Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Факультет інформатики

Кафедра інформатики

Розробка предметно-орієнтованої мови програмування для скінченних автоматів

Domain specific language development for finite state machine

Текстова частина до магістерської роботи

за спеціальністю «Інженерія програмного забезпечення»

Виконав: студент 2-го року навчання

Кузів Павло Михайлович

Керівник: Франчук Олег Васильович, кандидат технічних наук, доцент

Рецензент_____

(прізвище та ініціали)

Магістерська робота захищена

з оцінкою «_____»

Секретар ДЕК _____

«____» _____2021 р

Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра інформатики
Магістерська програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Зав.кафедри інформатики, доц.,
к.ф.-м.н.

_____С. С. Гороховський
(підпис)

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на дипломну роботу

студенту 2 року навчання МП «Інженерія програмного забезпечення» інформатики

Кузіву Павлу Михайловичу

на тему розробка предметно-орієнтованої мови програмування для скінченних автоматів

Зміст ТЧ до магістерської роботи:

Зміст

Анотація
Вступ
1. Визначення та типи предметно-орієнтованих мов програмування.

1.1. Скінченні автомати.

2. Взірці реалізації скінченних автоматів в імперативних мовах програмування.

2.1. Синтаксис предметно-орієнтованої мови програмування для скінченних автоматів.

3. Аналіз програмної реалізації компілятора предметно-орієнтованої мови програмування для скінченних автоматів

Висновки

Список літератури

Дата видачі 25 жовтня 2020 р.

Керівник: кандидат технічних наук, доц. О.В. Франчук

(підпис)

Завдання отримав П.М. Кузів

(підпис)

Зміст

Анотація

Робота складається з трьох розділів. У першому розділі розглянуто базові поняття. Дано визначення предметно-орієнтованих мов програмування, розглянуто типи та їх особливості, а також переваги та недоліки використання. Також описано математичну модель скінченного автомату, та наведено приклад використання. У другому розділі описано взірці реалізації моделі скінченного автомату в імперативних мовах програмування, розглянуто їхні переваги та недоліки. Також у другому розділі виведено синтаксис предметно-орієнтованої мови програмування для скінченних автоматів, та подано його в BNF формі. Третій розділ містить детальний опис реалізації компілятора для компіляції отриманого синтаксису.

Вступ

Програмне забезпечення проникає в усі сфери життя людей. Тому якість, стабільність та гнучкість програмного коду стає надзвичайно важливою. Для ефективного опису предметної області важливим є вибір правильної моделі та побудови гнучких абстракцій. Однією з таких моделей є скінченний автомат. Модель скінченного автомату можна ефективно застосовувати для вирішення широкого класу задача, від низькорівневого програмування до програмування графічних інтерфейсів. Щоправда, системи, що мають велику кількість станів, та подій, на які вони реагують, дуже швидко стає важко підтримувати, змінювати та доповнювати. Дану проблему можна вирішити за допомогою побудови невеликої предметно-орієнтованої мови програмування з лаконічним та виразним синтаксисом.

Предметно-орієнтовані мови програмування (ПОМП) існують вже доволі давно. Деякі з них стали не від'ємною частиною мов програмування загального призначення. Популярність ПОМП має декілька причин. ПОМП дозволяє значно збільшити продуктивність розробки програмного забезпечення, надаючи лаконічний синтаксис. Також, ПОМП покращує комунікацію розробників з експертами предметної області, оскільки синтаксис таких мов є водночас компільованим, виконуваним кодом програми та зрозумілим описом предметної області, що робить такі мови програмування доступними для не технічних спеціалістів. Поєднання предметно-орієнтованої мови програмування та скінченних автоматів є ефективним інструментом для аналізу та розробки складних систем.

Завдання полягає у створенні предметно-орієнтованої мови програмування для опису скінченних автоматів, що передбачає вивід лаконічного синтаксису, який дозволить компактно та зручно описувати скінченний автомат, та

реалізацію компілятора, що перетворюватиме цей синтаксис у виконуваний, швидкий код мови програмування загального вжитку. Більш того, етап компіляції необхідно використати також і для семантичного аналізу, щоб запобігти помилкам, що можуть виникнути на етапі виконання програмного забезпечення. Важливим критерієм у реалізації такого інструменту є також можливість з мінімальними зусиллям розширювати кількість мов програмування, у які код предметно-орієнтованої мови програмування може бути скомпільований.

Розділ 1. Теоретичні основи

Предметно-орієнтована мова програмування, визначення та типи.

Означення (1.1): предметно-орієнтованою мовою програмування називають мову комп'ютерного програмування з обмеженою областю вжитку, призначеною для розв'язання задач у конкретній предметній області [10].

З означення (1.1) випливають чотири ключові елементи предметно-орієнтованої мови програмування:

- Мова комп'ютерного програмування: предметно-орієнтована мова програмування використовується для опису інструкцій, що виконуються обчислювальною машиною. Як і будь-яка сучасна мова програмування, структура цієї мови повинна бути зрозуміла програмісту, і водночас повинна бути виконувана комп'ютером.
- Природність мови програмування: предметно-орієнтована мова програмування повинна надавати відчуття свободи виражальних можливостей, надаючи можливість використовувати не лише окремі синтаксичні вирази, а й можливість їх композиції.
- Обмеженість виразності: мова програмування загального вжитку надає багато можливостей, зокрема, підтримка різноманітних структур даних, абстрактних структур, керування пам'яттю – усе це корисні елементи мови, але вони ускладнюють вивчення та користування мовою. Предметно-орієнтовані мови програмування повинні підтримувати лише необхідний мінімум цих можливостей для надання зручного та простого синтаксису опису предметної області. Програміст не має змоги побудувати всю програмну систему за допомогою предметно-орієнтованої мови

програмування. Натомість, предметно-орієнтовану мову програмування використовуються лише для конкретного аспекту певної програмної системи.

- Орієнтованість на предметну область: обмежена мова програмування корисна лише в тому випадку, якщо має чітко визначену скерованість на невелику предметну область. Орієнтованість на предметну область – це визначальний елемент цінності предметно-орієнтованої мови програмування.

Варто зазначити, що орієнтованість на предметну область є наслідком обмеженої виразності предметно-орієнтованої мови програмування.

Предметно-орієнтовані мови програмування поділяють на три категорії: зовнішні, внутрішні та мовні інструменти середовища розробки.

Зовнішні предметно-орієнтовані мови програмування – це мови програмування, повністю відокремлені від основної мови розробки програмної системи. Як правило, зовнішні предметно-орієнтовані мови програмування мають власний синтаксис, але часто використовують синтаксичні вставки інших мов (часто XML). Синтаксичний аналіз таких ПОМП зазвичай виконують за допомогою інструментів основної мови програмної системи. Прикладами зовнішніх ПОМП, які часто є невід'ємним інструментом будь-якої мови програмування загального вжитку є: регулярні вирази, SQL, а також конфігураційні XML-файли для систем керування залежностями.

Внутрішні предметно-орієнтовані мови програмування – це специфічний спосіб використання мови програмування загального вжитку. Код внутрішньої ПОМП є коректним кодом мови програмування загального вжитку, у середині якої використовують дану ПОМП. Тобто, внутрішні ПОМП є підмножиною мови загального вжитку. Яскравим прикладом таких мов є екосистема Ruby. Ruby

володіє потужним та виразним синтаксисом. При розробці програмних системи за допомогою мови програмування Ruby, часто створюють внутрішні ПОМП, більш того, багато бібліотек та фреймворків для Ruby є предметно-орієнтованими мовами програмування. Найпопулярнішим прикладом ПОМП для Ruby є фреймворк Ruby on rails, який можна розглядати як набір різноманітних ПОМП.

Мовні інструменти середовища розробки – це спеціалізоване інтегроване середовище розробки для визначення і побудови предметно-орієнтованої мови програмування. Мовні інструменти середовища розробки дозволяють не лише побудувати предметно-орієнтовану мову програмування, але й надають інтерфейс для написання коду на імплементованій ПОМП.

Предметно-орієнтовану мову програмування також можна сприймати, як спосіб роботи з абстракціями. При розробці програмних систем спочатку будують абстракції для моделювання предметної області, а потім працюють з ними для вирішення поставлених завдань. Найбільш поширений спосіб визначення абстракцій – реалізація бібліотеки чи фреймворку. Найбільш поширений спосіб роботи з цими абстракціями – це виклики API бібліотеки чи фреймворку в основному коді програмної системи. Враховуючи це, предметно-орієнтовану мову програмування також можна вважати інтерфейсом, який надає альтернативний метод маніпуляції API. У такому контексті бібліотеку чи фреймворку можна вважати семантичною моделлю.

Незважаючи на те, що предметно-орієнтовані мови програмування існують доволі довго, їхня межа є нечіткою. Наприклад, регулярні вирази – частина будь-якої сучасної мови програмування загального призначення. Зазвичай, регулярні вирази сприймають як невід’ємну частину мови програмування, а не як окрему предметно-орієнтована мову програмування. Незважаючи на те, що однією з характеристик предметно-орієнтованої мови програмування є предметна

область, ця характеристика не є достатнім елементом для визначення такої мови програмування. Натомість, вирішальним критерієм є обмежена виразність мови. Наприклад, регулярні вирази використовують лише для обробки тексту, жодне інше завдання дана предметно-орієнтована мова програмування вирішити не здатна.

Поширеним методом документування класу, який визначає API, є перелік усіх його відкритих методів. Більш того, назва метода теж повинна описувати призначення метода. Натомість, методи внутрішньої предметно-орієнтованої мови програмування здебільшого мають сенс лише у більших синтаксичних виразах ПОМП. Наприклад, маємо два методи ‘transition’ та ‘to’. Нехай метод ‘transition’ задає стан, у якому може перебувати система, а метод ‘to’ вказує стан, у який система може перейти. Назва ‘to’ мало говорить про семантичне призначення цього методу, але в контексті застосування функція цього метода стає цілком зрозумілою: .transition(state1).to(state2). У результаті такого підходу до назв методів внутрішньої ПОМП, створюють синтаксис який є підмножиною синтаксису мови загального вжитку.

Обмеженість виражальних можливостей для внутрішньої предметно-орієнтованої мови програмування не є фундаментальною властивістю цієї мови, оскільки, мова внутрішньої ПОМП є мовою загального вжитку. У цьому випадку обмеженість виражальних можливостей пов’язані зі способом використання цієї мови. При формуванні виразів внутрішньої предметно-орієнтованої мови програмування програміст обмежує себе невеликою підмножиною мови програмування загального вжитку. Зазвичай, у цьому випадку намагаються уникати синтаксичних конструкцій умов (if-else), циклів та змінних. Внутрішню предметно-орієнтовану мову програмування часто називають суржиком базової мови.

У випадку зовнішньої предметно-орієнтованої мови програмування є чітка межа між зовнішньою ПОМП та мовою програмування загального вжитку. Незважаючи на це, мови програмування можуть бути орієнтованими на предметну область, але залишатися мовами програмування загального вжитку. Яскравим прикладом такого випадку є мова програмування R, яка за своїм призначення є платформою для роботи зі статистикою. Незважаючи на те, що дана мова програмування має чітку предметну область - робота зі статистикою, виражальні можливості R є типовими для мови програмування загального вжитку. Таким чином, не зважаючи на орієнтованість на предметну область, вже ж таки, мову програмування R важко назвати предметно орієнтованою.

Ще однією межею у випадку зовнішніх предметно-орієнтованих мов програмування є структури даних, які можна сереалізувати. Чи є список властивостей з присвоєними значеннями в конфігураційному файлі (pom.xml, package.json) предметно-орієнтованою мовою програмування? У даному випадку граничною умовою є природа такого синтаксису. Списку параметрів зі значення не вистачає гнучкості та виражального синтаксису, щоб відповідати критеріям предметно-орієнтованої мови програмування. Схожі міркування можна застосувати до різноманітних конфігураційних файлів. Багато середовищ розробки та фреймворків дозволяють частину завдань програмування вирішувати через конфігураційні файли, які часто використовують XML. Багато конфігураційних файлів, насправді, є предметно орієнтованими мовами програмування. Однак, це твердження не завжди правдиве. Іноді XML файли створюють за допомогою сторонніх інструментів, тому XML в даному випадку – це метод сереалізації даних, і такі файли не створені для читання програмістами. У цьому випадку, оскільки люди не будуть напряму модифікувати файл, такий формат опису конфігурації не є предметно-орієнтованою мовою програмування. Звичайно, формат сереалізації, який можна прочитати, є корисним, як мінімум,

для налагодження застосунку. Натомість, наскільки формат конфігураційного файлу є зрозумілим для програміста не є критерієм для класифікації такого синтаксису як предметно-орієнтовану мову програмування.

