ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Phạm Ngọc Quý

XÂY DỰNG CÔNG CỤ PHÁT HIỆN SỰ TUÂN THỦ MẪU THẾT KẾ CHO CÁC DỰ ÁN SỬ DỤNG JAVA

KHÓA LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY
Ngành: Công nghệ thông tin

HÀ NỘI - 2019

VIETNAM NATIONAL UNIVERSITY, HA NOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Pham Ngoc Quy

BUILDING TOOL FOR DETECTING THE COMPLIANCE ABOUT DESIGN PATTERN FOR JAVA USED PROJECTS

BACHELOR'S THESIS
Major: Information Technology

Supervisor: Assoc. Prof., Dr. Pham Ngoc Hung

HANOI - 2019

LỜI CẢM ƠN

TÓM TẮT

ABSTRACT

Keywords: *enterprise application, source code analyzing, change impact analyzing*

LỜI CAM ĐOAN

Hà Nội, ngày 26 tháng 04 năm 2019

Sinh viên

Phạm Ngọc Quý

Mục lục

Danh sách bảng

Danh sách chữ viết tắt

Danh sách hình vẽ

Chương 1

Đặt vấn đề

Chương 2

Kiến thức cơ sở

Chương 3

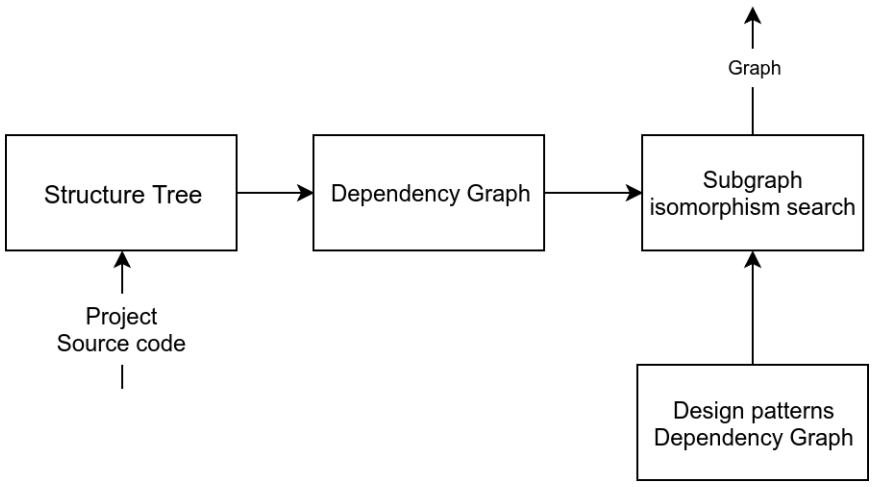
Phương pháp kiểm tra sự tuân thủ mẫu thiết kế cho dự án sử dụng Java

Mẫu thiết kế là tập hợp các luật nhằm mô tả cách giải quyết một vấn đề trong thiết kế có thể là vấn đề lặp lại nhiều lần trong dự án, là một khía cạnh của việc đảm bảo chất lượng mã nguồn, với những dự án công nghệ thông tin nói chung và dự án Java nói riêng. Ở các mẫu thiết kế hướng đối tượng, tập hợp các luật về thiết kế đem lại sự tương tác chặt chẽ giữa các thành phần trong mã nguồn với nhau. Những tương tác này có thể tổng quát hóa thành những dạng phụ thuộc nhất định. Đưa ra góc nhìn của mã nguồn về mặt cấu trúc. Do đó, phương pháp phát hiện sự tuân thủ mẫu thiết kế bên trong mã nguồn mà khóa luận này đễ xuất, đi theo hướng phân tích cấu trúc của mã nguồn.

3.1 Tổng quan phương pháp

Phương pháp phân tích cấu trúc mã nguồn dựa trên phấn tích tĩnh mã nguồn, bởi vì việc phân tích mã nguồn tĩnh đem lại độ chính xác tốt và quá trình phân tích không bắt buộc mã nguồn có thể thực thi được. Đem lại sự linh hoạt cho phương pháp này, dữ liệu đầu vào có thể là toàn bộ mã nguồn hoặc bất cứ một hay nhiều thành phần của mã nguồn.

Hình 3.1 mô tả phương pháp kiểm tra sự tuân thủ mẫu thiết kế. Đầu tiên, dữ liệu đầu vào được tiền xử lý thành cây cấu trúc, thông qua cây cấu trúc tiến hành phân tích phụ thuộc bên trong mã nguồn, xây dựng đồ thị phụ thuộc. Quá trình này được thực hiện với cả mã nguồn dự án và mã nguồn của những mấu thiết kế được qui định, từ đó tìm ra sự tương đồng của mẫu thiết kế bên trong mã nguồn dự án. Nhằm kiểm tra được sự tuân thủ về mẫu thiết kế bên trong mã nguồn.



