ДОДАТОК А

(додатковий)

ФРАГМЕНТИ КОДУ ПРОГРАМИ

```
const handlebars = require("handlebars");

class Template {
  constructor(schema: string) {
    this.schema = Handlebars.compile(schema);
  }

  getSchema(data: any) {
    return this.schema(data);
  }
}

class ModelFactory<T, U> {
  constructor(template: Template) {
    this.template = template;
  }

  buildArtifact(value: T): U {
    return new Artifact(this.template.getSchema(value));
  }
}

class ArtifactGenerator {
  static generate(factory: AbstractArtifactFactory, spec: any) {
    return factory.buildArtifact(spec);
  }
}

const code = `
describe value Note:
- title is string;
- text is string;
end
`;

const sokEntityToTs = {
  value: name => `class ${name} {}`
}

const sokTypeToTsType = {
  string: 'string'
};

function parseSok(source) {
```

```
  const statements =
source.trim().split(/describe(.+)end/).filter(Boolean);
  for (let statement of statements) {
    const operations = statement.split('\n');
    operations.pop();
    const [entityDescriptor, ...attributesDescriptos] = operations;
    const [_, type, name] = entityDescriptor.slice(0, -1).split('
');
    const castToTsType = sokEntityToTs[type];
    const cls = castToTsType ? castToTsType(name) : null;
    const attributes = attributesDescriptos.map(descr => {
      const [name, type] = descr.slice(2, -1).split(' is ');
      const targetType = `${sokTypeToTsType[type]}`;
      const tsField = targetType ? `${name}: ${targetType};` :
`${name};`;
      return tsField;
    });
    const fullClass = cls.replace('{}', `{${attributes.join('')}}`);
    console.log(fullClass);
  }
}

parseSok(code);
```

```
{
  "name": "acg",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "parse": "node index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "handlebars": "^4.7.6"
  }
}
```

ДОДАТОК Б

(обов’язковий)