Предметно-орієнтовані мови програмування також поділять на фрагментарні та автономні. Автономні мови програмування зазвичай зберігають в окремих файлах. Для того, щоб зрозуміти сценарій, описаний автономною ПОМП достатньо поглянути на її синтаксис, оскільки базова мова програмування (мова загального вжитку) застосунку або повністю відсутня, або заміщена синтаксисом ПОМП. Іншим способом використання предметно-орієнтованих мов програмування є фрагментарні вставки. У цьому випадку невеликі фрагменти предметно-орієнтованої мови програмування використовують у середині базової мови. Такий тип можна розглядати як вдосконалення, доповнення мови програмування загального вжитку. Тепер щоб зрозуміти сценарій виконання фрагмента предметно-орієнтованої мови програмування необхідно володіти синтаксисом базової мови. Прикладом фрагментарної предметно-орієнтованої мови програмування є регулярні вирази. У програмних проектах зазвичай не існує окремого файлу з регулярними виразами, натомість, невеликі фрагменти регулярних виразів зустрічаються усередині коду написаного на базових мов програмування. Іншим прикладом є SQL, який часто використовують як інструкції для реляційної бази даних в контексті більшої програми. Аналогічні фрагментарні підходи використовують і для внутрішніх предметно-орієнтованих мов програмування. Особливо широке використання фрагментарних внутрішніх мов програмування присутнє в модульному тестуванні. Популярною синтаксичною можливість для фрагментарної ПОМП є анотації, що дозволяють додавати метадані до програмних елементів коду базової мови. Це робить анотації хорошим інструментом для фрагментарних предметно-орієнтованих мов програмування. Варто зазначати, що предметно-орієнтована

мова програмування може бути використана, як автономна, так і фрагментарна. Прикладом такої мови є SQL.

Переваги та недоліки використання предметно-орієнтованих мов програмування

Описавши основні типи та способи використання предметно-орієнтованих мов програмування, розглянемо переваги використання таких мов програмування, та проблеми, які вони вирішують.

Предметно-орієнтовані мови програмування є інструментами з обмеженим застосуванням. Вони не схожі на об’єктно-орієнтовані мови програмування чи методики гнучкої розробки, які внесли фундаментальні зміни в уявлення про те, як треба розробляти програмні продукти. Предметно-орієнтовані мови програмування є вузькоспеціалізованим інструментом для вирішення специфічних проблем. У типовому програмному проекті можуть використовувати декілька предметно-орієнтованих мов програмування в різних місцях, що не є рідкістю. Дуже часто предметно-орієнтована мова програмування є інтерфейсом для використання моделі, що реалізована бібліотекою чи фреймворком. Зважаючи на це перед вибором предметно-орієнтованої мови програмування варто розділяти переваги, які надає сама предметно-орієнтована мова програмування, та переваги, які надає модель, інтерфейс доступу до якої реалізує дана мова.

Найпривабливіше в предметно-орієнтованих мовах програмування є те, що вони надають інструмент для лаконічнішого опису інструкцій програмної системи. Чим простіше читати код, тим простіше знайти в ньому вади, змінювати та підтримувати систему. Таким чином, з тієї ж причини, через яку ми прагнемо

використовувати назви змінних, які описують природу даних, які вони зберігають, документацію і зрозумілі конструкції коду, нам варто заохочувати використання предметно-орієнтованих мов програмування. Часто недооцінюють вплив вад програмного забезпечення на продуктивність, як і коду, так і процесу розробки. Вади не лише значно знижують якість кінцевого продукту, а й сповільнюють процес розробки, змушуючи програмістів витрачати час на знаходження та виправлення помилок в коді. Обмежені виражальні можливості предметно-орієнтованих мов програмування ускладнюють написання коду з помилками та спрощують пошук зроблених помилок. Сама концепції предметно-орієнтованих мов програмування забезпечує ріст продуктивності розробників програмного забезпечення. Вона дозволяє уникнути повторень коду, надає абстракцію, за допомогою якої можна швидше і зрозуміліше описати логіку застосунку. Предметно-орієнтована мова програмування розширює можливості, надаючи зручнішу форму для читання та маніпуляції абстракціями. Предметно-орієнтована мова програмування може допомогти у використанні API, оскільки вона зміщує увагу на те як різні методи повинні комбінуватися. Одним з підходів для цього є створення предметно-орієнтованої мови-обгортки для сторонніх, незручних у використанні бібліотек. ПОМП дозволяє створити більш зручний інтерфейс над моделлю, яку надає бібліотека. Більш того, предметно-орієнтована мова програмування, не повинна реалізовувати весь функціонал сторонньої бібліотека, а лише ту частину, яка необхідна для конкретного проекту.

Одним із найважчих аспектів розробки програмних проектів, і водночас причиною невдачі, є комунікація з експертами предметної області. Надаючи зрозумілу і точно мову для опису предметної області, предметно-орієнтована мова програмування може покращити цю комунікацію. Хоча, дана переваги має більше нюансів, а ніж збільшення продуктивності. Перш за все, багато предметно-орієнтованих мов програмування не підходять для комунікації з

експертами предметної області (наприклад, регулярні вирази). Таку перевагу надає лише частина автономних предметно-орієнтованих мов програмування. По-друге, існує хибне переконання, що використовуючи ПОМП, можна позбутися програмістів, та змусити експертів предметної області писати код програми. Хибне це твердження, тому що було перевірено на прикладі мови програмування COBOL, яка створена для розробки бізнес-застосунків, але тим не менш, експерти бізнес сфери не витіснили програмістів у написанні коду цією мовою програмування. Експерти предметної області не замінять програмістів, але експерти зможуть читати код програм, і таким чином, розуміти які насправді інструкції виконує програма. Маючи змогу читати код предметно-орієнтованої мови програмування, експерти предметної області зможуть виявити помилки та неточності. Вони також зможуть більш ефективно спілкуватися з програмістами, які пишуть інструкції виконання програми, можливо, навіть зможуть написати сценарій програми використовуючи псевдокод, близький до синтаксису предметно-орієнтованої мови програмування, який потім програмісти перетворять на справжній код програми. Таким чином, найбільший виграш від використання предметно-орієнтованої мови програмування команда розробників отримує тоді, коли може заохотити експертів предметної області читати код програми.

Залучення експертів предметної області до робити з ПОМП схоже до залучення експертів для побудови моделі предметної області. Побудова моделі предметної області разом з експертами приносить багато користі, оскільки дозволяє створити єдину мову (ubiquitous language) [12], що є елементом успіху програмного проекту. Предметно-орієнтована мова програмування забезпечує ще один засіб комунікації. Залежно від обставин, експертів можна залучати як і для створення моделі предметної області та самої предметно-орієнтованої мови програмування, так і лише для роботи з предметно-орієнтованою мовою

програмування. Мартін Фавлер (Martin Fowler) вважає, що описувати предметну області за допомогою ПОМП є корисним, навіть якщо дана предметно-орієнтовна мова не буде реалізована на практиці [10]. Це може бути корисним щонайменше у якості платформи комунікації. Експертів предметної області не завжди вдається залучити до розробки предметно-орієнтованої мови програмування, але якщо цього вдасться досягнути, результати виправдає зусилля. Навіть, якщо не вдасться залучити експертів, у будь-якому випадку ріст у продуктивності розробки програмного продукту компенсує ресурси, витрачені на створення предметно-орієнтованої мови програмування.

Іноді предметно-орієнтовану мову програмування використовують через те, що базова мова має певні обмеження, та не дозволяє зручно описати модель. ПОМП дозволяє нам моделювати предметну область в зручному форматі, а потім генерувати код для фактичного середовища виконання застосунку. Ще однією перевагою предметно-орієнтованої мови програмування є те, що вона дозволяє однаковий сценарій виконувати в різних мовних середовищах. Можна описати бізнес-правила, які потім перетворити код в Java, або описати перевірки, які можуть працювати в Java середовищі на сервері і Javascript середовищі на клієнті.

Більшість задач розробки програмно забезпечення вирішують за допомогою імперативної моделі обчислень. Ця модель передбачає, що чітко вказують комп’ютеру, що робити і в якій послідовності; керування потоком виконання програми відбувається за допомогою умовних операторів та циклів, Імперативні моделі обчислення стали популярними, тому що цю модель відносно легко зрозуміти та застосувати до багатьох проблем. Однак це не завжди найкращий вибір. Яскравим прикладом цього є скінченні автомати. Ми можемо написати імперативний код та умови для обробки такого роду поведінку, який також може бути досить добре структурований. Але опис такої математичної моделі, використовуючи терміни та елементи скінченних автоматів є набагато

продуктивніше. Іншим поширеним прикладом є визначення способу побудови програмного забезпечення [14]. Побудувати (build) програмне забезпечення можна використовуючи імперативну модель, натомість більшість людей усвідомлює, що це простіше зробити за допомогою опису мережі залежностей (наприклад, для запуску тестів, скомпільований, виконуваний файл повинен бути актуальними). Як результат, мови, призначені для опису побудови програмного проекту (наприклад, Make або Ant) використовують залежності між завданнями як основний механізм структурування. У літературі часто згадують такі неімперативні підходи, які називають декларативним програмуванням. Поняття декларативного програмування полягає в тому, що цей стиль дозволяє описувати результат, який ми хочемо отримати внаслідок виконання певних інструкцій, а не конкретні сценарії які повинна виконувати обчислювальна машина. Предметно-орієнтована мова програмування не потрібна для використання альтернативної обчислювальної моделі. Основна поведінка альтернативної обчислювальної моделі походить із семантичної моделі. Однак ПОМП має велике значення в цьому контексті, оскільки набагато полегшує програмістам процес маніпулювання декларативними програмами, описуючи семантичну модель.

Проблеми використання предметно-орієнтованих мов програмування

Описавши переваги використання предметно-орієнтованих мов програмування, варто вказати, коли краще їх не використовувати, або принаймні звернути увагу на проблеми, пов’язані з використанням ПОМП.

По суті, єдиною причиною не використовувати предметно-орієнтовані мови програмування є те, що ви не бачите, що будь-яка перевага такої мови програмування є релевантною до вашого випадку, або, щонайменше, ви не

бачите, що вартість побудови ПОМП окупиться в процесі її використання. Навіть коли застосування предметно-орієнтованої мови програмування виправдане, деякі проблеми виникають. Ці проблеми є дещо перебільшені, як правило, тому що програмісти недостатньо знайомі з тим, як будувати предметно-орієнтовані мови програмування, і як ці мови спрощують процес розробки програмного забезпечення.

Багато проблем пов’язаних з використанням предметно-орієнтованих мов програмування є специфічними для одного із конкретних стилів ПОМП, і для розуміння цих проблем необхідно глибше зрозуміти, як впроваджуються ці мови.

Найбільш поширеним запереченням проти використання предметно-орієнтованих мов програмування, є те, що називають проблемою “мовної какофонії”: використання багатьох мов програмування в одному проекті значно ускладнює процес розробки, ніж використання однієї мови. Використання декількох мов ускладнює роботу над системою та ознайомлення зі системою для нових програмістів. Коли говорять про це занепокоєння, зазвичай буває декілька помилкових уявлень. По-перше, часто помилково припускають що зусилля, необхідні для вивчення предметно-орієнтованої мови програмування, співставні із зусиллям вивчення мови загального призначення. Предметно-орієнтовані мови програмування набагато простіші, ніж мови загального призначення, а отже, набагато простіші у вивченні. Багато критиків це розуміють, але все одно заперечують використання ПОМП, оскільки, навіть якщо їх порівняно легко вивчити, наявність багатьох ПОМП ускладнює розуміння того, що відбувається в проекті. Це хибна думка, оскільки проект завжди матиме складні частини, які важко вивчити. Навіть якщо у проекті немає ПОМП, як правило, у вашій кодовій базі, у будь-якому разі, буде багато абстракцій, які розробники повинні розуміти. Зазвичай ці абстракції запаковані у вигляді бібліотек, щоб зробити їх доступними. Навіть якщо розробникам не доведеться вивчати декілька

предметно-орієнтованих мов програмування, усе одно доведеться вивчити кілька бібліотек. Отже, питання важкості вивчення ПОМП полягає в тому, наскільки важче вивчити ПОМП, аніж вивчати базові абстракції, що використовують у програмному проекті. На мою думку, додаткові витрати на вивчення ПОМП досить малі в порівнянні з важкістю розуміння моделі предметної області. Справді, оскільки вся суть предметно-орієнтованих мов програмування полягає в тому, щоб полегшити розуміння та маніпулювання моделлю, наявність ПОМП має зменшити вартість навчання.