Hình 3.1: Quá trình kiểm tra sự tuân thủ mẫu thiết kế của mã nguồn

3.2 Tiền xử lý mã nguồn Java

3.2.1 Xây dựng cây cấu trúc

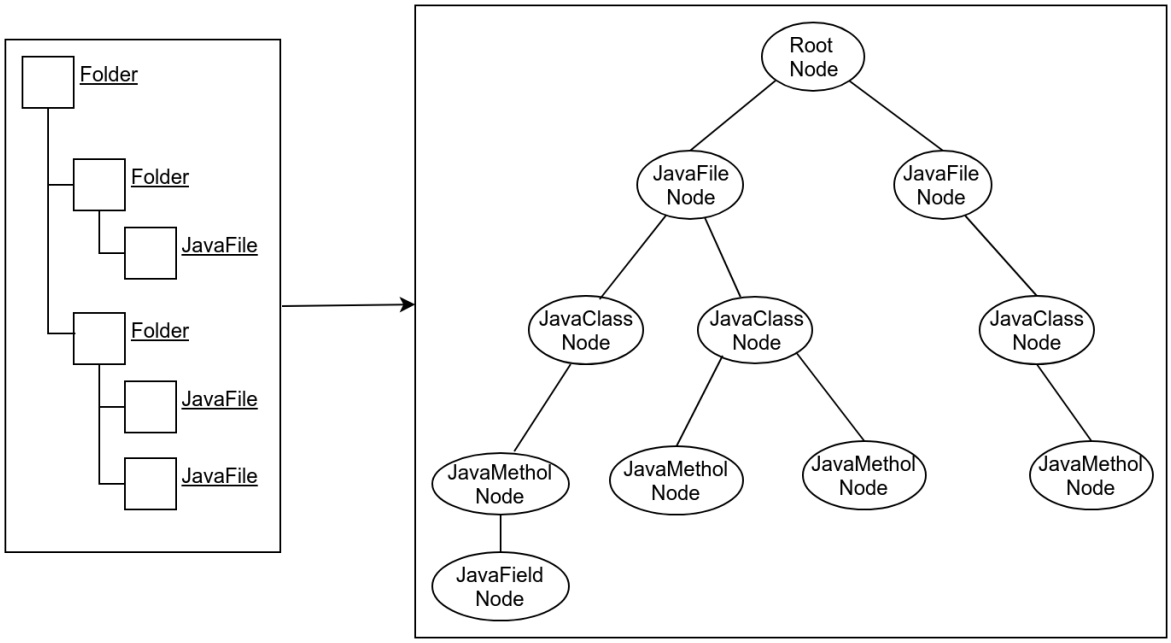
Đối với phương pháp kiểm tra sự tuân thủ mẫu thiết kế mà khóa luận này đề xuất. Việc trực tiếp phân tích mã nguồn đầu vào là một giải pháp không tối ưu do sự phức tạp của mã nguồn, các thành phần không được sử dụng tới của mã nguồn ví dụ như các tệp tin xml, yml, v.v. Do đó, cần có một kiểu dữ liệu tường minh và thể hiện được toàn bộ thông tin, cấu trúc của mã nguồn. Nếu dùng trực tiếp mã nguồn sẽ gây khó khăn trong quá trình giải quyết bài toán và ảnh hưởng tới hiệu năng của công cụ được xây dựng. Vấn đề đặt ra, đó là mã nguồn cần được tiền xử lý, loại bỏ những thành phần không sử dụng, ánh xạ mã nguồn sang kiểu cấu trúc dữ liệu phù hợp. Cây cấu trúc được để xuất như là một kiểu cấu trúc dữ liệu phù hợp nhất thể hiện được toàn bộ cấu trúc của mã nguồn dự án với những ưu điểm như kiểu cấu trúc dữ liệu tượng minh, việc quan lý các đối tượng là dễ dàng nhằm mục đích phụ vụ cho việc phân tích cây cấu trúc sau này, hỗ trợ thể hiện phụ thuộc giữa các đối tượng bằng việc xây dựng liên kết giữa các nút trên cây.

Định nghĩa: (*Cây cấu trúc* [2]) Là một đồ thị liên thông với $T = (N, E)$ trong đó $N = \{n_1, n_2, n_3...n_n\}$ là tập các nút trên cây đại diện cho tệp, lớp, phương thức, biến, v.v. $E = \{(e_i, e_j)|e_i, e_j \in N\}$ mỗi cặp e_ie_j là cặp hai nút cha con của cây.

Cây cấu trúc được xây dựng từ mã nguồn của dự án bằng cách ánh xạ lại những thông tin từ mã nguồn thành các nút trên cây. Hình 3.2 mô tả cây cấu trúc từ mã nguồn dự án. Trong đó toàn bộ mã nguồn có cấu trúc là một thư mục gồm các tệp tin và thư mục con được đưa về dạng cây cấu trúc với các nút trên cây được ánh xạ về bốn loại: tệp tin (*Java*), lớp, phương thức và một loại nút thể hiện cho những định dạng còn lại. Mỗi loại nút của cây chứa những thuộc tính khác nhau và thông tin về nút cha, con của chúng.