СЛАЙДИ ПРЕЗЕНТАЦІЇ

Рисунок Б.1 – Слайд №1

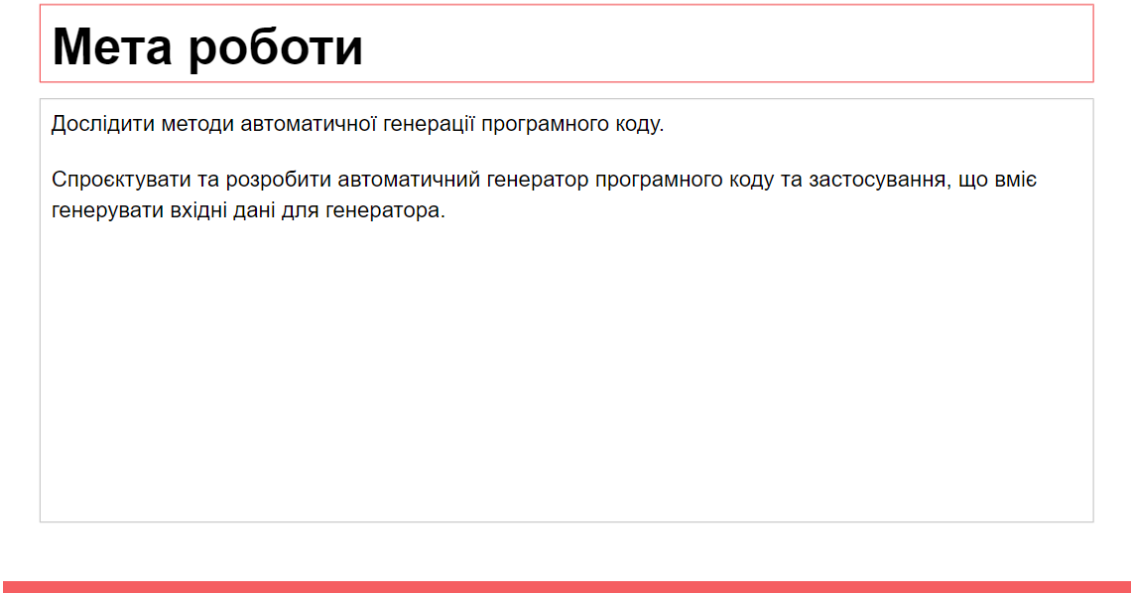


Рисунок Б.2 – Слайд №2

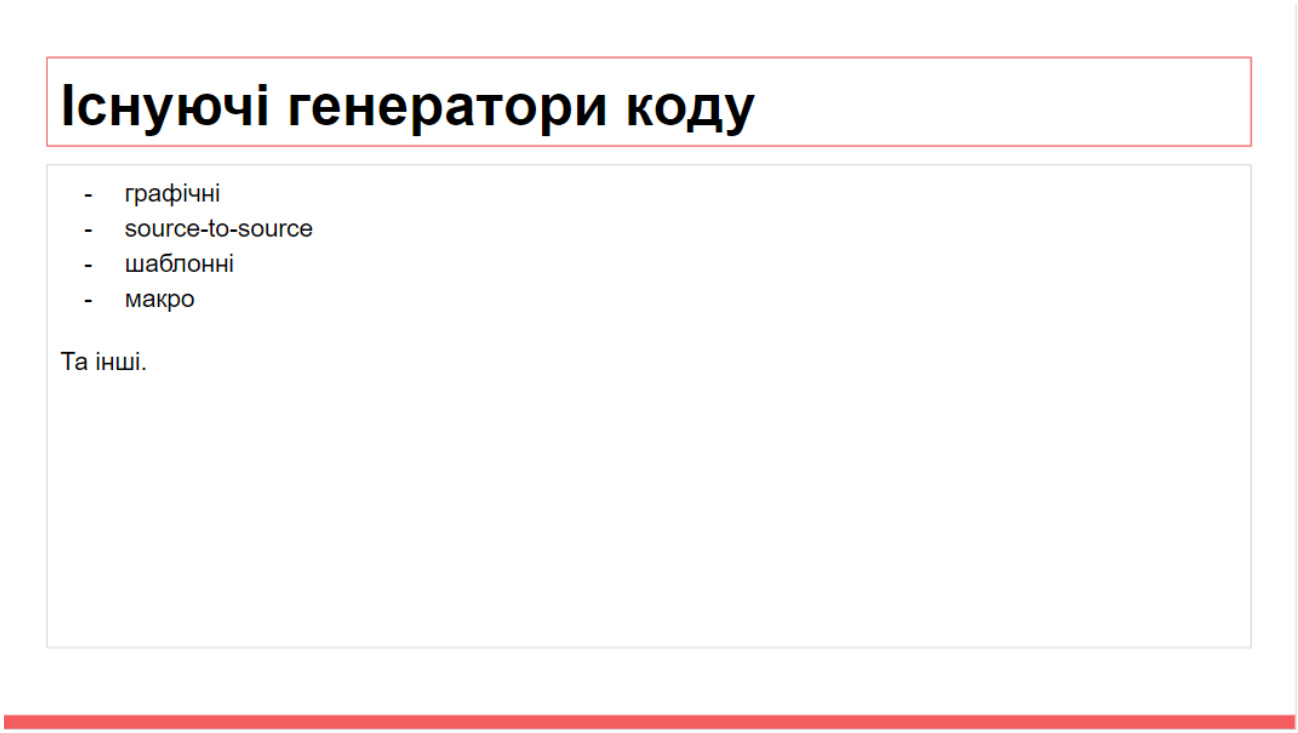


Рисунок Б.3 – Слайд №3