Предметно-орієнтована мова програмування може потребувати невеликих ресурсів для її реалізації, але все ж таки це додаткові витрати, додатковий код, який необхідно підтримувати. Як і будь-який код, реалізація предметно-орієнтованої мови програмування збільшує складність проекту. Тим паче, не кожна бібліотека чи абстрактна модель предметної області потребує таку обгортку. Якщо інтерфейс бібліотеки, чи набір абстракцій, реалізованих імперативною мовою програмування, надає зручний та виразний інструмент для маніпуляцій, реалізація ПОМП є не потрібною. Навіть, якщо ПОМП допоможе спростити деякі аспекти роботи з програмним проектом, реалізація такого інструментом буде надлишковою, порівняно з ресурсами, що будуть витрачати на підтримку цієї реалізації. Підтримка предметно-орієнтованої мови програмування є важливим фактором. Навіть найпростіша мова може викликати проблеми, якщо більшості команді розробників важко її зрозуміти. Найбільше складнощів додають зовнішні предметно-орієнтовані мови програмування, оскільки необхідно підтримувати багато елементів цієї реалізації, як-от, парсери чи генератори коду. Одна річ, яка значно девальвує переваги таких мов програмування, це те, що більшість розробників зазвичай не реалізовують ПОМП. Але якщо, все ж таки, впровадити таку практику, то з часом цей аргумент нівелюється. Будь-яка складна предметна область потребує механізму керування

цією складністю. Висновок: якщо предметна область достатньо складна, щоб розглядати побудову предметно-орієнтованої мови для неї, тоді переваги використання такої мови варті ресурсів витрачених на реалізацію предметно-орієнтованої мови програмування.

Ще одну проблему, яка виникає при використанні багатьох мов програмування всередині одного проекту, називають мовою гетто. Припустимо, що певна компанія для реалізації проекту побудувала декілька специфічних мов програмування, які використовується лише всередині цієї компанії. Це значно ускладнює залучення нових програмістів до проекту, а також оновлення інструментів та бібліотек, що використовують для реалізації програмного продукту. Аналізуючи цей аргумент, можна дійти висновку, що якщо ви пишете цілі системи або великі частини певною мовою програмування, це означає, що використана предметно-орієнтована мова програмування, насправді, є мовою загального призначення. Хоча можна використовувати багато методів предметно-орієнтованих мов програмування для побудови мов загального призначення, побудова та підтримка мови загального призначення - це велике завдання, яке потребує колосальних ресурсів. На мою думку, є кілька реальних проблем, пов’язаних із проблемою мовного гетто. Перший з них полягає в тому, що предметно-орієнтована мова програмування завжди має небезпеку випадково перерости в мову загального призначення. Ви берете невелику ПОМП і поступово додаєте нові функції: умовні вирази, цикли, і в кінцевому результаті отримуєте мову загального вжитку. Єдиний захист від цього - це переконатися, що є чітке розуміння, на яку вузьку проблему орієнтована предметно-орієнтована мова програмування. Необхідно піддати сумніву будь-які нові функції, які не стосується обраної спеціалізації. Якщо виникає потреба в розширення функціоналу мови, варто спочатку розглянути використання декількох мов та їх комбінування, замість того, щоб дозволити одній предметно-орієнтованою

мовою програмування стати занадто громіздкою. Ця ж проблема може спіткати і розробників фреймворків. Хороша бібліотека має чітку проблему, яку вона вирішує. Якщо певна бібліотека чи фреймворк обрахунку цін акції включає реалізацію протоколу HTTP, то по суті цей програмний продукт страждаєте від невдалої ізоляції проблеми. Друге питання полягає в тому чи варто реалізовувати певну утиліту. Це стосується як і бібліотек так і предметно-орієнтованих мов програмування. Наприклад, зараз мало причин для побудови власного ORM рішення. Варто керуватися правилом щодо програмного забезпечення, яке полягає в тому, що завжди необхідно прагнути використати готові, добре перевірені програмні рішення. Зокрема, із появою інструментів з відкритим кодом часто має сенс працювати над розширенням існуючих програмних продуктів, ніж писати власні.

Перевага ПОМП полягає в наданні абстракції, яку можна використовувати при роботі з предметною областю. Така абстракція дійсно дуже цінна; вона дозволяє описати поведінку предметної області набагато простіше, ніж при її розгляді в термінах конструкцій низького рівня. Однак будь-яка абстракція, предметно-орієнтована мова або семантична модель, завжди несе в собі небезпеку обмеженого погляду на проблему. При використанні таких нішевих абстракцій ви витрачаєте більше зусиль на зіставлення реальності і абстракції, аніж на вирішення реальних проблем. Найчастіше це проявляється, коли якийсь елемент системи не вписується в абстракцію, багато часу витрачають на те, щоб підігнати реалії під модель, замість того щоб змінити абстракцію так, щоб вона могла впоратися з новою поведінкою. Така проблема характерна для багатьох абстракцій, не тільки для ПОМП, але ПОМП може цю проблему посилити. Оскільки ПОМП забезпечує більш зручний спосіб маніпулювання абстракцією, програмісти неохоче йдуть на внесення будь-яких змін в таку мову. Ситуація ще більше ускладнюється при використанні ПОМП експертами предметної області,

які з ще більшим небажанням ставляться до змін звичної абстракції. Як і в разі будь-якої абстракції, на ПОМП слід дивитися як на щось, що може змінюватися, а не стійку систему.

Скінченні автомати

Якщо взяти дві книги, які мають заголовок «Теорія автоматів», але одна книга написана для дизайнерів апаратного забезпечення, а інша - для людей, що займаються програмним забезпеченням, то складеться враження, що існує дві різні теорії автоматів [6]. Книга апаратних засобів стосується проектування цифрових систем. Книга програмного забезпечення стосується математичного опису обчислень, особливо мов програмування. Обидві використовують концепцію скінченного автомата, як основу своїх методів і теорем, але термінологія та акценти різні. Я буду розглядати скінченні автомати як математичну модель, без прив’язки до будь-якої сфери застосування.

Скінченні автомати – це математична модель, яка слугує певним наближенням до фізичних процесів чи абстрактних подій. Корисність цієї моделі полягає в тому, що вона не прив’язана до конкретної предметної області, натомість, застосовна до широкого класу задач: від психології та менеджменту до комп’ютерної лінгвістики та побудови обчислювальних машин.

Більшість задач, які виникають в науці та інженерії можна поділити на дві категорії [4]: задачі аналізу, де основною метою є передбачення поведінки системи, яку досліджують; задачі синтезу, де основною метою є побудова системи з певною поведінкою. Для двох категорій корисним є поділ змінних системи на три категорії: параметри збурення, параметри реакції та проміжні параметри. Параметр збурення відображає певне значення, що продукує джерело,

яка діє на систему. Параметр реакції – значення, яке продукує система у відповідь на збурення. Проміжний параметр – це значення, яке генерує система внаслідок збурення, що впливає на вихідний результат (параметр реакції). Схематично систему зі скінченною множиною параметрів збурення можна зобразити як:

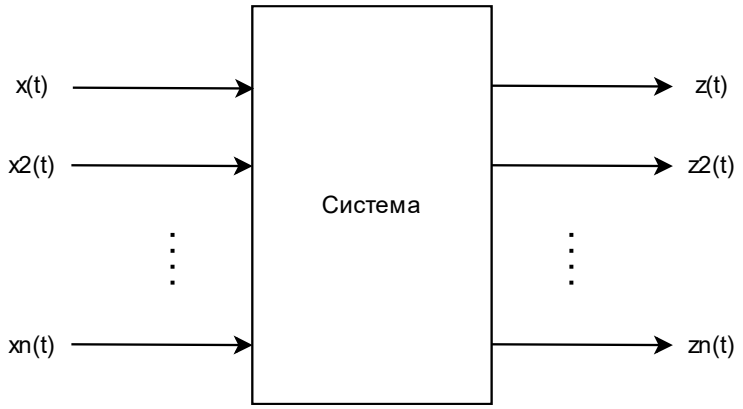

Рисунок (1.1)

У даному випадку система – це чорна скринька, де параметри збурення напрямлені в середину, а вихідні параметри – назовні.

Кожна система, яка використовує модель скінченних автоматів, передбачає, що джерело параметрів збурення є синхронним. Параметри, які надходять до системи підкоряються певному дискретному закону розподілу. Як наслідок, відповідь на збурення (зміна стану) відбувається в конкретний момент часу, який можна відокремити. Момент часу позначимо t_v $(v = 1, 2 \dots n)$. Більш того, ще одним припущення є те, що поведінка системи в будь-який момент часу t_v не залежить від інтервалу (t_v, t_{v+1}). Таким чином, незалежним параметром, який можна використовувати для аналізу системи є впорядковане значення інтервалу часу t, тобто v. Варто зазначити, що припущення, зазначені вище, не передбачають, що інтервали часу t_v є однаковими; чи що параметри цієї системи

мають однаковий вплив на зміну стану (наприклад, не змінюють стан системи). Єдиним наслідком цих припущень є те, що внутрішні параметри системи не залежать від значення інтервалу t, а лише від значення v.

Означення 1.2: системи, що приймають вхідні параметри в дискретний час та підпорядковуються вище описаним припущенням називають синхронними.

Означення 1.3: системи, що приймають вхідні параметри в дискретний час та не підпорядковуються вище описаним припущенням називають асинхронними.

На практиці, багато системи, які є асинхронними, для спрощення аналізу вважають синхронними. Це справедливо, оскільки в реальному житті між подіями можна визначити Δt, яке близьке до нуля, але не дорівнює нулю. Як приклад, розглянемо перемикач, яким можна увімкнути чи вимкнути лампу. Вхідними параметрами системи буде позиція перемикача: (вимкнений, увімкнутий), вихідними параметрами – (лампа світиться, лампа згасла). Якщо змінювати стан перемикача повільно, то, здається, що стан системи не залежить від інтервалу часу t_v. Якщо змінювати положення перемикача доволі швидко, то лампа не згасатиме (або не вмикатиметься). Таким чином, дана система є асинхронною за своєю реакцією на потік вхідних параметрів. Незважаючи на це, при аналізі комп'ютерних систем використовують синхронні моделі.

Наступним припущенням для моделі скінченних автоматів є скінченність множини значень для вхідних, проміжних та вихідних параметрів системи. Множину значень, які може набувати v називають v – абеткою, та позначають V. Елементи множити V називають $v-$символом. Нехай $K^1, K^2, \dots K^m$ скінченні множини, які містять елементи $k_i^1, k_i^2, \dots k_i^m$ відповідно, тоді декартовий добуток

цих множин $K^1 \times K^2 \times \ldots \times K^m$ – усі впорядковані $m-$ кортежі $(k_i^1, k_i^2, \ldots k_i^m)$. Якщо вхідні параметри системи $x^1, x^2, \ldots x^u$, тоді вхідна абетка:

$$X = X^1 \times X^2 \times \ldots \times X^u$$

Формула (1.1)

Де $X^i, i = 1, 2, \ldots . , u$ – абетка вхідних параметрів x^i. Аналогічно, якщо $z^1, z^2, \ldots z^w$, абетку вихідних параметрів позначають Z і визначають як декартовий добуток:

$$Z = Z^1 \times Z^2 \times \ldots \times Z^w$$

Формула (1.2)

де $Z^i, j = 1, 2, \ldots , w$ – абетка вихідних параметрів z^i. Нехай потужність множин X^i та Z^i , дорівнює p_i та q_j відповідно. Тоді потужність множин X та Z:

$$\mathrm{p} = \prod_{i=1}^{u} p_i$$

Формула (1.3)

$$q = \prod_{j=1}^{w} p_j$$

Формула (1.4)

Величини p та q є скінченними.

З формули (1.1) бачимо, що достатньо одного елементу, для того, щоб описати усі вхідні параметри u у певний момент часу t_v. Аналогічно, з формули

(1.1), достатньо одного елементу множини Z, щоб описати усі вихідні параметри w в певний момент часу t_v. Звідси маємо, що вхідні параметри $x^1, x^2, \dots x^u$ можна замінити на параметр x, абетка якого визначена у формулі (1.1); вихідні параметри $z^1, z^2, \dots z^w$ можна замінити на параметр z, абетка якого визначена у формулі (1.2). У результаті отримаємо схематичне представлення, яке використовують для позначення моделі скінченних автоматів найчастіше.

Рисунок (1.2)

Вхідні та вихідні параметри системи є елементами, які можна оцінити та виміряти, в той час, як проміжні змінні мають неявний характер, і їх значення часто є невимірним. Проте, важливість проміжних змінних проявляється їхнім впливом на вихідні параметри системи. Цей вплив називають станом системи. Стан системи в певний час t_v позначатимемо s_v. Множину станів, у яких може перебувати система, позначатимемо S.

Концепцію стану можна чітко описати за допомогою ролі, яку він виграє всередині моделі скінченних автоматів. Цю роль можна описати двома пунктами:

- Вихідний параметр в певний момент часу можна однозначно визначити використовуючи вхідний символ та стан системи в цей момент часу.
- Стан системи в наступний момент часу можна однозначно визначити використовуючи вхідний параметр та стан в теперішній момент часу.