Những thông tin trên mỗi nút được phân tích từ cây cú pháp trừu tượng(Abstract

Syntax Tree - AST) được sinh ra từ mã nguồn. Ngoài ra, quá trình sinh cây AST từ mã nguồn dự án, là một bước trung gian của quá trình xây dựng cây cấu trúc, kiến thức về cây AST và công cụ JP đã được trình bày tại **Chương 2** của khóa luận.



Hình 3.2: Xây dựng cây cấu trúc từ mã nguồn

3.2.2 Xác định thuộc tính cho mỗi nút trên cây cấu trúc

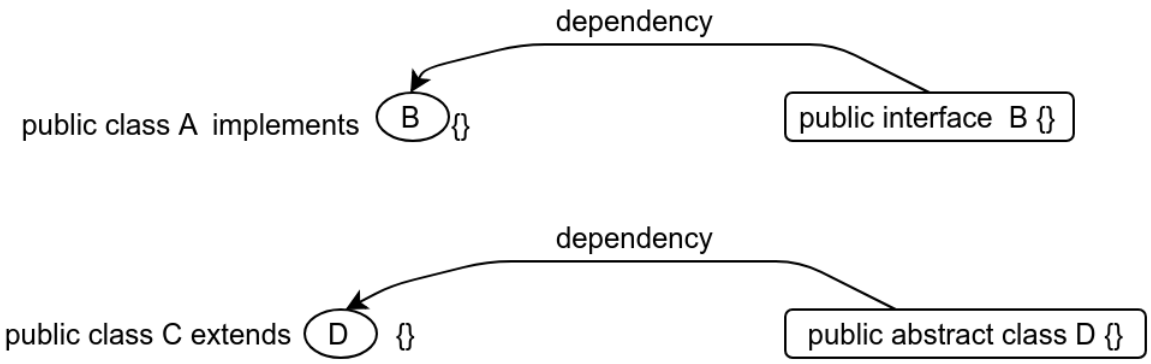
Xác định thuộc tính trên mỗi nút, nhằm ánh xạ đầy đủ thông tin cần thiết của mã nguồn lên cây cấu trúc mà ta cần xây dựng. Phục vụ quá trình phân tích phụ của mã nguồn. Quá trình xác định thuộc tính trên mỗi nút dựa trên việc phân tích mã nguồn ở các mức trừu tượng khác nhau như Class, Method, Variable, v.v.

Thành phần của một lớp gồm bốn phần chính: *Class type, Class dependency, Class variables, Method* . Trong đó:

- *Class type* của một nút (class) thể hiện nút đó đóng vai trò như một: *Class, Abstract class, Template class* hay *Interface.*

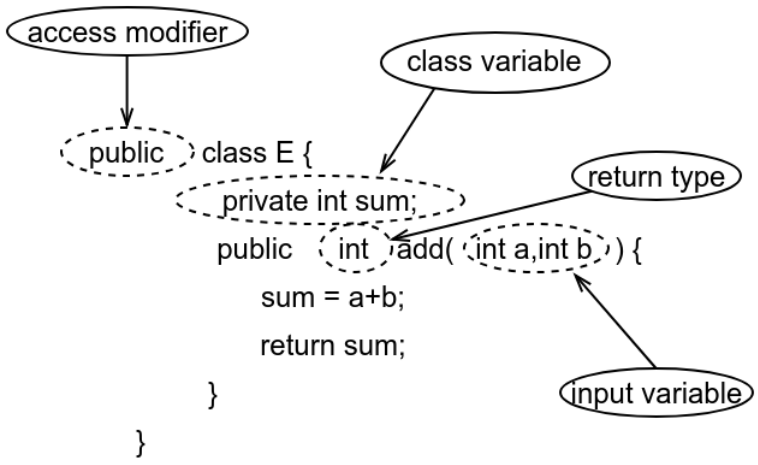
- *Class depedency* ở đây ta xét tới phụ thuộc thừa kế của lớp, phụ thuộc thừa kế bao gồm hai loại: kế thừa từ một class, kế thừa từ Interface.
- *Method* là định nghĩa một hành vi của lớp, *Method* bao gồm các thành phần: *Local variable, Return type, Input paramater.*
- *Class variables* là biến của một lớp được khởi tạo bên ngoài các *Method. Local variable* là biến chỉ được khai báo và sử dụng trong phạm vi *Method.*
- *Return type* là kiểu dữ liệu mà phương thức sẽ trả về nếu *Return type* là kiểu *void* thì phương thức sẽ không trả về giá trị.
- *Input paramter* xác định kiểu giá trị đầu vào cho phương thức.

Để rõ ràng hơn về quá trình xác định thuộc tính cho các nút của cây cấu trúc, ta tiến hành xem xét các ví dụ. Hình 3.3 mô tả hai loại phụ thuộc kế thừa qua phương thức extends và inplements. Trong đó A là một Class thừa kế từ B là một Interface qua phương thức extend, C là một class thừa kế D qua phương thức implement với D là một abstract class.



Hình 3.3: Phụ thuộc thừa kế của lớp

Hình 3.4 Các thành phần cơ bản của *Class*. Trong đó E là một Class với Access modifer là public, Class variable là sum với kiểu giá trị int và Access modifier là private, Method add() có kiểu trả về là int và hai biến đầu vào là a và b.