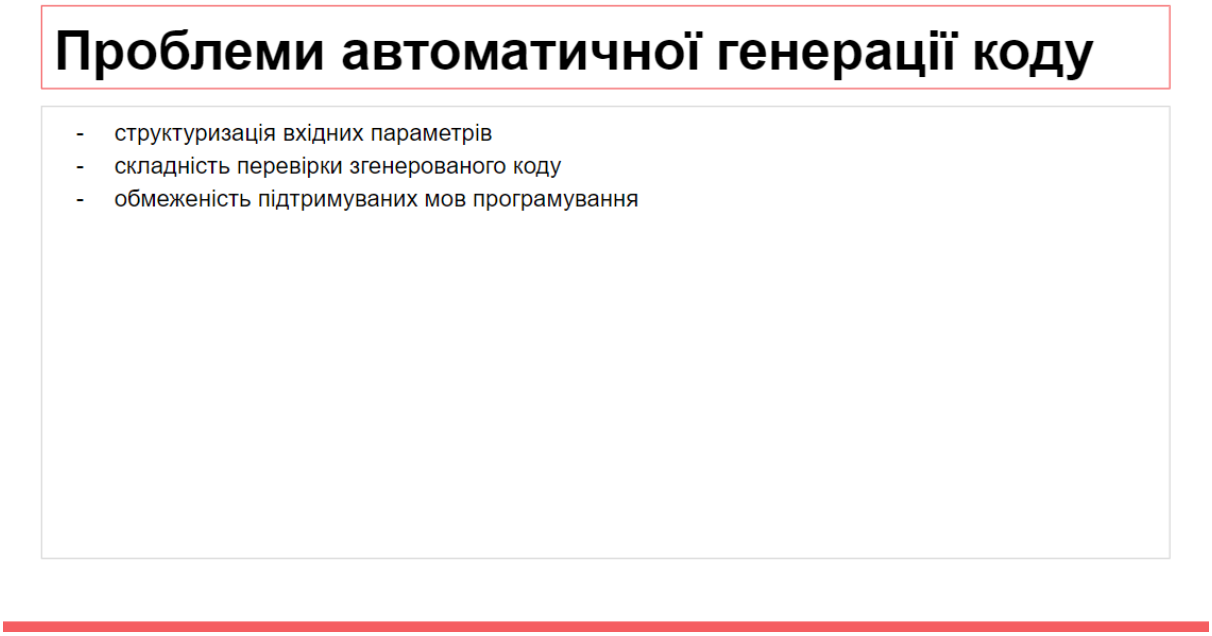


Рисунок Б.4 – Слайд №4

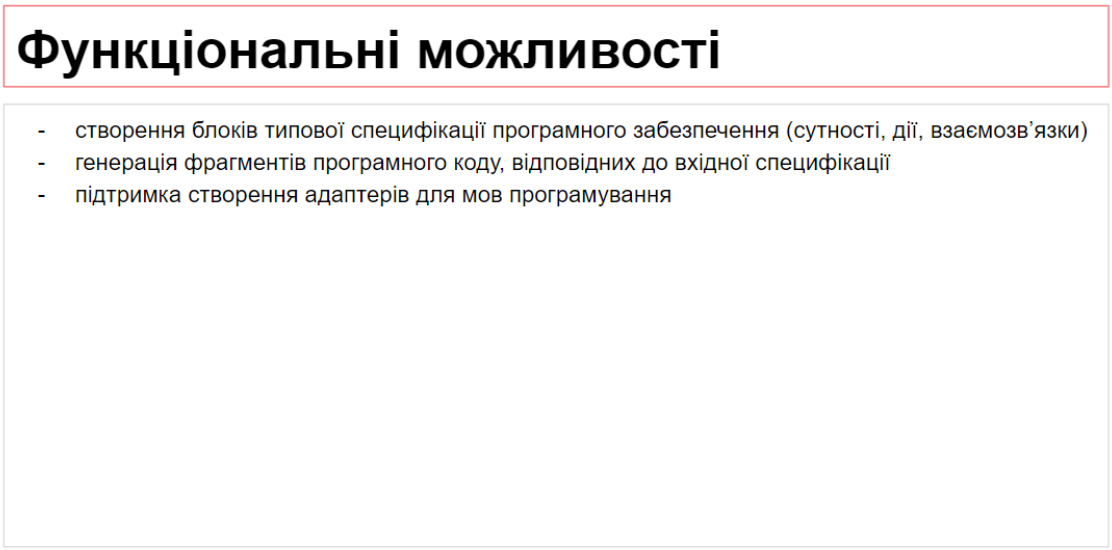


Рисунок Б.5 – Слайд №5

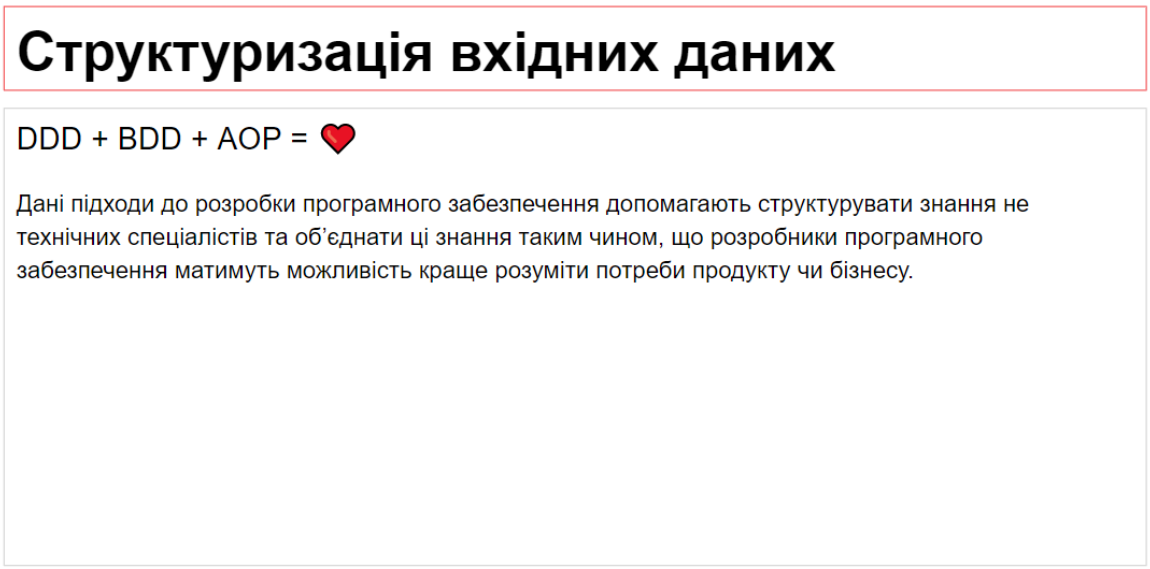