Як наслідок, стан моделі скінченних автоматів в будь-який момент часу t_v – це змінна, за допомогою якої, разом з вхідним параметром в цей момент часу можна визначити вихідні параметри, та стан в наступний момент часу.

Розглянувши усі основні елементи моделі скінченних автоматів можемо дати формальне визначення.

Означення (1.4): скінченний автомат M – це синхронна система зі скінченною абеткою вхідних параметрів $X = \{x_1, x_2, \dots, x_p\}$, скінченною абеткою вихідних параметрів $Z = \{z_1, z_2, \dots, z_q\}$, початковим станом та парою функцій f_1 та f_2: $z_v = f_1(x_v, s_v)$, $s_{v+1} = f2(x_v, s_v)$, де x_v, z_v, s_v – вхідний параметр, вихідний параметр та стан скінченного автомату M в момент часу $t_v, (v = 1, 2, \dots)$.

Частковий випадок скінченного автомату при $f_1(x_v, s_v) = f_2(x_v)$ називають тривіальним скінченним автоматом. При цьому, проміжні значення системи не мають жодного впливу на вихідні параметри, а концепція стану є надлишковою. Скінченний автомат називають не тривіальним, якщо $f_1(x_v, s_v) \neq f_2(x_v)$.

Використання скінченних автоматів для моделювання предметної області.

Тепер розглянемо приклад того, як описану математичну модель скінченних автоматів можна використати для моделювання частини програмного продукту. На рисунку 1.3 показано UML-діаграму, що моделює скінченний автомат для програмного коду, що відповідає за обробку замовлень інтернет-магазину. В даному випадку можна виділити сім станів: створено, очікує підтвердження, сплачено, відправлено, доставлено, скасовано та додано в архів.

Напрямлені відрізки задають можливі переходи між станами, а назви відрізків є подіями, настання яких необхідне для переходу з одного стану в інший.

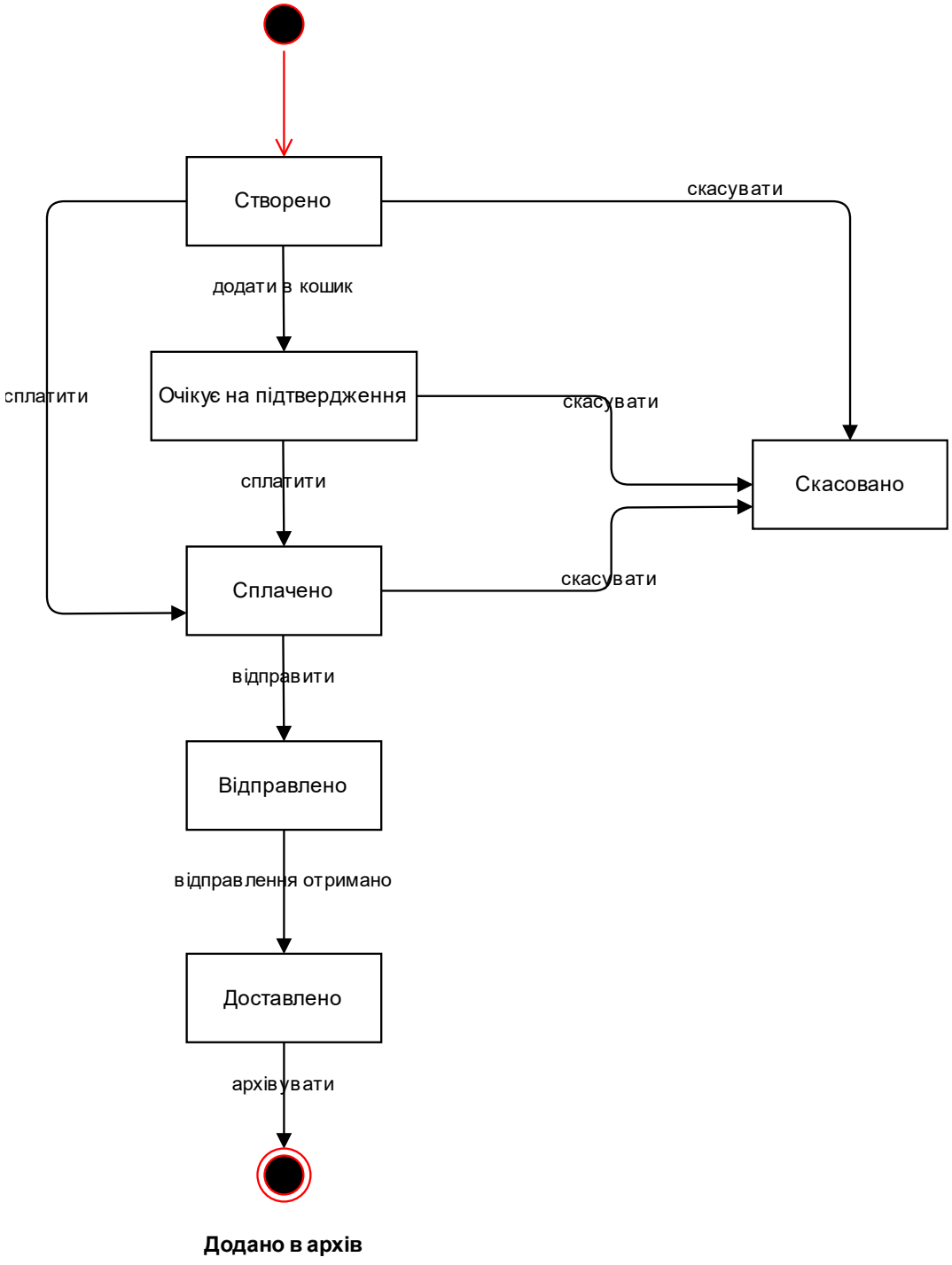

Рисунок 1.3

Розділ 2. Взірці реалізації скінченних автоматів використовуючи імперативні мови програмування

Представлення моделі скінченного автомату за допомогою UML діаграми та таблиці переходів

Розглянемо простий приклад застосування моделі скінченного автомату. Маємо програмну систему, яка керує турнікетом. Турнікет має два стани: ‘Locked’, ‘Unlocked’. Коли турнікет перебуває в стані ‘Locked’, можна кинути монетку в спеціальний слот, і турнікет перейде в стан ‘Unlocked’. Приклад такої моделі

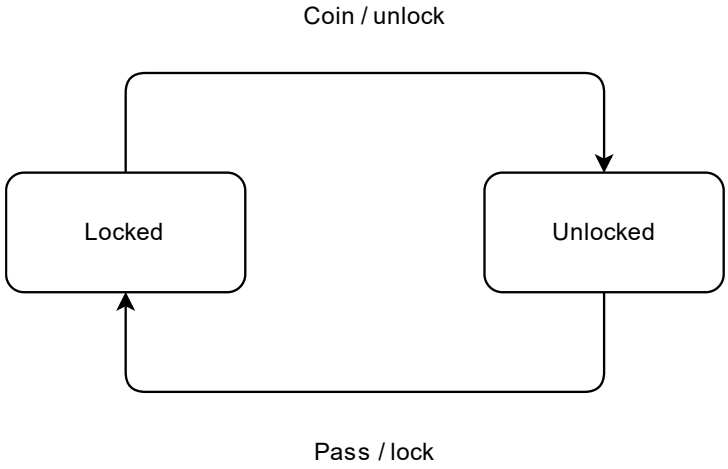

Рисунок 2.1

показано на рисунку 2.1. Ламана на цій діаграмі – це перехід з одного стану в інший. Назва ламаної складається з двох частин, розділена косою лінією: перша частина є назвою події (вхідним параметром), друга частина – це назва дії (методу, функції), яку виконує система при зміні стану. Більш формально цю систему можна інтерпретувати наступними твердженням:

- Якщо турнікет перебуває в стані ‘Locked’, і отримує на вхід подію ‘Coin’, тоді турнікет переходить в стан ‘Unlocked’ та викликає дію ‘unlock’.
- Якщо турнікет перебуває в стані ‘Unlocked’, і отримує на вхід подію ‘Pass’, тоді турнікет переходить в стан ‘Locked’ та викликає дію ‘lock’.

Ускладнимо систему, додавши декілька сценаріїв користування турнікетом. Щоб пройти крізь турнікет, необхідно спочатку кинути у слот монетку. Ця дія переводить турнікет у стан ‘Unlocked’. Потім, якщо хтось проходить крізь турнікет, генерується подія ‘Pass’, і система переходить в стан ‘Locked’. Припустимо, що хтось намагається пройти крізь турнікет, попередньо не вкинувши монетку в слот. У цьому випадку система повинна увімкнути звуковий сигнал. Інший сценарій користування турнікетом є дещо аномальним. Припустимо в слот кинули монетку, і система перебуває у стані ‘Unlocked’. Після цього в слот кидають ще одну монетку. У цьому випадку система просто виведе на екран ‘thank you’. На рисунку 2.2 зображено модель турнікету з

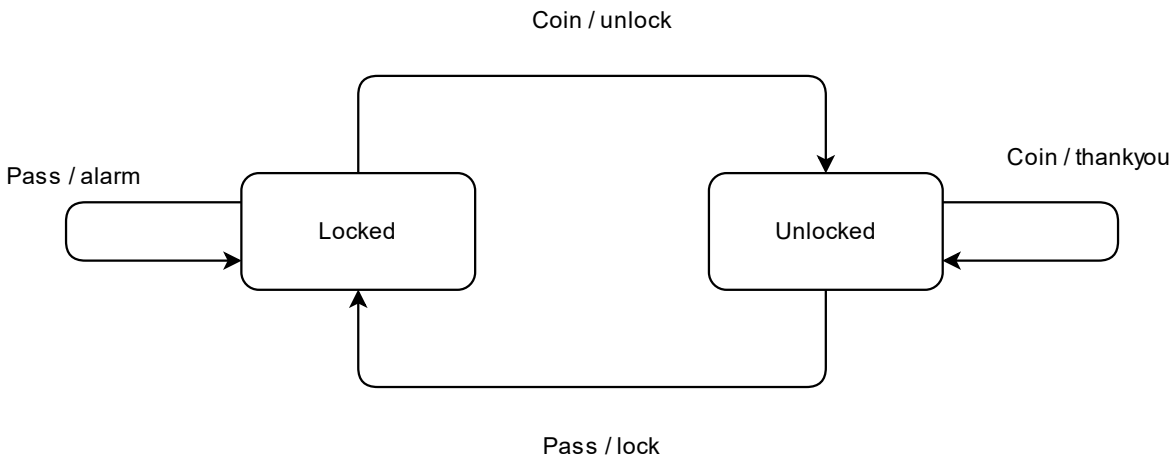

Рисунок 2.2

доповненими станами. Нові сценарії можна інтерпретувати наступним чином:

- Якщо турнікет перебуває в стані ‘Locked’, і отримує на вхід подію ‘Pass’, тоді турнікет залишається в стані ‘Locked’ та викликає дію alarm.
- Якщо турнікет перебуває в стані ‘Unlocked’, і отримує на вхід подію ‘Coin’, тоді турнікет залишається в стані ‘Unlocked’ та викликає дію ‘unlock’.

Більш компактно скінченний автомат можна подати у вигляді таблиці переходів, де перша колонка – це стан, у якому перебуває система, друга – подія або вхідний параметр, який отримав скінченний автомат, третя – стан у який перейде система, четверта – дія, яка виконається при переході в наступний стан. У лістингу 2.1 побудовано таку таблицю переходів, яка повністю відповідає діаграмі скінченного автомату, зображеного на рисунку 2.2.

```
syntax_base.txt
1 CURRENT STATE | EVENT | NEXT STATE | ACTION¬
2 ¬
3 Locked          Coin    Unlocked     unlock¬
4 Locked          Pass    Locked       alarm¬
5 Unlocked        Coin    Unlocked     thankYou¬
6 Unlocked        Pass    Locked       lock¬
```

Лістинг 2.1

Усього за допомогою 16 слів описано модель скінченного автомату для програмної системи, що керує турнікетом. Таке лаконічне подання і буде основою синтаксису предметно-орієнтованої мови програмування для скінченних автоматів.

Наступним кроком я розгляну взірці (патерни), за допомогою яких, можна запрограмувати або згенерувати модель скінченного автомату в імперативних мовах програмування загального призначення.

Вкладений switch-case.