Hình 3.4: Các thành phần cơ bản trong class

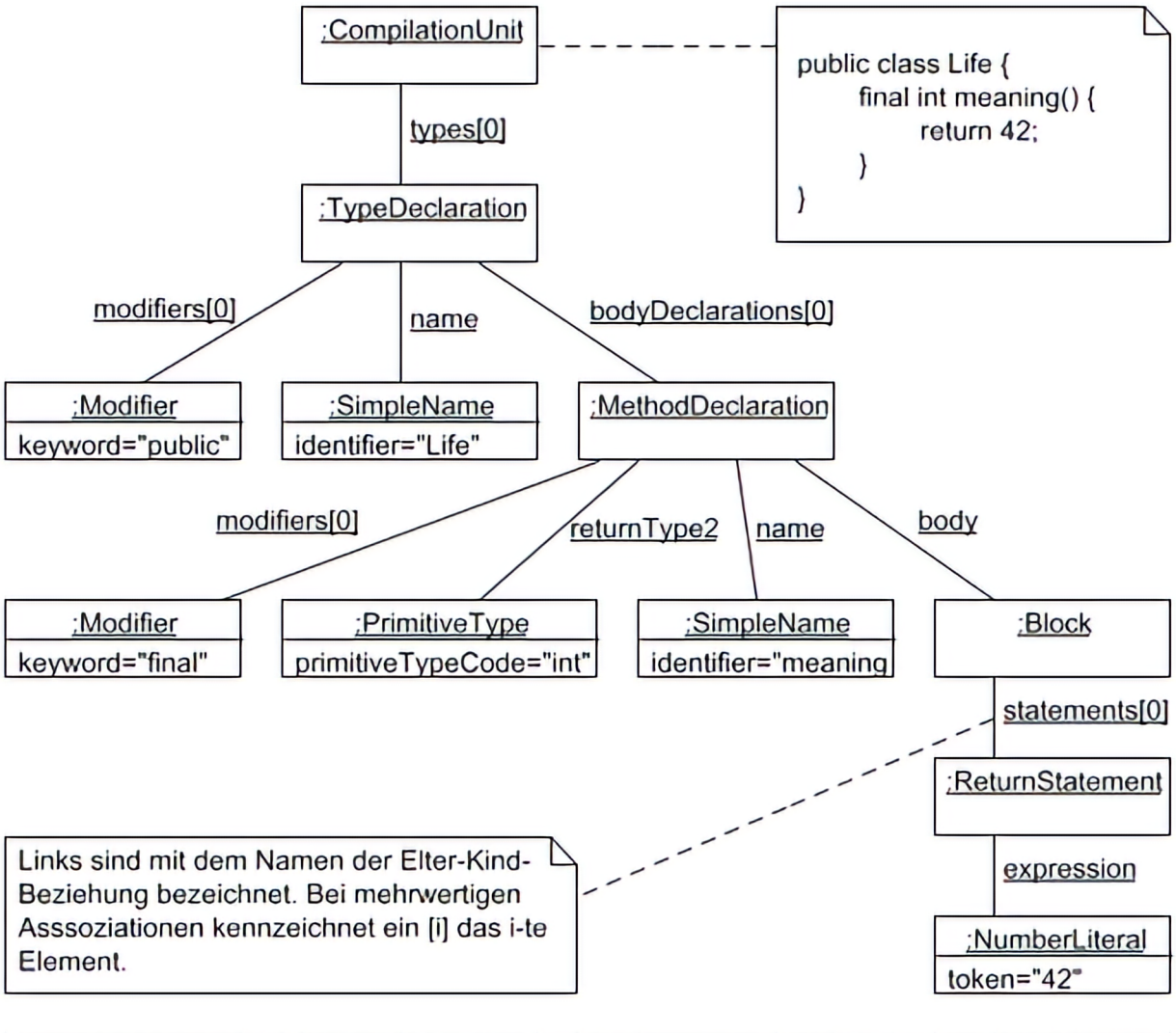
Bảng 3.1 mô tả đầy đủ những thông tin cần xác định cho mỗi loại nút trên cây cấu trúc, nhằm phục vụ cho việc phân tích cấu trúc mã nguồn và xây dựng đồ thị phụ thuộc sẽ được trình bài tại mục **3.2** và **3.3** của chương này.

Node	**Popertiesoperties**
Class	NameType Access modifier Extended Class Implemented Class Childrent Node: Field, Method
Method	Name NameReturn Type Access modifier Parameter Body
Field	Name Value type Access modifier

Bảng 3.1: Thuộc tính trên mỗi nút

Việc trích xuất các thông tin từ mã nguồn cho các nút trên cây, được thực hiện thông qua AST. Với mỗi thành phần mã nguồn, ta sử dụng JP để sinh AST tương ứng

với thành phần đó từ đó trích xuất các thuộc tính cần thiết cho mỗi nút trên cây cấu trúc. Hình 3.5 mô tả một AST với một Class Java tương ứng. Trong đó một lớp Java được phân tách thành dạng cây với các nút gốc chứa các toán tử, các nút lá chứa các toán hạng. Ví dụ như return = 42, trong đó return là một toán tử ứng với 'ReturnStatement' và '42' là toán hạng ứng với nút lá.