Рисунок Б.6 – Слайд №6

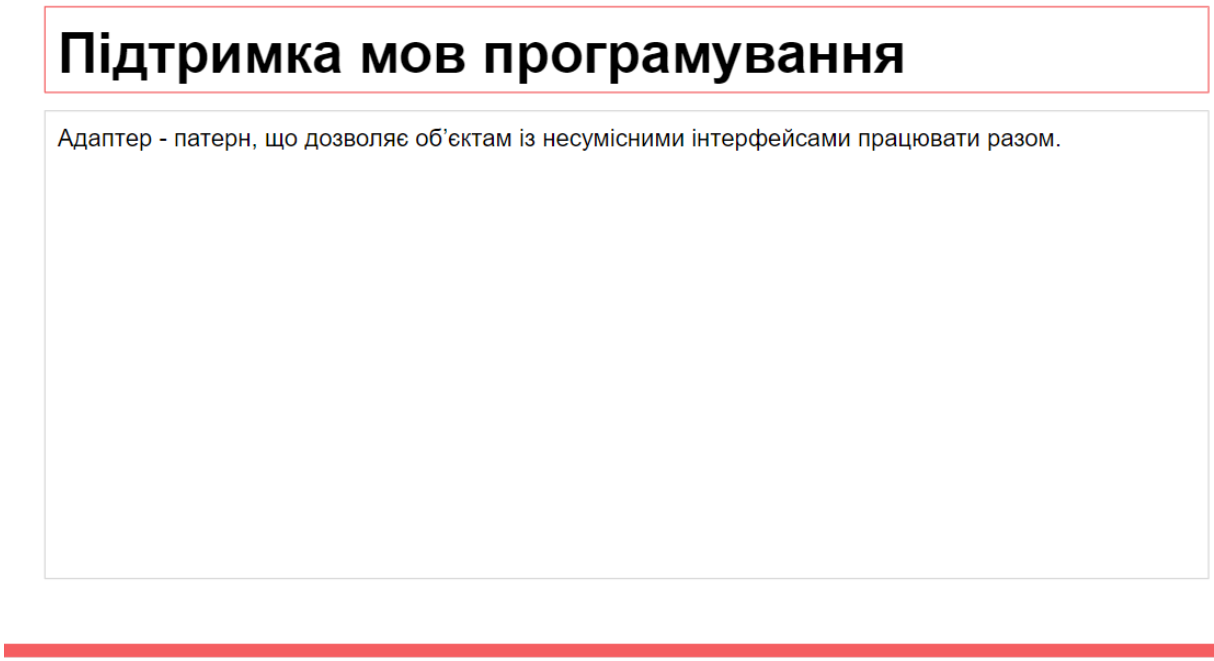


Рисунок Б.7 – Слайд №7

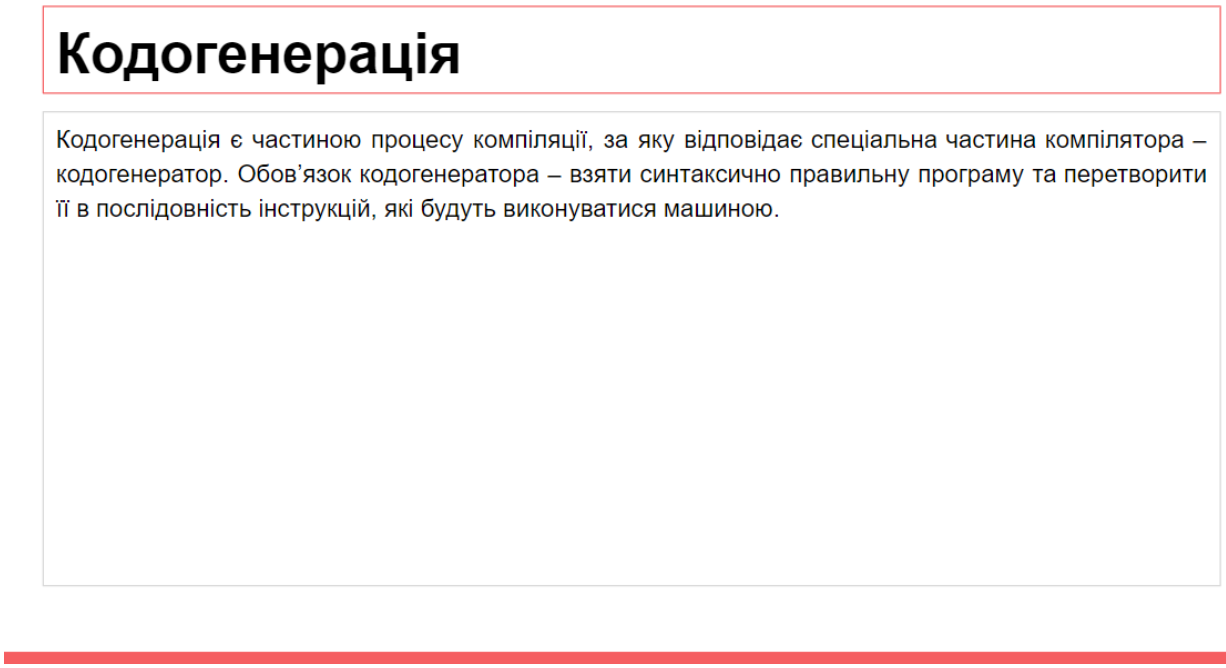
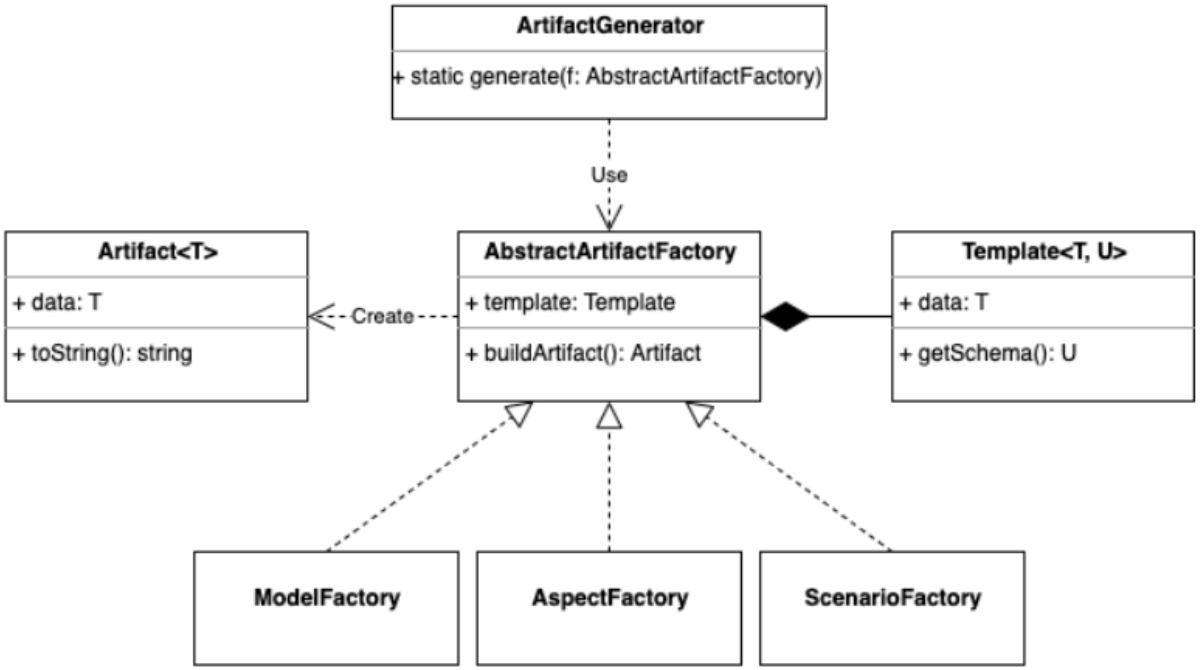