Найпростішим способом реалізації моделі скінченного автомату є вкладений switch-case. У лістингу 2.2 надано приклад реалізації скінченного автомату для системи турнікету на мові Java. Верхній switch-case описує стани, у яких перебуває система. Вкладений switch-case описує, як повинен змінюватися стан та які дії необхідно виконати при переході з одного стану в інший, залежно від події, яку система отримала вхідним параметром.

```
public class TurnstileFSM {
  public enum Event {COIN, PASS}
  private enum State {LOCKED, UNLOCKED}
  private State state = State.LOCKED;
  private Turnstile turnstile;

  public TurnstileFSM(Turnstile turnstile) {
    this.turnistile = turnstile;
  }

  public void handleEvent(Event event) {
    switch (case) {
      case LOCKED:
        switch (event) {
          case COIN:
            state = State.UNLOCKED; turnstile.unlock(); break;
          case PASS:
            turnstile.alarm(); break;
        }
        break;
      case UNLOCKED:
        switch (event) {
          case COIN:
            turnstile.thankYou(); break;
          case PASS:
            state = State.LOCKED; turnstile.lock(); break;
        }
        break;
    }
  }
}
```

Лістинг 2.2

Верхній switch-case будемо називати switch-case-ом стану, оскільки він описує, як повинна змінюватися система, залежно від стану, у якому вона зараз перебуває. Відповідно, вкладений switch-case називатимемо switch-case-ом подій, оскільки вкладений switch-case описує, як повинен змінюватися стан, та які дії необхідно виконати, залежно від події, яку на вхід отримав скінченний автомат. Клас ‘Turnstile’ є контейнером методів, які описують зміни, що виконуватимуться залежно від теперішнього стану системи, та події яка була

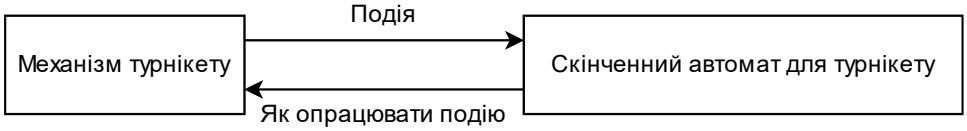

Рисунок 2.3

подана на вхід. Таким чином використання скінченних автоматів для моделювання предметної області, надає ефективний інструмент для реалізації важливого принципу розробки програмного коду – розподіл обов'язків (separation of concern). Частина коду, яка відповідає за обробку події (клас TurnstileFSM) описує що необхідно робити при настанні певної події. Контейнер методів містить деталі реалізації, та описує як саме необхідно опрацювати подію. Такий підхід дозволяє провести чітку межу в програмному коді між тим, що необхідно робити, та тим як це робити.

Серед переваг використання вкладених switch-case можна виділити наступні: простота реалізації, швидкість виконання коду. Зважаючи на це вкладений switch-case підхід можна використовувати в низькорівневому програмуванні, з обмеженими обчислювальним ресурсом та пам'яттю. У лістингу 2.2 наведено частину коду, що використовує такий підхід, але даний приклад містить всього два стани та дві події. Якщо ж кількість станів та подій зростає до сотень, код, що використовує вкладений switch-case стає важко читати та змінювати. Тому вкладений switch-case для реалізації моделі скінченного автомату варто використовувати, якщо: надзвичайно важливою є швидкість виконання програми, кількість станів та подій є невеликою, або ж якщо з кодом такої реалізації не будуть безпосередньо взаємодіяти програмісти.

Таблиця переходів (Transition table).

Ще одним простим підходом до реалізації моделі скінченного автомату є таблиця переходів. Приклад такої реалізації наведено у лістингу 2.3. Код є схожим до того, який реалізує підхід з вкладеним switch-case. У класі ‘TurnstileFSM’ оголошено два ‘enum’-типи: один описує стани, інший – події. Змінна ‘state’ зберігає поточний стан системи, а змінна типу ‘Turnstile’ зберігає указник на клас-контейнер, що містить методи, які описують деталі реалізації переходів між станами. Також, оголошено внутрішній клас ‘Transition’, що є структурою даних для опису переходу між станами. Клас ‘Transition’ зберігає теперішній стан системи, подію, стан, у який система повинна перейти, та дію, яку необхідно виконати при переході, у вигляді лямбда-функції.

```
public class TurnstileFSM {
  public enum Event {COIN, PASS}
  private enum State {LOCKED, UNLOCKED}
  private State state = State.LOCKED;
  private Turnstile turnstile;

  public TurnstileFSM(Turnstile turnstile) {
    this.turnistile = turnstile;
  }

  class Transition {
    public State currentState;
    public Event event;
    public State newState;
    public Consumer<Turnstile> action;

    Transition(State currentState, Event event, State newState, Consumer<Turnstile> action) {
      this.currentState = currentState;
      this.event = event;
      this.newState = newState;
      this.action = action;
    }
  }

  Transition[] transitions = new Transition[]{
    new Transition(State.LOCKED, Event.COIN, State.UNLOCKED, t -> t.unlock()),
    new Transition(State.LOCKED, Event.PASS, State.LOCKED, t -> t.alarm()),
    new Transition(State.UNLOCKED, Event.COIN, State.UNLOCKED, t -> t.thankYou()),
    new Transition(State.UNLOCKED, Event.PASS, State.LOCKED, t -> t.lock())
  };

  public void handleEvent(Event event) {
    for (Transition transition : transitions) {
      if (transition.currentState == state && t.event == event) {
        state = transition.newState;
        transition.action.accept(turnstile);
        break;
      }
    }
  }
}
```

Лістинг 2.3

Змінна ‘transition’ містить таблицю переходів, що повністю описує модель скінченного автомату для системи турнікету. На кожну подію викликають метод ‘handleEvent’, який ітерує таблицю переходів, шукаючи елементи, який описує перехід з теперішнього стану для події, яку передано аргументом в метод ‘handleEvent’. Після цього викликають метод, який описує деталі реалізації переходу.

Такий підхід реалізації моделі скінченних автоматів легко читати, і як наслідок модифікувати та додавати нові стани навіть для систем з великою кількістю події та станів. Якщо частину коду, яка реалізовує скінченний автомат будуть часто змінювати, варто використовувати підхід таблиці переходів. Попри це, код реалізації таблиці переходів є значно повільнішим, аніж вкладений switch-case, оскільки метод ‘handleEvent’ використовує лінійний пошук, складність якого O(n), де n – це кількість елементів в таблиці. Звичайно, можна використати хеш-таблицю як структуру даних для зберігання таблиці переходів, але в будь-якому разів, підхід з використанням вкладених switch-case буде значно швидшим у виконанні, аніж таблиця переходів. У випадку, коли час виконання коду є надзвичайно важливим, таблицю переходів як взірець реалізації скінченного автомату використовувати не варто.

Взірець стану (State pattern).

Підхід до реалізації моделі скінченних автоматів за допомогою вкладених switch-case є надзвичайно швидким, але такий підхід важко підтримувати та

читати, особливо якщо скінченний автомат має багато станів. Реалізації з використанням таблиці переходів легко читати, розуміти та підтримувати навіть для скінченного автомату з багатьма станами, натомість швидкість виконання такої програми є значно повільнішою порівнюючи з switch-case методом. Використання взірця стану є компромісом між цими двома підходами. Взірець стану дозволяє сконструювати програмний код таким чином, що він буде достатньо швидко виконуватися (не так швидко як switch-case, але значно швидше, аніж таблиця переходів), та буде достатньо зрозумілим і легким в підтримці навіть для великих систем.

Взірець стану – це поведінковий взірець, що дозволяє змінювати поведінку об’єкту внаслідок зміни внутрішнього стану [5]. Для реалізації такого механізму використовують поліморфізм та делегацію. Взірець стану використовують, коли поведінка залежить від даних, які цей об’єкт зберігає, та коли необхідно змінити поведінку об’єкту під час виконання програми. На рисунку 2.4 показано UML діаграму взірця стану. Клас Context

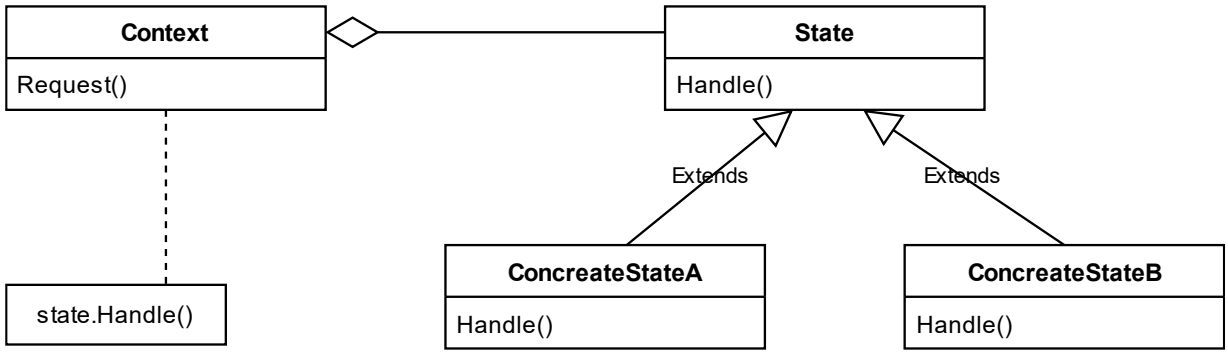

Рисунок 2.4

зберігає посилання (агрегує) певну реалізацію класу State. Клас ‘State’ визначає інтерфейс, а також інкапсулює поведінку для певного стану контексту. ‘ConcreateStateA’ та ‘ConcreateStateB’ реалізує поведінку, що відповідає конкретному стану. ‘Context’ делегує реалізацію поведінки, що властива об’єкту в конкретних станах. Іноді класам ‘ConcreateStateA’, ‘ConcreteStateB’ передають посилання на об’єкт ‘Context’, щоб вони мали доступ до даних, які зберігаються в класі ‘Context’. Такий підхід дозволяє локалізувати деталі реалізації поведінки об’єкта для кожного стану, і таким чином, для того, щоб додати новий стан не потрібно модифікувати існуючі класи, а лише додати новий клас, який реалізує поведінку для нового стану. Це дозволяє також реалізувати open closed принцип дизайну коду, який говорить, що програмний код повинен надавати можливість розширення без модифікації існуючого коду.

На рисунку 2.5 зображено UML-діаграму класів, що репрезентують модель скінченних автоматів, використовуючи взірець стану. Змінна ‘state’ містить посилання на певний об’єкт стану, у якому зараз перебуває система. Клас ‘TurnstileFSM’ відповідає за опис скінченного автомату. Два відкриті методи ‘coin’ та ‘pass’ репрезентують події в системі. Ці методи будуть викликати відповідні методи класів, що реалізують поведінку для конкретних станів. Методи ‘thanks’, ‘unlock’, ‘alarm’, ‘lock’ будуть викликатися при переходах між станами.

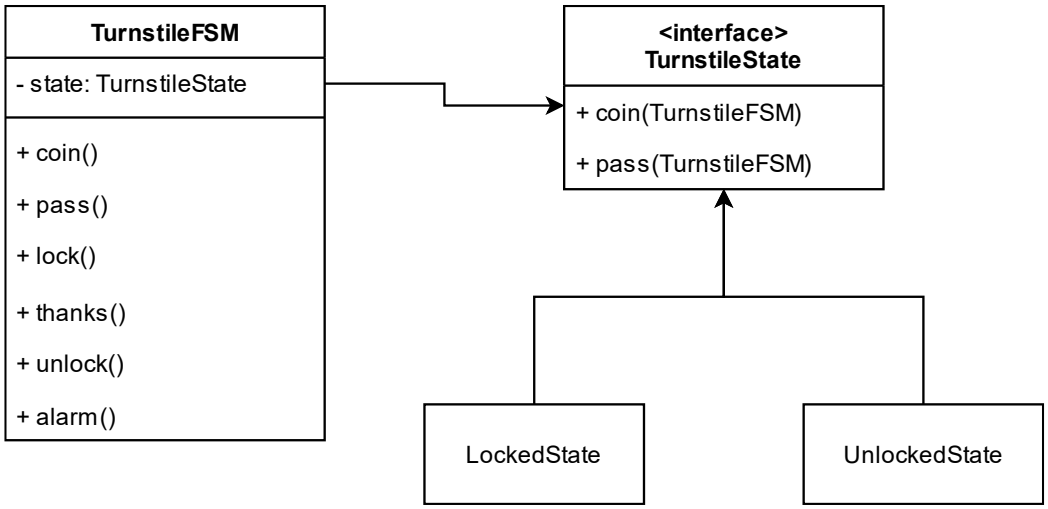

Рисунок 2.5

Інтерфейс ‘TurnstileFSM’ складається з двох методів: ‘coin’ та ‘pass’, які на вхід приймають ‘TurnstileFSM’, для того, щоб мати змогу змінити стан та викликати відповідні методи при переході з одного стану в інший. ‘LockedState’ та ‘UnlockedState’ класи реалізують інтерфейс ‘TurnstileState’. Розглянемо приклад переходу між станами. Припустимо, що система перебуває у стані ‘Locked’, тоді змінна ‘state’ класу ‘TurnstileFSM’ зберігає указник на об’єкт типу LockedState. Нехай система отримує подію ‘Coin’. У результаті, метод ‘coin’ класу ‘TurnstileFSM’ буде викликано, який виконає метод ‘coin’ класу ‘LockedState’, передавши указник на себе. Метод ‘coin’ класу ‘LockedState’ змінить присвоїть змінній ‘state’ об’єкту ‘TurnstileFMS’ об’єкт типу ‘UnlockedState’, та виконає метод ‘unlock’.