Hình 3.5: Abstract syntax tree đối với Java class

3.3 Phân tích cấu trúc mã nguồn Java

Cấy cấu trúc thể hiện thể hiện chi tiết về cấu trúc của mã nguồn bao gồm các khía cạnh về tính hướng đối tượng bên trong mã nguồn. Phân tích cấu trúc mã nguồn nhằm xác định được những đặc điểm về mặt phụ thuộc giữa các thành phần mã nguồn được hình thành bới việc áp dụng những mẫu thiết kế bên trong mã nguồn. Xác định được những được những đặc nêu trên là tiền để để kiểm tra sự tuân thủ mẫu thiết kế bên trong mã nguồn.

3.3.1 Phân tích phụ thuộc giữa các thành phần trong mã nguồn

Đối với phương pháp mà khóa luận này đề xuất, việc phân tích phụ thuộc giữa các thành phân bên trong mã nguồn xoay quanh việc phân tích phụ thuộc giữa các lớp trong mã nguồn. Đối với loại phụ thuộc giữa các lớp trong mã nguồn Java bao gồm: Direct & Indirect dependency, Polymorphism dependency, Inheritance Dependency, Use Dependecy,Behavior Dependency. Chi tiết về đăc trưng và ví dụ minh họa của từng loại phụ thuộc được trình bày như sau.

Polymorphism dependency: Là loại phụ thuộc khi có sự thừa kết giữa các đối tượng với nhau, tạo ra tính chất đa hình, đa trạng thái của đối tượng (Java) như tính chất liên kết động, upcasting, downcasting,v.v. Ở đây ta xem xét hai trường hợp của loại phụ thuộc này. Trường hợp thứ nhất khi một Class thừa kế một Interface bằng phương thức Implement. Ví dụ Class, A thừa kế một interface B, Class A sẽ thừa kế những phương thức của Interface C, tức là tại Class A những phương thức được Interface C định nghĩa sẽ được triển khai. Ngoài ra tham chiếu của Interface B có thể trỏ tới đối tượng của Class A, trong trường hợp đó đối tượng tạo được trỏ tới bởi B chỉ có thể thực hiện những phương thức mà B đã định nghĩa, nhưng phương thúc khác của A sẽ bị làm mờ đi. Trường hợp thứ hai, phụ thuộc xảy ra khi một class thừa kế một class khác thông qua phương thức extends. Ví dụ, class C thừa thế class D, lúc này ta coi D như là Class cha, với C là class con, C sẽ thừa hưởng mọi thuộc tính

và phương thức của D, do đó C có thể ghi đè những phương thức của D, ngoài ra, tham chiếu của Class D có thể trỏ tới đối tượng của class C. Hình 3.6 và 3.7 mô tả ví dụ về hai trường hợp mà ta đã đề cập.

```
public interface B {
    int add(int n1, int n2);
}
public abstract class A implements B {
    @Override
    public int add(int n1, int n2) {
        return n1 + n2;
    }
}
```

Hình 3.6: Mối quan hệ giữa một Class vớ một Interface qua phương thức Implements

```
public class C extends D {
 @Override
 public void getAge() {}
 @Override
 public void getName(String name) {
  System.out.println("i'm C");
 }
}
public class Main {
 public static void main(String[] args) {
  D d = new C();
  d.getAge();
 }
}
```

Hình 3.7: Mối quan hệ giữa một Class với một Class qua phương thức extends

Inheritance Dependency: Khi một Class có được các thuộc tính và phương thức của một Class khác. Những thuộc tính và phương thức này được quản lý theo thứ tự phân cấp từ lớp con tới lớp cha, việc xử lý phân cấp được quyết định trong quán trình chương trình đang thực thi bởi JVM. Ví dụ, ta có Class D thừa kế Class E với phương thức *extends*, khi đó D sẽ thừa hưởng các phương thưc và thuộc tính của E. Trong trường hợp các phương thức và thuộc tính của D có *Access modifier* là *private*, khi đó đối tượng của Class A sẽ không thể gọi tới những thuộc tính, phương thức này. Hình 3.8 mô tả mối quan hệ thừa kết giữa hai Class Java.

```
public class E {
 private String name;
 public void setName(String name) {
  this.name = name
 }
}

public class D extends E {
 @verride
 public void setName(String name) {
  super.setName(name)
 }
}
```

Hình 3.8: Mối quan hệ giữa một Class với một Interface qua phương thức Implements

Use Dependency: Không giống với *Inheritance Dependency* hay *Polymorphism dependency*, Use Dependency thể hiện sự tương quan giữa các Class về mặt tương tác dữ liệu,khi các đối tượng của lớp này được sử dụng nhứ là thuộc tính, giá trị trả về, kiểu dữ liệu đầu vào, biến địa phương của phương thức của Class khác hoặc được sử dụng để khai kiểu cho một Generic Class. Hình 3.9 mô tả *Use dependency*. Chia làm hai ví dụ nhỏ. Ví dụ thứ nhất, đối tượng của Class A được khai báo là thuộc tính của Class B, những phương thức của Class B có kiểu trả về, kiểu dữ liệu đầu vào là đối tượng của Class A. Ví dụ thứ hai, Class C khai bảo kiểu Generic, và định nghĩa những phương thức sử dụng kiểu Generic, tức là kiểu dữ liệu mà Class định nghĩa cho các phương thức, thuộc tính của nó sẽ phụ thuộc vào kiểu dữ liệu được khai báo khi