Рисунок Б.8 – Слайд №8

Архітектура кодогенератора



Діаграма класів

Рисунок Б.9 – Слайд №9

Приклад шаблону

```
describe {{type}} {{name}}:
      {{#each values}}
            {{this.name}} is {{this.type}};
      {{/each}}
end
```

Даний шаблон використовується для описання сутностей, необхідних для створення специфікації.

Рисунок Б.10 – Слайд №10

Фрагменти специфікації

```
@(VALUE: Note, OPERATION: Delete)
SCENARIO: User creates a Note
      GIVEN: User finds a note to delete
      WHEN: User presses the delete button
      THEN: The note is deleted
ASPECTS:
      BEFORE: validate
```

При використанні BDD та AOP з'являється можливість структурувати сценарії та процеси продукту чи бізнесу.

Рисунок Б.11 – Слайд №11

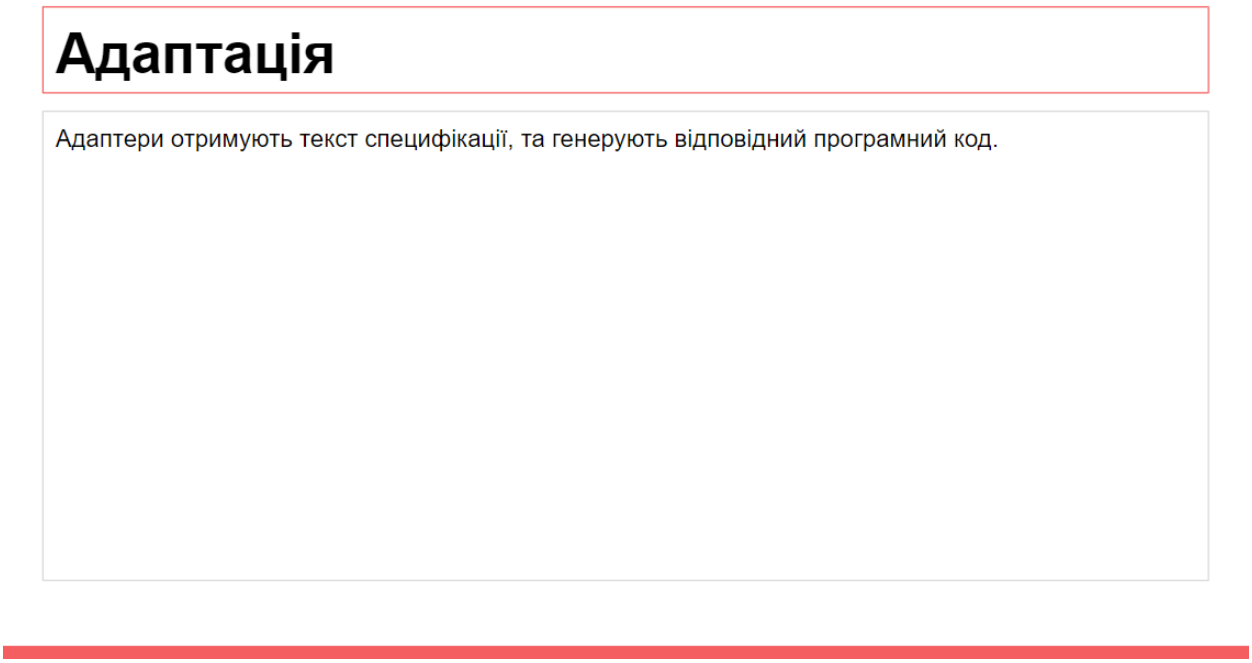
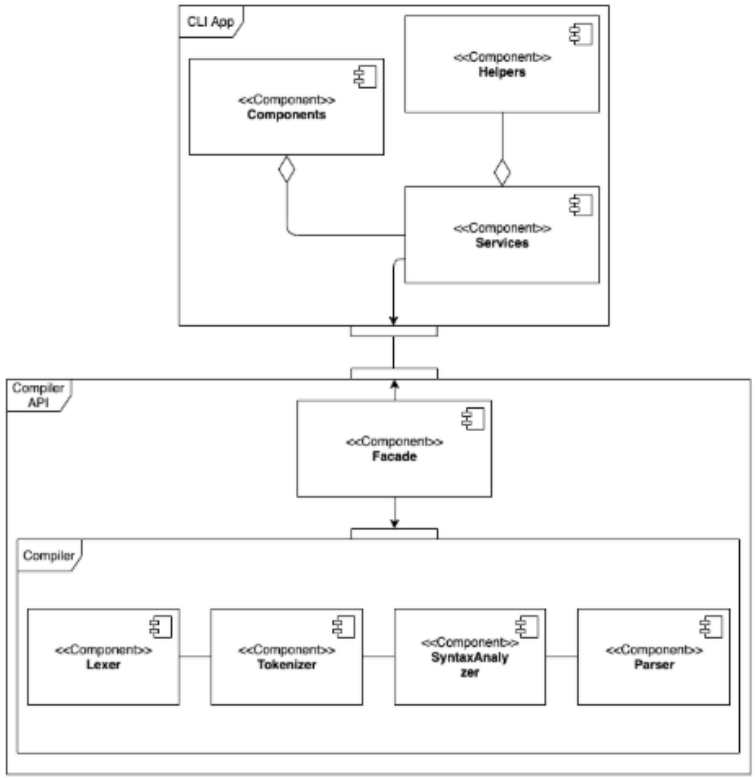