За часовою складністю реалізація скінченного автомату за допомогою скінченного автомату є близькою до реалізації за допомогою вкладених switch-case. Оскільки класи, що інкапсулюють опис поведінки системи для конкретних станів не містять жодних даних, тому достатньо лише одного екземпляру класу

для кожного стану системи, який можна зберігати як статичну сталу змінну в ‘TurnstileState’ інтерфейсі. Читати та модифікувати код, який реалізовує такий підхід є значно зручніше, навіть для систем з великою кількістю станів та подій, у порівнянні з switch-case методом. Хоча, таблиця переходів є легшою у читанні та розумінні коду, порівняно з взірцем стану.

Синтаксис предметно-орієнтованої мови програмування для скінченних автоматів

Основою синтаксису предметно-орієнтованої мови програмування для скінченних автоматів буде представлення скінченного автомату за допомогою таблиці переходів. Таку таблицю переходів будемо записувати між фігурними дужками. Над таблицею переходів будемо вказувати наступні значення: ‘FSM’, ‘Actions’, ‘Initial’. ‘FSM’ – це назва класу, структуру чи функції (залежно від мови, у яку генерують код), яка реалізовуватиме скінченний автомат, логіку зміни станів при настанні подій. ‘Actions’ – це назва класу контейнера методів, що описуватимуть деталі реалізації переходів. ‘Initial’ – це початковий стан системи.

```
Initial: Locked
Actions: Turnstile
FSM: TurnstileFSM
{
  Locked   Coin    Unlocked    unlock
  Locked   Pass    Locked      alarm
  Unlocked Coin    Unlocked    thankyou
  Unlocked Pass    Locked      lock
}
```

Лістинг 2.4

Опис таблиці можна спростити, надавши можливість групувати переходи за початковими станами, таким чином уникаючи повторень. Після назви початкового стану надамо можливість вказувати групу переходів між фігурними дужками, як це показано у лістингу 2.5

```
Initial: Locked
Actions: Turnstile
FSM: TurnstileFSM
{
  Locked    {
    Coin    Unlocked    unlock
    Pass    Locked      alarm
  }
  Unlocked  {
    Coin    Unlocked    thankyou
    Pass    Locked      lock
  }
}`
```

Лістинг 2.5

Для деяких переходів стан залишиться незмінним після певної події. Щоб не вказувати стан повторно, дозволимо вказувати ‘*’, якщо стан не повинен змінитися.

```
Initial: Locked
Actions: Turnstile
FSM: TurnstileFSM
{
  Locked    {
    Coin    Unlocked    unlock
    Pass    *           alarm
  }
  Unlocked  {
    Coin    *           thankyou
    Pass    Locked      lock
  }
}`
```

Лістинг 2.6

Модель скінченного автомату для системи турнікету не враховує стан, у якому перебуває система, коли сигналізує про порушення. Назвемо цей стан ‘Alarming’. Для того, щоб припинити сигнал необхідно вкинути монетку або стороннє втручання. Коли сигнал вимикають, буде створена подія ‘Reset’, і система перейде в стан ‘Locked’. Для цього випадку додамо можливість вказувати декілька дій, які необхідно виконати при переході. Декілька подій можна вказати всередині фігурних дужок. Синтаксис такого скінченного автомату показано в лістингу 2.7. Додавши подію ‘Reset’, необхідно вказати рядок обробки цієї події для кожного стану. Для того, щоб уникнути повторень, додамо абстрактний стан, який можна використовувати, як батьківський для інших станів. Назву абстрактного класу записуватимемо в круглих дужках.

```
Initial: Locked
Actions: Turnstile
FSM: TurnstileFSM
{
  Locked    {
    Coin    Unlocked    unlock
    Pass    *           alarm
    Reset   *           {lock alarmOff}
  }

  Alarming  Reset Locked {lock alarmOff}

  Unlocked  {
    Coin    *           thankyou
    Pass    Locked      lock
    Reset   Locked      {lock alarmOff}
  }
}
```

Лістинг 2.7

Наслідування записуватимемо за допомогою двокрапки. Система не може перебувати в абстрактному стані. Абстрактний стан використовують лише для наслідування. Також для стану ‘Alarming’ зручніше вказати події, що відбуватимуться при переході в цей стан, та виході з цього стану, у визначенні самого стану, аніж при описі стану з якого можна потрапити в стан ‘Alarming’. Будемо задавати вхідну подію за допомогою символа ‘>’ та вихідну подію за допомогою ‘<’. Кінцевий синтаксис предметно-орієнтованої мови програмування подано в лістингу 2.8.

```
Initial: Locked
Actions: Turnstile
FSM: TurnstileFSM
{
  (Resetable) Reset Locked lock

  Locked : Resetable {
    Coin    Unlocked    unlock
    Pass    *           alarm
  }

  Alarming : Resetable >alarmOn <alarmOff {
    * * *
  }

  Unlocked : Resetable {
    Coin    *           thankyou
    Pass    Locked      lock
  }
}
```

Лістинг 2.8

Тепер необхідно представити описаний синтаксис у формальній формі. Для цього я використаю BNF (Backus Naur Form). У лістингу 2.9 наведено формальну специфікацію, використовуючи BNF мета-синтаксис, для зовнішньої предметно-орієнтованої мови програмування для скінченних автоматів, описану вище.

Визначена предметно-орієнтована мова програмування позначена як ‘<FSM>’, і складається з одного або більше хедерів та логіки переходів. Хедер складається з двох частин: назви та значення. Назва має три можливих значення: ‘Actions’, ‘FSM’ та ‘Initial’. Таблиця переходів позначена як ‘<logic>’ і є одним або декілька переходів, оточених фігурними дужками. Перехід (‘<transition>’) – це специфікація стану, у якому перебуває система та решта опису переходу, або специфікація стану та декілька описів переходу між фігурними дужками. Друга

частина переходу (<subtransition>) складається з специфікації події (<event-spec>), наступного стану (<next-state>) та специфікації дій, які необхідно виконати при переході (<action-spec>).

```
<FSM> ::= <header>* <logic>
<header> ::= "Actions:" <name> | "FSM:" <name> | "Initial:" <name>

<logic> ::= "{" <transition>* "}"

<transition> ::= <state-spec> <subtransition>
              |   <state-spec> "{" <subtransition>* "}"

<subtransition>   ::= <event-spec> <next-state> <action-spec>
<action-spec>     ::= <action> | "{" <action>* "}" | "*"
<state-spec>      ::= <state> <state-modifiers>
<state-modifiers> ::= "" | <state-modifier> | <state-modifier> <state-modifiers>
<state-modifier>  ::= ":" <state>
                    |   "<" <action-spec>
                    |   ">" <action-spec>

<next-state> ::= <state> | "*"
<event-spec> :: <event> | "*"
<action> ::= <name>
<state> ::= <name>
<event> ::= <name>
```

Лістинг 2.9

‘<action-spec>’ – це назва події або декілька назв події між фігурними дужками, або термінальний символ ‘*’. ‘<state-spec>’ – це назва стану та ‘<state-modifiers>’. ‘<state-modifiers>’ складається з порожнього термінального символу, або одного <state-modifier>, або декількох <state-modifier>. ‘<state-modifier>’ складається з термінального символу ‘:’, який позначає наслідування, та назва абстрактного стану від якого наслідують; або термінальний символ ‘<’ та ‘<action-spec>’; або термінальний символ ‘>’ та ‘<action-spec>’. ‘<next-state>’ складається з стану або термінального символу ‘*’. ‘<event-spec>’ складається з <event> або термінального символу ‘*’. ‘<action>’, ‘<state>’, ‘<event>’ – це назви відповідних

елементів таблиці переходів. Позначення ‘<name>’ в BNF вважають термінальним.

Розділ 3. Реалізація компілятора предметно-орієнтованої мови програмування для скінченних автоматів

Структура компілятора предметно-орієнтованої мови програмування для скінченних автоматів.

Для описаного синтаксису в лістингу 2.9 я реалізував компілятор, який генерує код на мові програмування загального вжитку. Такий підхід до реалізації предметно-орієнтованої мови програмування дозволяє уникнути залежності від сторонніх бібліотек, чи надлишкових абстракцій у середовищі виконання програми.

Для реалізація компілятора я обрав мову Golang. Причинами вибору саме цієї мови програмування є наступні: швидкість виконання програм написаних на цій мові; статична типізації; набір інструментів для тестування є частиною дистрибутива Golang; Golang компілюється в один виконуваний файл для трьох платформ (Gnu/Linux, OSX, Windows), та не потребує додаткового оточення для запуску програм.

Структура компілятора зовнішньої предметно-орієнтованої мови для скінченного автомата подана на рисунку 3.1.

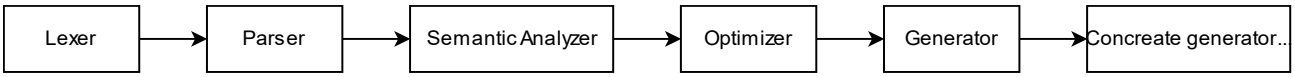

Рисунок 3.1

Лексер – це частина компілятора, як перетворює вхідний код на потік лексем, які є подіями для Парсера.

Для реалізації парсера можна обрати генератори парсера, наприклад, ANTLR або уасс. Я вирішив не використовувати такий підхід, щоб не залучати жодних сторонніх залежностей в програму-компілятор. Парсер я реалізував як скінченний автомат, що відповідає формальному визначенню синтаксису за допомогою BNF.

Семантичний аналізатор перевіряє побудовану парсером синтаксичну структуру даних на наявність семантичних помилок визначеного скінченного автомату, та генерує семантичну структуру даних, якщо жодних помилок не було виявлено.

Оптимізатор перетворює семантичну структуру даних в просту таблицю переходів, видаляючи весь синтаксичний цукор, на кшталт, абстрактних станів, символу ‘*’ чи згрупованих переходів для одного стану.

Генератор перетворює структуру даних, згенеровану оптимізатором, на набір окремих структур, що моделюють взірець вкладених switch-case без прив’язки до будь-якої мови програмування.

Реалізатор генератора коду використовує взірець відвідувач для генерації кінцевого коду конкретної мови програмування.

Реалізація лексера.

Перший крок у парсингу мови – це перетворення потоку вхідних символів на потік токенів, що репрезентують синтаксис мови програмування – цей процес називають лексичним аналізом. Частину компілятора, що відповідає за лексичний аналіз, називають лексером. Спочатку визначимо список усіх токенів, які запишемо як константи в пакеті ‘tokens’. Пакет ‘tokens’ містить усі

термінальні одиниці синтаксису, описані у BNF поданні предметно-орієнтованої мови для скінченних автоматів.

t/token.go+

```
1  package tokens¬
2  ¬
3  const (¬
4  ··EOF          = "EOF"¬
5  ··NAME         = "NAME"¬
6  ··OPEN_PAREN   = "("¬
7  ··CLOSE_PAREN  = ")"¬
8  ··OPEN_BRACE   = "{"¬
9  ··CLOSE_BRACE  = "}"¬
10 ··ENTRY_STATE  = "<"¬
11 ··EXIT_STATE   = ">"¬
12 ··STAR         = "*"¬
13 ··COLON        = ":"¬
14 )¬
```

Лістинг 3.1

Тепер визначимо структуру ‘Lexer’, яка зберігатиме рядок та номер символу, який зараз опрацьовується лексером, а також поле ‘collector’, яке зберігатиме структуру, що реалізує інтерфейс ‘TokenCollector’. Даний інтерфейс описує методи, що будуть викликатися, коли лексер натрапить на певний токен. Кожен токен має відповідний метод. Такий підхід є взірцем програмування ‘Builder’. Інтерфейс в мові Golang реалізовують неявно. Інтерфейс ‘TokenCollector’ реалізує парсер, через який поступову будемо будувати синтаксичну структуру даних.

l/tokencollector.go

```
1  package lexer¬
2  ¬
3  type TokenCollector interface {¬
4  ··OpenBrace(lineNumber, position int)¬
5  ··CloseBrace(lineNumber, position int)¬
6  ··OpenParen(lineNumber, position int)¬
7  ··CloseParen(lineNumber, position int)¬
8  ··OpenAngle(lineNumber, position int)¬
9  ··CloseAngle(lineNumber, position int)¬
10 ··Star(lineNumber, position int)¬
11 ··Colon(lineNumber, position int)¬
12 ··Name(name string, lineNumber, position int)¬
13 ··Error(lineNumber, position int)¬
14 }¬
```

Лістинг 3.3

Процес лексичного аналізу починається у відкритому методі ‘Lex’, який на вхід приймає вхідний код програми. Далі порядкову перевіряється код на наявність одно символьних токенів чи імен, пропускаючи порожні символи. Якщо не знайдено ні імені, ні одно символьного токена, створюється лексична помилка з номером рядка та символу, який не вдалося розпізнати. Метод ‘findSingleCharacter’ відповідає за аналіз одно символьних токенів, як-от: двокрапка, круглі та фігурні дужки. Якщо такий токен вдалося знайти, викликається відповідний, метод структури, що реалізовує інтерфейс ‘TokenCollector’. Для пошуку імені я використовую регулярні вирази.

