим доступу: https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html
6. React [Електронний ресурс] / Опис бібліотеки — Режим доступу: http://web.archive.org/web/20200411084918/https://uk.reactjs.org/
7. Redux [Електронний ресурс] / Опис бібліотеки — Режим доступу: http://web.archive.org/web/20200605060438/https://redux.js.org/
8. Carl Hewitt. A Universal Modular ACTOR Formalism for Artificial Intelligence. / Carl Hewitt, Peter Bishop, Richard Steiger // IJCAI — 1973 — Режим доступу:

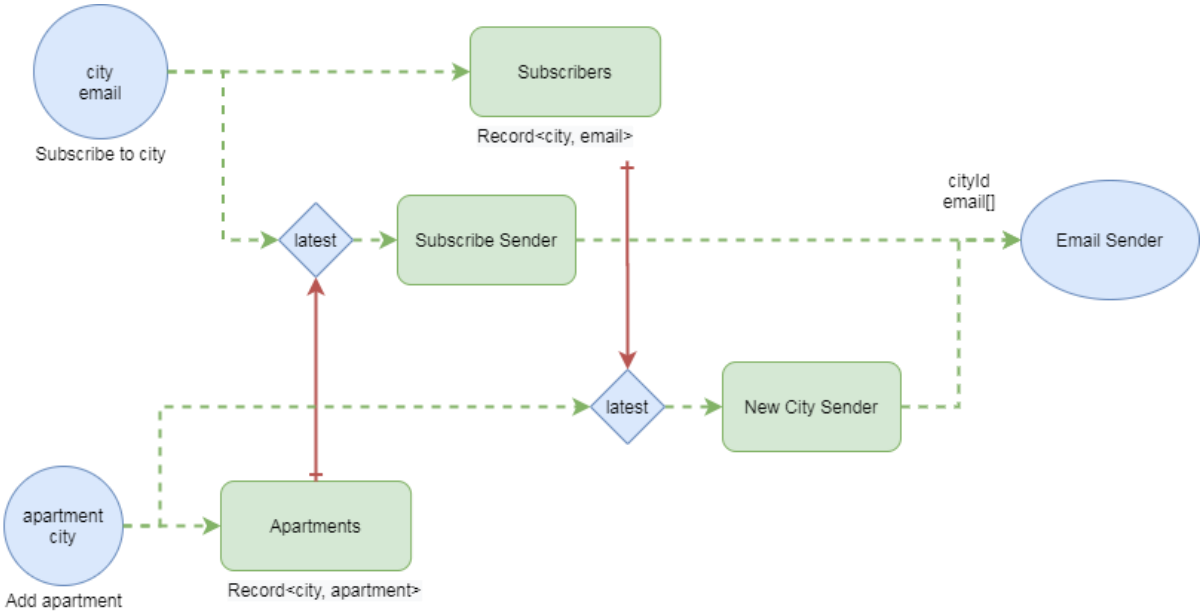
https://www.ijcai.org/Proceedings/73/Papers/027B.pdf

9. Eduardo Pinho. An object-oriented parallel programming language for distributed-memory parallel computing platforms. [Електронний ресурс] / Eduardo Pinho, Francisco de Carvalho-Junior. — 2013 — Режим доступу: https://doi.org/10.1016/j.scico.2013.03.014
10. The Message Passing Interface (MPI) standard [Електронний ресурс] / Специфікація MPI-1.1 — 1995 — Режим доступу: https://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-1.1/node2.htm#Node2
11. Erlang Concurrent Programming [Електронний ресурс] — Режим доступу: http://web.archive.org/web/20200530054255/https://erlang.org/doc/getting_started/conc_prog.html
12. Engineer Bainomugisha. A survey on reactive programming // Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, Wolfgang de Meuter — 2013 — Режим доступу: https://doi.org/10.1145/2501654.2501666
13. ReactiveX [Електронний ресурс] / Опис бібліотеки — Режим доступу: http://web.archive.org/web/20200513114925/http://reactivex.io/intro.html
14. Sping FebFlux [Електронний ресурс] / Документація — Режим доступу: http://web.archive.org/web/20200521071159/https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html
15. ReactiveX Operators [Електронний ресурс] / Документація — Режим доступу:

http://web.archive.org/web/20200520074530/http://reactivex.io/documentation/operators.html

Додаток А

Приклад конфігурації бізнес-процесу засобами реактивного програмування



Легенда:

коло	вхід подій
овал	вихід подій
прямокутник	реактор (об’єкт, який підписаний на події та є джерелом подій)
ромб	оператор об’єднання потоків
пунктирна стрілка	постійна підписка
суцільна стрілка	підписка на одиничне значення