Рисунок Б.12 – Слайд №12

Архітектура адаптерів



Діаграма кооперації

Рисунок Б.13 – Слайд №13

Переваги даної реалізації

- структура вихідного програмного коду чітко відповідає вхідній специфікації
- доволі широка підтримка існуючих мов програмування
- використання практик, що довели свою ефективність при розробці програмного забезпечення

Рисунок Б.14 – Слайд №14

Висновки

Автоматичний генератор коду працює таким же чином як й існуючі компілятори та інтерпретатори багатьох мов програмування. Головною вимогою такого генератора є структурованість вхідних даних, це було досягнуто завдяки поєднанню предметно-орієнтованого, поведінково-орієнтованого та аспектно-орієнтованого проєктування.

Рисунок Б.15 – Слайд №15

ДОДАТОК В

(додатковий)

ТЕКСТ НАУКОВОЇ ПУБЛІКАЦІЇ

DOI 10.36074/01.05.2020.v2.08

ДОСЛІДЖЕННЯ МЕТОДІВ АВТОГЕНЕРАЦІЇ ПРОГРАМНОГО КОДУ

Паламар Вячеслав Олексійович
здобувач вищої освіти факультету комп'ютерних наук
Харківський національний університет радіоелектроніки, Україна

Науковий керівник: Каук Віктор Іванович
ORCID ID: 0000-0002-2780-2666
доц. каф. програмної інженерії, наук. кер. ЦТДН, канд. техн. наук, доцент
Харківський національний університет радіоелектроніки, Україна

Написання коду передбачає декілька (найчастіше складних) кроків – розбиття процесу чи явища на чіткі, послідовні інструкції, управління буферами вводу та виводу, керування пам'яттю тощо. Із розвитком індустрії процес написання коду полегшувався – створювалися мови програмування із автоматичним управлінням пам'яттю (також відомі як мови програмування із збірником сміття), для типових задач (робота із файлами певних форматів, веб-сервери, математичні розрахунки) створювалися бібліотеки та фреймворки, для полегшення моделювання створювалися об'єктно-орієнтовані мови програмування тощо. Тенденцію посилення абстракцій при розробці програмного забезпечення можна побачити і сьогодні.

Розробник програмного забезпечення може цього не знати, але кодогенерація вже є частиною процесу написання коду. Одне з властивостей теорії обчислюваності – повнота по Тьюрингу – полягає в тому, що програма може написати іншу програму. Це цікава ідея, яка не так оцінена, як того заслуговує, хоча і зустрічається досить часто. Більш того, кодогенерація – один з найвагоміших процесів, який виконується компілятором. Це частина процесу компіляції, за яку відповідає спеціальна частина компілятора – кодогенератор. Обов'язок кодогенератора – взяти синтаксично правильну програму та перетворити її в послідовність інструкцій, які будуть виконуватися машиною. Існування механізму кодогенерації напряму пов'язане із автоматичною генерацією коду та дає нам зрозуміти, що той код, що ми пишемо – це абстракція над справжніми інструкціями, які будуть виконуватися машиною. Отже, абстракція над безпосередньо кодом також має право на життя. Інструмент автоматичної генерації коду не обов'язково має бути прив'язаним до тієї чи іншої мови програмування, ми маємо розглядати це як кінцевий продукт автоматичного кодогенератора. Схожий підхід можна побачити в мові програмування Java – код компілюється у так званий байт-код, який виконується віртуальною машиною Java. Але таких машин існує декілька, кожна з них має ті чи інші переваги та недоліки. Тому, в ідеальному випадку, автоматичний генератор коду мав би перетворювати певні інструкції у вигляді чи то графіків та схем, чи то тексту англійською мовою, у спеціальну «проміжну мову», для якої можна розробити спеціальні адаптери для будь-якої іншої мови програмування.