Важливим аспектом розробки компілятора є тестування. Інтерфейс ‘TokenCollector’ реалізує парсер, тому написати тести для лексера без додаткових зусиль не вдасться. Щоб провести тестування ізольовано, та перевірити, як працює механізм лексичного аналізу, я імплементував додаткову структуру та набір метод для неї, що реалізують інтерфейс ‘TokenCollector’. Структура ‘TestCollector’ зберігає стрічкове поле tokens. Кожного разу коли лексер,

розпізнає певну лексему та викликає відповідний метод на структурі ‘TestCollector’, до значення поля tokens додається назва токена.

l/lexer.go l/testcollector.go l/lexer_test.go

```
type Lexer struct {
    lineNumber   int
    readPosition int
    collector    TokenCollector
}

func New(collector TokenCollector) *Lexer {
    lexer := &Lexer{collector: collector}
    return lexer
}

func (lexer *Lexer) Lex(source string) {
    lexer.lineNumber = 1
    lines := strings.Split(source, "\n")
    for _, line := range lines {
        lexer.lexLine(line)
        lexer.lineNumber++
    }
}

func (lexer *Lexer) lexLine(line string) {
    for lexer.readPosition = 0; lexer.readPosition < len(line); {
        lexer.lexToken(line)
    }
}

func (lexer *Lexer) lexToken(line string) {
    if !lexer.findToken(line) {
        lexer.collector.Error(lexer.lineNumber, lexer.readPosition+1)
        lexer.readPosition++
    }
}

func (lexer *Lexer) findToken(line string) bool {
    return lexer.findSkipSpace(line) || lexer.findSingleCharacterToken(line) || lexer.findName(line)
}
```

Лістинг 3.2

Так як, тестування лексера передбачає багато однотипних тестів, я використав підхід таблиць тестування, у якому визначають масив структур, що містить усі необхідні дані для того, щоб виконати тест. Приклад такої структури наведено в лістингу 3.5. Протестовано одно символьні токени, імена, порожні символи, а також лексичні помилки, коли вжито символ, що не є синтаксичним елементом предметно-орієнтованої мови програмування.

Вдалося досягти стовідсоткове покриття тестами лексера. Результат тестування наведено у лістингу 3.6.

l/testcollector.go

```
package lexer

type TestCollector struct {
    tokens     string
    firstToken bool
}

func NewTestCollector() *TestCollector {
    return &TestCollector{firstToken: true}
}

func (collector *TestCollector) addToken(token string) {
    if !collector.firstToken {
        collector.tokens += ","
    }
    collector.tokens += token
    collector.firstToken = false
}

func (collector *TestCollector) OpenBrace(lineNumber int, position int) {
    collector.addToken("openBrace")
}

func (collector *TestCollector) CloseBrace(lineNumber int, position int) {
    collector.addToken("closeBrace")
}

func (collector *TestCollector) OpenParen(lineNumber int, position int) {
    collector.addToken("openParen")
}
```

Лістинг 3.4

```
=== RUN   TestSingleTokens
--- PASS: TestSingleTokens (0.00s)
=== RUN   TestComments
--- PASS: TestComments (0.00s)
=== RUN   TestIntegration
--- PASS: TestIntegration (0.00s)
PASS
coverage: 100.0% of statements
ok      github.com/larkvincer/dsl-fsm/lexer     0.003s
```

Лістинг 3.5

```
func TestSingleTokens(t *testing.T) {
	testTable := []testInputs{
		{input: "{", want: "openBrace"},
		{input: "}", want: "closeBrace"},
		{input: "(", want: "openParen"},
		{input: ")", want: "closeParen"},
		{input: "<", want: "openAngle"},
		{input: ">", want: "closeAngle"},
		{input: "*", want: "star"},
		{input: ":", want: "colon"},
		{input: "mystate", want: "#mystate#"},
		{input: "state_with_numbers_222", want: "#state_with_numbers_222#"},
		{input: " ", want: ""},
		{input: " \r  \t\n", want: ""},
		{input: " \r  \t *\n", want: "star"},
		{input: ".", want: "error"},
	}

	runLexerTestTable(testTable, t)
}
```

Лістинг 3.6

Реалізація парсера.

Парсер перетворює потік токенів, згенерованих лексером, в синтаксичну структуру даних, часто в абстрактне синтаксичне дерево. В основі реалізації парсера лежить скінченний автомат, оскільки для того, щоб реалізувати парсер для мови програмування достатньо імплементувати скінченний автомат, що задає цю мову програмування. На рисунку 3.1 наведено діаграму переходів, що описує частину скінченного автомата для парсера. Якщо порівняти з лістингом 2.9, на якому визначено синтаксис предеметно-орієнтованої мови програмування для скінченних автоматів в BNF, можна побачити багато схожих елементів. У якості взірця реалізації скінченного автомату я обрав таблицю переходів,

оскільки для мене є важливим легко читати та модифікувати код даного скінченного автомату.

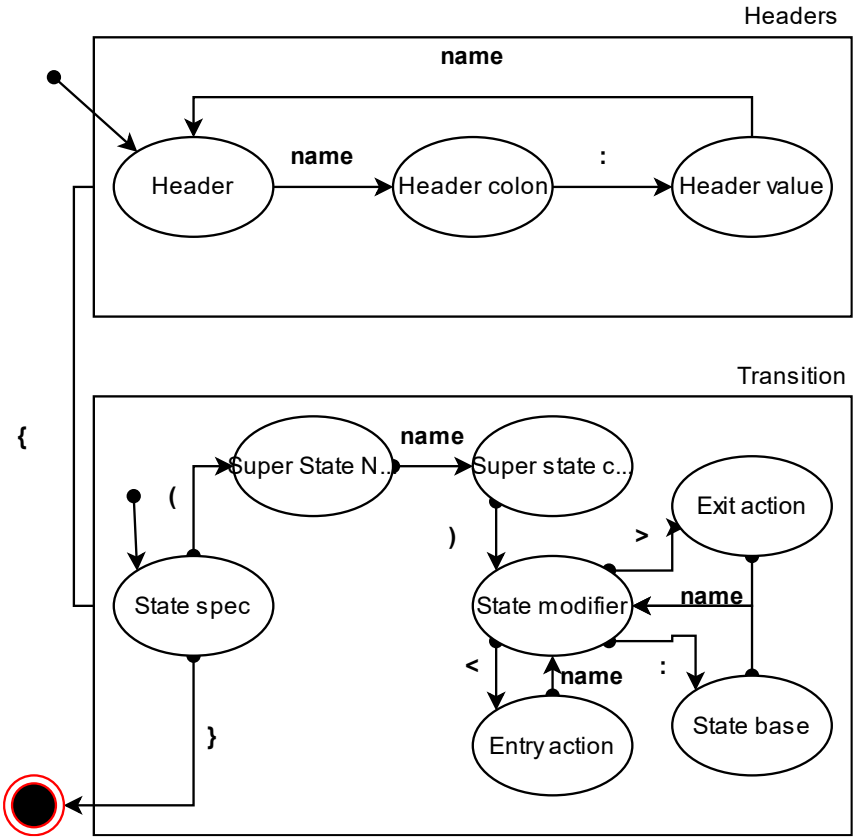

Рисунок 3.1

Синтаксичні структури даних, що зберігають предметно-орієнтовану мову програмування визначено у файлі ‘fsmsyntax.go’. ‘FsmSyntax’ зберігає програму написану використовуючи предметно-орієнтовану мову програмування для скінченних автоматів, у вигляді масиву типу ‘[]Header’; набору переходів, що зберігаються у змінній ‘Logic’, яка є масивом указників на структуру ‘FsmTransition’; масиву синтаксичних помилок та змінної ‘Done’, яка вказує чи процес парсингу закінчено. ‘FsmTransition’ складається з двох полів: ‘State’, що є ‘stateSpec’ структурою, та ‘SubTransitions’, що є масивом структур ‘SubTransition’, які описують подію, яку необхідно опрацювати в стані ‘State’,

стан, у який потрібно перейти та масив методів, які треба виконати при переході. Структура ‘stateSpec’ складається з імені стану, масиву батьківських станів, вхідних та вихідних методів, а також змінної ‘AbstractState’, яка є бульовим значенням, що позначає чи даний стан є абстрактним.

```
type FsmSyntax struct {
	Headers []Header
	Logic   []*FsmTransition
	Errors  []syntaxError
	Done    bool
}

type Header struct {
	Name  string
	Value string
}
```

Лістинг 3.7

У файлі ‘parser.go’ визначено алгоритм парсингу. Структура ‘Parser’ реалізує інтерфейс лексера ‘TokenCollector’, та складається з поточного стану, у якому перебуває парсер, змінної, що містить указник на структуру, що реалізує інтерфейс ‘SyntaxBuilder’, та масиву переходів, що є таблицею переходів для скінченного автомата парсера. Інтерфейс ‘SyntaxBuilder’, який показано на лістингу 3.8, визначає набір методів, що будуть виконуватися при переходах парсера з одного стану в інший. Усі можливі стани, у яких може перебувати парсер визначені як константи у пакеті ‘states’. Подіями скінченного автомату парсера є токени, які генерує лексер, та виконує відповідний метод інтерфейсу ‘TokenCollector’, що реалізований парсером. Методи ‘TokenCollector’, реалізовані парсером, викликають метод ‘HandleEvent’, передавши відповідний токен, номер рядка, та символу. Метод ‘HandleEvent’, знаходить в таблиці

переходів стан у який парсер повинен перейти, та викликає відповідний метод. Якщо такого методу не існую, буде згенеровану синтаксичну помилку.

```
type SyntaxBuilder interface {
--newHeaderWithName()
--addHeaderWithValue()
--setStateName()
--done()
--setSuperStateName()
--setEvent()
--setNullEvent()
--setEntryAction()
--setExitAction()
--setStateBase()
--setNextState()
--setNullNextState()
--transitionWithAction()
--transitionNullAction()
--addAction()
--transitionWithActions()
--headerError(state, event string, lineNumber, position int)
--stateSpecError(state, event string, lineNumber, position int)
--transitionError(state, event string, lineNumber, position int)
--transitionGroupError(state, event string, lineNumber, position int)
--endError(state, event string, lineNumber, position int)
--syntaxError(lineNumber, position int)
--setName(name string)
}
```

Лістинг 3.8

За обробку помилок відповідає метод ‘handleEventError’. Умовно синтаксичні помилки в даному випадку можна поділити на 5 категорій: помилки опису хедера, помилки стану, помилки переходів, помилки опису групи переходів, та синтаксичні помилки, пов’язані з використанням символів, що не є частиною абетки визначеної предметно-орієнтованої мови програмування. Формат помилки наступний: тип помилки, стан у якому перебував парсер під час виникнення помилки, подія (токен), який було отримано на вхід, а також номер рядка та символу який не вдалося розпарсити.

```
1  package errortypes¬
2  ¬
3  const (¬
4  ··HEADER           = "HEADER"¬
5  ··STATE            = "STATE"¬
6  ··TRANSITION       = "TRANSITION"¬
7  ··TRANSITION_GROUP = "TRANSITION_GROUP"¬
8  ··END              = "END"¬
9  ··SYNTAX           = "SYNTAX"¬
10 )¬
```

Лістинг 3.9

Для тестування парсера можна виділити два основні підходи: подавати на вхід частину вихідного коду програми, а потім перевіряти синтаксичну структуру даних на кількість та наявність необхідних елементів; реалізувати стрічкове представлення синтаксичної структури даних, яке буде ідентичним до вхідного коду (за винятком форматування), та порівнювати його з очікуваним значенням. Я обрав другий підхід. Як і у випадку з лексером, при тестуванні парсера присутні багато однотипних тестів, тому я використовував таблиці тестувань. Тести можна розділити на чотири категорії: тестування хедера, тестування таблиці переходів (тіла програми), тестування правильності появи синтаксичних помилок та інтеграційні тести. Вдалося покрити більше дев'яноста відсотків коду парсера.

```
→  parser git:(master) ✗ go test --cover
PASS
coverage: 91.5% of statements
ok      github.com/larkvincer/dsl-fsm/parser    0.007s
```

Лістинг 3.10

Реалізація семантичного аналізатора.

Лексер та парсер називають фронтендом компілятора [9]. Парсер генерує перше внутрішнє представлення синтаксису мови програмування. Починаючи з даного підрозділу, ми будемо будувати бекенд компілятора.