Class được sử dụng.

```
public class B {
 private A a;
 public A setNewProp(A a) {
  this.a = a;
  return a;
 }
 public A getA() {
  return a;
 }
}

class C < T > {
 private T obj;
 public T getObj() {
  return obj;
 }
 public void setObj(T obj) {
  this.obj = obj;
 }
}
```

Hình 3.9: Mô tả Use dependency

Behavior Dependency: Phụ thuộc thể hiện hành vi của các dối tượng của mỗi Class, tức là sự tương tác của đối tượng của Class này với đối tượng của một Class khác. Điều kiện cần của loại phụ thuộc này đó là giữa Class có tồn tại Inheritance dependency.

3.3.2 Xây dựng đồ thị phụ thuộc từ cây cấu trúc

Đồ thị phụ thuộc nhằm thể hiện mối quan hệ về cấu trúc giữa các thành phần trong mã nguồn, sự tương tác giữa các thành phần như Class, Methol, Field, v.v. với nhau bên trong mã nguồn, tạo nên sự ảnh hưởng qua lại giữa chúng việc phân tích và xây dựng đồ thị phụ thuộc xoay quanh phương pháp kiểm sự tuân thủ mẫu thiết kế bên trong mã nguồn dự án. Đối với phương pháp mà khóa luận này đề xuất, đồ thị phụ thuộc được xây dựng là đồ thị có hướng.

Đồ thị phụ thuộc (*Định nghĩa*): là một đồ thị có hướng $G = \{V, E\}$, trong đo $V = \{v_1, v_2, v_3..v_k\}$ là tập các đỉnh của đồ thị với mỗi đỉnh tương ứng với một Class trong mã nguồn, $E = \{e_i e_j | e_i \in V, e_j \in V\}$ là tập các cạnh đinh hướng của đồ thị từ đỉnh e_i tới e_j. Trên mỗi cạnh nối hai đỉnh chứa thuộc tính thể hiện sự phụ thuộc giữa hai đỉnh (class) của đồ thị (mã nguồn).

Từ việc phân tích các loại phụ thuộc phái trên, ta tiến hành định nghĩa một tập những phụ thuộc, được hiểu như là các cạnh của đồ thị phụ thuộc. Tập các đỉnh là những Class. Được kiệt kê tất cả trong Bảng 3.2 và Bảng 3.3 [1]

Bảng 3.2: Các loại đỉnh của đồ thị phụ thuộc

Kí hiệu	**Loại**
C	Class
I	Interface
A	Abstract class
T	Template class

Bảng 3.3: Các loại phụ thuộc Java

Kí hiệu	**Ý nghĩa**
X	class A extends class B
I	class A implement class B
C	class A create object of class B
R	class A has the return type of class B
MC	class A call a method of class B
F	class A has the field type of class B
MR	class A has a method with return type of class
MI	class A has a method that has an input parameter with the type of Class B
ML	class A has a method that defines a local variable with the type of class B
G	class A uses class B in a generic type declaration
M	class A has related with its method of class B
O	class A overrides of class B

Để xây dựng đồ thị phụ thuộc ta duyệt lần lượt từng cặp nút (class) trên cây cấu trúc, với mỗi cặp nút ta tiến hành phân tích thông tin của mỗi nút nhằm kiểm tra sự tồn tại phụ thuộc giữa chúng. Nếu phụ thuộc tồn tại một đối tượng *Dependency* được khởi tạo nhằm định danh phụ thuộc giữa hai nút.

Thuật toán 1: *JavaDependencyAnalyze(Root)*

Input : T là tập các nút trên cây cấu trúc

Output: Graph là đồ thị phụ thuộc

C = tập các nút lớp (Class Node) trên cây T ;

$G = NewGrpah()$;

foreach $c_i \in C$, C **do**

  d , $c_j = analyzerClassLevel(c_i, C, G)$;

  if *d not empty* **then**

    new Dependency(c, c_j, d);

  $analyzerMethodLevel(c_i, C, G)$;

  if *d not empty* **then**

    new Dependency(c, c_j, d);

  $analyzerdFieldLevel(c_i, C, G)$;