Рисунок В.1 – Сторінка №1

Мета автоматичного генератора коду – конструювання (низького рівня) програмного забезпечення із специфікацій (високого рівня) [1]. Таким чином, автоматичний генератор коду має працювати наступним чином:

– зчитати вхідну специфікацію;
– перевірити правильність специфікації;
– зібрати фрагменти коду (провести «компіляцію» специфікації);
– провести оптимізацію (якщо можливо);
– згенерувати реалізацію на вихідній мові для відповідної платформи.

Велика проблема автоматичного генератора коду полягає в тому, що при використанні звичайного компілятору прірва між тим, що написано, та тим, що буде виконуватися (синтаксичне дерево – машинний код), не така велика, як при використанні автоматичного кодогенератора (модель – машинний код) [1]. Окрім того, навіть якщо подолати цю проблему, існує ще одна.

Ця проблема полягає в тому, що ми можемо довіряти такому кодогенератору тільки якщо ми маємо на увазі кодогенератор як частину компілятора, то так – цей інструмент вже перевірено часом та, що більш важливо, він має трансформувати лише обмежений набір даних або символів, тобто трансформувати абстрактне синтаксичне дерево (яке має обмежений набір понять) в інструкцію для машин. Автоматична генерація коду в цьому плані – більш складний та масштабний процес, адже набір наших даних необмежений. Ми можемо задати уявному автоматичному генератору коду задачу будь-якої складності – від генерування функції множення чисел до драйверу для системи управління бази даних. Іншими словами, ми можемо гарантувати правильність згенерованого коду компілятора, але жоден існуючий автоматичний генератор коду не може цього гарантувати (окрім тих, які розроблялися під чітку задачу – створення таблиць бази даних, простого REST-сервера, математичних формул тощо).

Нажаль, але автоматичний кодогенератор ніяк не відноситься до існуючих кодогенераторів в компіляторах. Як було зазначено вище, вони ідейно схожі, але практична реалізація буде зовсім інша, адже автоматична генерація коду – це не про компілятори, це про перетворення високорівневої специфікації на низькорівневий програмний код.

Слід зазначити, з чого складається типова специфікація. Частиною специфікації можуть бути UML та ER діаграми.

UML (Unified Modeling Language) – це, як видно з назви, унифікована мова моделювання, яку використовують в об'єктно-орієнтованому програмуванню та проєктуванні. Вважається, що UML є невід'ємною частиною процесу розробки програмного забезпечення. Існує багато видів UML діаграм.

ER модель, також відома як модель «сутність-зв'язок» - це така модель даних, що надає можливість описувати концептуальні схеми завдяки узагальненим конструкційним блокам. ER-модель використовується при високорівневому (концептуальному) проектуванні баз даних. Автоматична генерація коду з такої діаграми доволі популярна, адже така діаграма доволі точно описує взаємозв'язок сутностей. Більш того, створення таблиць в базі

Рисунок В.2 – Сторінка №2

даних – задача доволі рутинна (такі діаграми найчастіше використовують для описання саме зв'язку таблиць в реляційних базах даних).

Такі засоби чудово себе проявляють в том, щоб описати як об'єкти пов'язані один з одним. Але Ерік Еванс, автор предметно-орієнтованого проєктування, стверджує, що цього недостатньо [2]. Справа в тому, що вони чудово підходять для описання того, що є що та що із чим пов'язано. Але вони зовсім або майже зовсім не відображають, як і що працює. Майже всі реальні задачі та предметні області, які розробник програмного забезпечення хоче відобразити в коді – це не лише набір сутностей та зв'язок між ними. Так, ці підходи беруть цю задачу на себе, але задача побудови правил, обмежень, взагалі бізнес логіки все ще залишається на програмісті. Безумовно, такі засоби забезпечують ідентичність доменних та програмних моделей, але вони ніяк не можуть гарантувати те, що бізнес процеси та програмні процеси також будуть ідентичні. Тим не менш, інструменти, що працюють на базі цих підходів, можна вважати автоматичними генераторами коду, хоча й обмеженими у своїх можливостях.

На щастя, існує такий підхід, який допомагав би наблизити специфікацію, написаною мовою людей, до програмного коду, написаною на мові програмування. Більш того, їх існує декілька.

Розпочати слід з найбільш відомого – предметно-орієнтованого проєктування. Цей підхід спрямований на моделювання комплексного об'єктно-орієнтованого програмного забезпечення. Його переваги полягають у тому, основа увага концентрується на предметній області, а програмні моделі створюються таким чином, щоб відображати глибоке розуміння предметної області. Вперше цей термін було запроваджено Еріком Евансом в його книзі, яка має таку ж назву що і у даного підходу [2].