Однією з переваг зовнішніх предметно-орієнтованих мов програмування над внутрішніми є наявність етапу компіляції. Під час етапу компіляції можна провести перевірку на наявність семантичних помилок, що можуть бути причиною невизначеної поведінки чи помилок виконання програми.

Реалізований семантичний аналізатор знаходить 11 типів помилок, а також здатен згенерувати одне попередження, а саме:

- Відсутність назви скінченного автомату.
- Відсутність назви класу контейнеру з набором методів.
- Невідомий хедер.
- Повторна ініціалізація хедеру.
- Перехід в невизначений стан.
- Невизначений батьківський стан.
- Стан не використовується.
- Повторне визначення рядка переходу.
- Використання абстрактного стану, як наступного.
- Назва стану і назва абстрактного стану однакові – попередження.
- Повторне визначення методу в рядку переходу.
- Декілька батьківських станів визначають різні переходи для однакової події.

Основний алгоритм семантичного аналізу описано у відкритому методі ‘Analyze’. Спочатку створюється нова структура семантичного представлення скінченного автомату, значення якої поступову заповнюють виклики методів ‘analyzeHeaders’, ‘checkSemanticValidity’ та ‘produceSemanticStateMachine’. ‘produceSemanticStateMachine’ також перевіряє правильність спадкування усіх станів, використовуючи структуру та її методи ‘superClassCrawler’.

```
27 func (sa *SemanticAnalyzer) Analyze(fsmSyntax *parser.FsmSyntax) *SemanticStateMachine {
28 ··sa.semanticStateMachine = NewSemanticStateMachine()¬
29 ··sa.analyzeHeaders(fsmSyntax)¬
30 ··sa.checkSemanticValidity(fsmSyntax)¬
31 ··sa.produceSemanticStateMachine(fsmSyntax)¬
32 ¬
33 ··return sa.semanticStateMachine¬
34 }¬
```

Лістинг 3.11

Результатом семантичного аналізу є нове внутрішнє представлення скінченного автомату, для якого використано структуру, що наведено в лістингу 3.12.

```
 8 type SemanticStateMachine struct {¬
 9 ··Errors       []AnalysisError¬
10 ··Warnings     []AnalysisError¬
11 ··States       map[string]*SemanticState¬
12 ··Events       map[string]bool¬
13 ··Actions      map[string]bool¬
14 ··InitialState SemanticState¬
15 ··ActionClass  string¬
16 ··FsmName      string¬
17 }¬
```

Лістинг 3.12

Для тестування також використано таблиці тестування. Тести, що перевіряють правильність появи помилок, аналізують вміст поля ‘Errors’ структури ‘SemanticAnalyzer’ на предмет наявності відповідних помилок. Тести, що перевіряють відповідність згенерованої семантичної структури даних синтаксичній структурі даних, використовують стрічкове подання семантичної структури даних.

```
→  semanticanalyzer git:(master) x go test --cover
PASS
coverage: 99.6% of statements
ok      github.com/larkvincer/dsl-fsm/semanticanalyzer  0.015s
```

Лістинг 3.13

Семантичний аналізатор покритий тестами практично на сто відсотків.

Реалізація оптимізатора.

У даному випадку завдання оптимізатора – це позбутися усього синтаксичного цукру, на кшталт абстрактних станів, та представити скінченний автомат у вигляді простого масиву переходів.

Оптимізатор приймає на вхід семантичну структуру даних, згенеровану семантичним аналізатором. Відкритий метод ‘Optimize’ описує алгоритм побудови структури ‘OptimizedStateMachine’, визначення якої наведено в лістингу 3.14. Спочатку зчитуються значення хедерів, та створюють структуру ‘Header’, яка їх зберігає. Потім метод ‘addList’ додає списки усіх станів, подій та

дій, ітеруючи семантичну структуру даних. Наступним етапом є побудова таблиці переходів. Саме на цьому етапі видаляються абстрактні стани, усі наслідування приводять в лінійний формат, вхідні та вихідні методи стають просто методами, при чому зберігається порядок виклику.

```
type Optimizer struct {
--optimizedStateMachine OptimizedStateMachine
--semanticStateMachine  semanticanalyzer.SemanticStateMachine
}

func Optimize(ast semanticanalyzer.SemanticStateMachine) *OptimizedStateMachine {
--optimizer := Optimizer{
----semanticStateMachine:  ast,
----optimizedStateMachine: OptimizedStateMachine{},
--}

--optimizer.addHeader(ast)
--optimizer.addLists()
--optimizer.addTransitions()

--return &optimizer.optimizedStateMachine
}
```

Лістинг 3.13

```
type OptimizedStateMachine struct {
--States      []string
--Events      []string
--Actions     []string
--Header      Header
--Transitions []Transition
}
```

Лістинг 3.14

Реалізація генератора коду.

Останнім етапом компіляції предметно-орієнтованої мови програмування для скінченних автоматів є генерації коду для конкретної мови програмування. Оскільки, метою є створення компілятора, який з легкістю можна розширити для генерації коду для будь-якої мови програмування, архітектура модуля генератора коду повинна строго дотримуватися принципу ‘Open-closed’, який стверджує, що код програми повинен надавати механізм розширення функціоналу без модифікації існуючого коду [12].

Перед генерацією коду необхідно обрати взірець, який буде використано для організації коду, що генерують. Для цього я обрав підхід з використанням вкладених switch-case. Під час вибору було враховано наступні критерії: легкість реалізації, швидкість виконання коду та складність підтримки. Оскільки підтримувати та читати необхідно не згенерований код, а код предметно-орієнтованої мови програмування, то останні критерієм можна знехтувати. Найшвидшим у виконанні, а також простим у реалізації є підхід з використанням вкладених switch-case – саме цей взірець було обрану для організації коду, що генерують.

Основний алгоритм генерації коду описано у відкритому методі ‘Generate’ структури ‘NSCGenerator’. Файл ‘nscnode.go’ містить структури що моделюють усі вузли коду, що генерують. Інтерфейс ‘NSCNodeVisitor’ описує структуру, яку необхідно реалізувати, для генерації вузлів коду для конкретної мови програмування. Такий підхід називають взірцем ‘Відвідувач’ (pattern ‘Visitor’). Використання взірця програмування ‘Відвідувач’ дозволяє нам відокремити основний алгоритм генерації коду від деталей реалізації для конкретної мови програмування. У файлі ‘codegenerator.go’ оголошено інтерфейс

‘LanguageCodeGenerator’. Він задає набір службових методів, що реалізують запис згенерованого коду, та створюють структуру, що відповідає за генерацію вузлів коду для конкретної мови програмування. Таким чином, щоб згенерувати необхідно реалізувати лише два інтерфейси ‘NSCNodeVisitor’, ‘LanguageCodeGenerator’.

```
func main() {
	const sourceInput = `Initial: Locked
Actions: Turnstile
FSM: TurnstileFSM
{
  Locked    {
    Coin    Unlocked    unlock
    Pass    Locked      alarm
  }
  Unlocked  {
    Coin    Unlocked    thankyou
    Pass    Locked      lock
  }
}`
	syntaxBuilder := parser.NewFsmSyntaxBuilder()
	parser := parser.NewParser(syntaxBuilder)
	lexer := lexer.New(parser)
	lexer.Lex(sourceInput)
	parser.HandleEvent("EOF", -1, -1)
	fsm := syntaxBuilder.GetFSM()

	if len(fsm.Errors) == 0 {
		semanticStateMachine := semanticanalyzer.New().Analyze(fsm)
		optimizedStateMachine := optimizer.Optimize(*semanticStateMachine)
		flags := make(map[string]string)
		flags["package"] = "firsttry"
		javaImplementor := implementors.NewJavaNestedSwitchCaseImplementor(flags)
		javaCodeGenerator := generator.NewJavaCodeGenerator(javaImplementor)
		codeGenerator := generator.NewCodeGenerator(optimizedStateMachine, javaCodeGenerator)
		codeGenerator.Generate()
	} else {
		fmt.Print(fsm.Errors)
	}
}
```

Лістинг 3.15

У лістингу 3.15 наведено приклад запуску усіх компонентів компілятора для генерації коду на мові програмування ‘Java’. Результат генерації показано в лістингу 3.16. Після генерації, залишилося лише реалізувати методи, що виконуватимуться при переході.

```
→  code git:(master) go run main.go
Generated code:
package firsttry;
public abstract class TurnstileFSM {
public abstract void unhandledTransition(String state, String event);
private enum State {Locked,Unlocked}
private enum Event {Coin,Pass}
private State state = State.Locked;
private void setState(State s) { state = s; }
public void Coin() {handleEvent(Event.Coin);}
public void Pass() {handleEvent(Event.Pass);}
private void handleEvent(Event event) {
switch(state) {
case Locked:
switch(event) {
case Coin:
setState(State.Unlocked);
unlock();
break;
case Pass:
setState(State.Locked);
alarm();
break;
default: unhandledTransition(state.name(), event.name()); break;
}
break;
case Unlocked:
switch(event) {
case Coin:
setState(State.Unlocked);
thankyou();
break;
case Pass:
setState(State.Locked);
lock();
break;
default: unhandledTransition(state.name(), event.name()); break;
}
break;
}
}
}
```

Лістинг 3.16

Висновок

У даній роботі було розглянуто предметно-орієнтовані мови програмування. Дано означення, наведено визначальні критерії, типи та різницю між предметно-орієнтованими мовами програмування і мовами програмування загального призначення. Описано переваги та недоліки використання предметно-орієнтованих мов програмування. Також розглянуто математичну модель скінченного автомату, та приклади застосування цієї моделі для опису предметної області.

Виведено синтаксис предметно-орієнтованої мови програмування для скінченних автоматів у форматі BNF. В основі синтаксису лежить представлення скінченного автомату у вигляді таблиці переходів. Описано взірці реалізації моделі скінченного автомату, та наведено приклади такої реалізації. Розглянуто переваги та недоліки кожного взірця.

У практичній частині реалізовано програму-компілятор для визначеного синтаксису, що компілює предметно-орієнтовану мову програмування в код мови програмування загального призначення. Імплементовано юніт та інтеграційне тестування для всіх компонентів компілятора, окрім генератора. Наявність семантичного аналізатора надає можливість запобігти багатьом помилкам у проектуванні скінченного автомату. Генератор спроектовано таким чином, щоб дотриматися принципу ‘open-close’, та надати можливість легко розширювати кількість мов програмування, у які можна проводити компіляцію, без модифікації існуючого коду. Генератор кінцевого коду було реалізовано лише для мови програмування ‘Java’.

Під час реалізації виникало багато проблем з коректним представлення скінченного автомату у вигляді синтаксичної структури даних. Більшість з них вдалось виправити у процесі обширного тестування.

Щодо подальшого розвитку, для початку, варто збільшити кількість інтеграційних тестів та покращити алгоритми парсингу і семантичного аналізу, проводячи перевірку на більшу кількість помилок. Також слід збільшити кількість мов програмування, у які можна компілювати предметно-орієнтовану мову програмування для скінченних автоматів, реалізувавши відповідні генератори.

Список літератури

1. Kenneth H. P. Finite State Based Models and Applications / H. Rosen Ph.D. Kenneth. – London: A CHAPMAN & HALL BOOK.
2. G. Goos. Lecture Notes in Computer Science / G. Goos, J. Hartmanis, J. van Leeuwen. – Santa Barbara, CA, USA, 2003.
3. Nikos H. Implementation of Workflows as Finite State Machines in a National Doctoral Dissertations Archive / H. Nikos, D. Zavaliadis, K. Stamatis.
4. Gill A. Introduction to the Theory of Finite-State Machines / Arthur Gill. – New York, 1962.
5. Design Patterns : Elements of Reusable Object-oriented Software / [E. Gamma, R. Helm, R. Johnson та ін.]., 1997.
6. Wagner F. Modeling Software with Finite State Machines / Ferdinand Wagner., 2006.
7. Finite State Machines: A Model of Behaviour for C++ [Електронний ресурс] – Режим доступу до ресурсу: http://objectvalue.com/articles/CppFiniteStateMachine.html.
8. John C. M. Introduction to Languages and The Theory of Computation / Martin John C.., 2003. – (4).
9. Keith L. Engineering a compiler-Morgan Kaufmann / L. Keith, T. D Cooper., 2004.
10. Fowler M. Domain-Specific Languages / Martin Fowler., 2010. – (Addison-Wesley Professional).
11. Making a Pratt Parser Generator [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://www.robertjacobson.dev/designing-a-pratt-parser-generator.

12. Martin R. Agile Software Development, Principles, Patterns, and Practices / Rober C. Martin.
13. Thorsten B. Writing An Interpreter In Go / Ball Thorsten., 2018.
14. Markus Voelter. DSL Engineering / Markus Voelter.