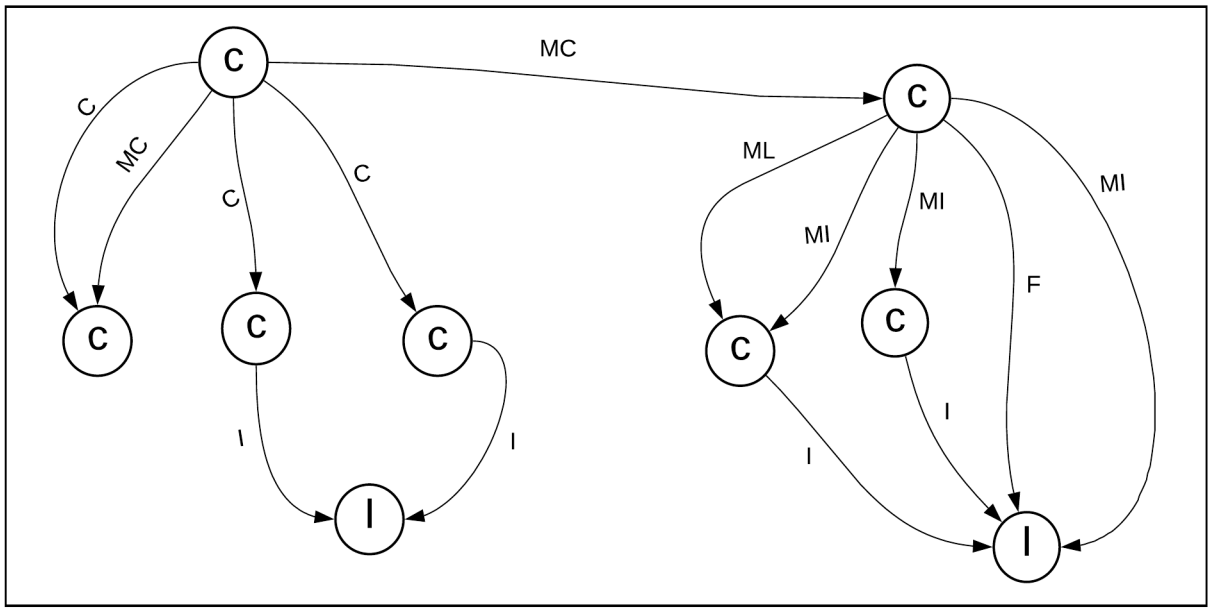
  if *d not empty* **then**

    new Dependency(c, c_j, d);

return G;

3.3.3 Ví dụ minh họa

Để hiểu rõ hơn về đồ thị phụ thuộc ta xem xét ví dụ Hình 3.10. Là độ thị phụ thuộc sinh ra từ một đoạn mã nguồn. Trong đó, đỉnh của đồ thị diểuiểu thị cho các lớp , **C** tương ứng với class và **I** tương ứng với Interface. Những class, interface này tương tác lẫn nhau qua những phụ thuộc như **MI** *: class A has a method that has an input parameter with the type of Class B*, **ML** : *class A has a method that defines a local variable with the type of class B*, v.v.



Hình 3.10: Ví dụ minh họa về đồ thị phụ thuộc

Đồ thị này là tiền đề đê kiểm tra sự tuân thủ mẫu thiết kế trong mã nguồn sẽ được trình bày ở phần sau.

3.4 Kiểm tra sự tuân thủ mẫu thiết kế bên trong mã nguồn

Quá trình kiểm tra sự tuân thủ mẫu thiết kế trong mã nguồn, tức là kiểm tra sự tồn tại của một mẫu thiết kế bên trong mã nguồn. Đầu vào của phần này gồm hai thành phần. Thứ nhất, **đồ thị phụ thuộc** của mã nguồn dự án. Thứ hai, **đồ thị phụ thuộc** của những mẫu thiết kế. Ta sẽ đi tiến hành kiểm tra sự tồn tại của từng mẫu thiết kế bên trong mã nguồn. Đầu vào của bài toán sẽ là hai đồ thị có hướng. Bản chất của vấn đề sẽ là tìm kiếm sự tồn tại của một đồ thị bên trong một đồ thị khác. Đo đó, thuật toán tìm kiếm đồ thị đẳng cấu **VF2 algorithm for the Subgraph Isomorphism Problem** được đề xuất để giải quyết bài toán.

Ý tưởng của giải thuật VF2 [4]:

- Ta cần tím kiếm một đồ thị con của đồ thị G1 mà đẳng cấu với đồ thị G2.
- Ý tương ở đây là đi xây dựng một trạng thái S chứa một phần các đỉnh của G1 và G2.
- M(s) là tập ánh xạ xác định hai đồ thị con của G1 và G2, gỉa sử là G1(s) và G2(s) thu được bằng cách chọn từ G1 và G2 cách cặp đỉnh chỉ chứa trong M(s) và các cạnh kết nối giữa chúng. Trong đó s là trạng thái của quá trình khớp đồ thị
- Ta cần mở rộng tập M(s) với những cặp đỉnh mới.
- Với mỗi trạng thái s, ta tính toán những cặp ánh xạ (n,m) là ứng viên cho M(s)
- Tập các luật được định nghĩa trước nhằm xác định một cặp đỉnh có là ánh xạ đúng hay không. Xác định cặp ánh xạ (n,m) có phải là ảnh xạ chính xác hay không. Việc thêm một cặp (n,m) vào M(s), sẽ chuyển trạng thái của s tới s'.