Для того, щоб краще розуміти даний підхід, необхідно розуміти основні визначення, які запроваджуються даним підходом.

Першим таким визначенням є домен. Домен – це предметна область, що буде використовуватися програмістом під час розробки програмного забезпечення. Для кращого описання моделі використовуються моделі, загальна мова та контекст.

Модель – це абстракція або система абстракцій, метою якої є описання окремих аспектів домену.

Загальна мова – це така мова, що будується навколо моделей домену. Вона використовується і програмістами, і експертами предметної галузі.

І, нарешті, контекст – це середовище, в якому чітко означені значення предметів та дій. Суттю предметно-орієнтованого проектування є конкретне визначення контекстів і обмеження моделювання в їх рамках. Треба точно знати контекст, в якому буде використовуватися модель.

На сьогоднішній день цей підхід є найбільш підходящим для вирішення нашої задачі – перетворення високорівневої специфікації на низькорівневий код. Але не лише цей підхід може допомогти досягти цієї мети.

Іншим підходом, який слід розглянути в розрізі даного дослідження – це аспектно-орієнтоване програмування.

Рисунок В.3 – Сторінка №3

Аспектно-орієнтоване програмування (АОП) – це парадигма, яка дозволяє відокремити наскрізну функціональність. Наскрізна функціональність - це така функціональність, яку не можна віднести до конкретного шару або рівня. Мета АОП – відокремити технічні проблеми (логування, транзакції, бази даних) з доменної моделі та полегшити розробку та реалізацію моделей предметної області, сконцентрованих перш за все на логіці бізнес-домену [3].

В даній парадигмі під аспектом розуміють модуль, що реалізує наскрізну функціональність. Для того, щоб змінити поведінку іншого коду, аспекти застосовують так звані поради в точках з'єднання, визначених зрізом.

Порада – це додаткова логіка. Це той код, що може бути виконаний до, після чи замість точки з'єднання.

Точка з'єднання – це точка в програмі, де рекомендується застосувати пораду. Під точкою в програмі можна розуміти створення об'єкта, виклик змінної тощо. Точки з'єднання можна об'єднати в зрізи, які, у свою чергу, визначають, наскільки дана точка з'єднання підходить до поради.

Завдяки аспектно-орієнтованому підходу можна розглядати програму як набір класів або модулів, кожен з яких виражає ту чи іншу особливість функціонування системи.

Аспектно-орієнтований підхід можна застосувати і для автоматичного генератора коду. Це дозволило би розробнику програмного забезпечення зосередитися на бізнес-логіці, а генератору коду – піклуватися про технічні, не пов'язані з бізнес-логікою моменти, які все ж необхідні.

І останній підхід, який також слід мати на увазі під час створення автоматичного генератора коду – це розробка, керована поведінкою (BDD). Основною ідеєю даної методології є суміщення чисто технічних інтересів та інтересів бізнесу під час розробки, дозволяючи тим самим керуючому персоналу та програмістам розмовляти однією мовою. Для спілкування між цими групами використовується предметно-орієнтована мова, основу якої складають конструкції із природної мови, зрозумілі не фахівцям, які зазвичай виражають поведінку програмного продукту та очікувані результати.

Тим не менш, сам по собі даний підхід не зовсім підходить для використання в автоматичному генераторі коду, оскільки він оснований на розробці, керованій тестами, а тому концентрується лише на передумовах та на очікуваних результатах. Але під задачу автоматичного генератора коду ця методологія цілком підпадає – вона наближає високорівневу специфікацію та низькорівневу програмну реалізацію один до одного.

Ці парадигми поєднує те, що вони запроваджують певний словник термінів, що допоможе зробити вхідну специфікацію більш структурованою. Таким чином, це полегшує задачу, адже хоча й специфікація може мати будь-який зміст, при чіткій структурі ми можемо працювати з такою специфікацією. Це так само, як і при звичайному написанні програми – ми можемо виразити майже будь-яку думку через код, але мі підкорюємося певним правилам мов програмування, тобто використовуємо певну структуру та конструкції при написанні програми.

Таку специфікацію необхідно чітко описати, щоб структура була зрозуміла кожному.

Рисунок В.4 – Сторінка №4

Отже, необхідно розробити формат специфікації, який поєднував би терміни DDD, BDD та AOD, та автоматичний генератор коду, що розумів би цю специфікацію та міг би перетворювати її в ту чи іншу мову програмування.

Висновки. Автоматична генерація коду можлива лише при чіткій структуризації вхідних даних, саме тому було описано структурні аспекти предметно-орієнтованого та аспектно-орієнтованого програмування, а також поведінково-орієнтованого проєктування. Наявність інструменту, що надавав ми можливість створювати чітко структуровану специфікацію, є мінімальною та необхідною умовою для створення автоматичного генератора коду.

Список використаних джерел:

1. Fischer, Bernd. (2006). Logical Foundations for Automated Code Generation. Estonian Summer School in Computer and Systems Science,№5, с. 11 – 13.
2. Evans, Eric. (2004). Domain-Driven Design. Tackling Complexity in the Heart of Software. Addison-Wesley.
3. Groves, Matthew. (2013). AOP in .NET. Practical Aspect-Oriented Programming. Manning.

Рисунок В.5 – Сторінка №5