3.4.1 Xây dựng đồ thị đầu vào cho giải thuật VF2

Đồ thị đầu vào cho giải thuật là một đồ thị có hướng $G = (V, E)$. Trong đó V là các đỉnh tương ứng với các nút (class) trong cây cấu trúc, $E \subseteq V \times V$ là tập các cạnh với thuộc tính trên cạnh là phụ thuộc giữa các đỉnh trên đồ thị

Để xây dựng đồ thị gọi cho giải thuật VF2, ta cần trích xuất các đỉnh (class) và các phụ thuộc giữa chúng từ cây cấu trúc. Các bước để xây dựng đồ thị được trình bày ở Thuật toán 2.

Thuật toán 2: Xây dựng đồ thị từ cây cấu trúc

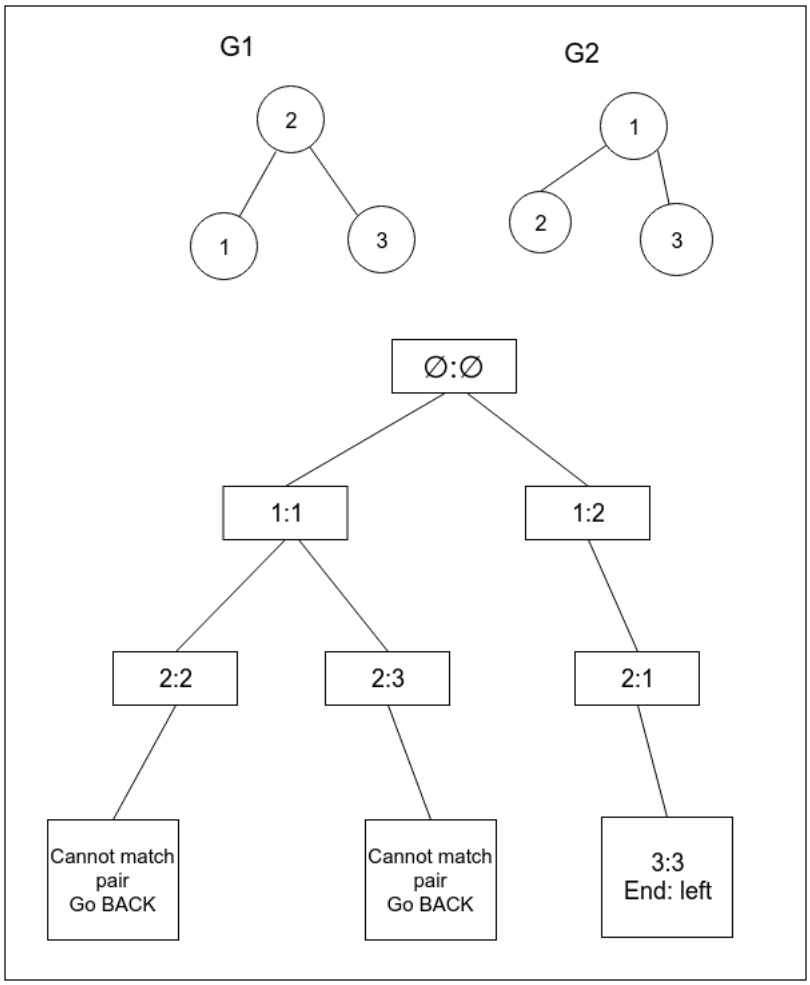
Input : *root*: nút gốc cây cấu trúc
Output: $G = (V, E)$: đồ thị gọi
Use : $getAllClassNode(root)$ các nút (class) của cây
Use : $getDepedencies()$ phụ thuộc của một nút

```
N = T.getAllClassNode(root);
G = Graph();
D = Set();
foreach n ∈N do
    D.add(n.getDepedencies());
G.addNode(N);
foreach d ∈D do
    G.addEdge(d);
return G ;
```

3.4.2 Kiểm tra sự tuân thủ mẫu thiết kế

Như ý tưởng đã trình bày ở trên, việc kiểm tra sự tuân thủ mẫu thiết kế, trình là bài toán tìm kiếm đồ thị con-đẳng cấu giữa hai đồ thị phụ thuộc đại diện cho mã nguồn và mẫu thiết kế. Quá trình tìm kiếm đồ thị đẳng cấu là quá trình ánh xạ các cặp đỉnh từ hai đồ thị. Việc này tạo nên một không gian trạng thái của những đáp án đổi với

mỗi trường hợp ánh xạ của mỗi cặp định khác nhau của hai đồ thị. Quá trình này được mô tả như Hình 3.11.



Hình 3.11: Mô tả thuật toán VF2

Giải thuật VF2 ra đời nhằm tối ưu quá trình tìm kiếm đáp án tối ưu trong không gian trạng thái, bằng phương pháp đề xuất tập các luật có thể xác định trạng thái tối ưu trong quá trình tìm kiếm cặp đỉnh ánh xạ giữa hai đồ thị, ngoài ra cũng làm giảm số lượng trạng thái được tạo ra trong quá trình tìm kiếm. Tập các luật được biểu diễn như sau:

$$F(s, n, m) = F_{syn}(s, n, m) \wedge F_{sem}(s, n, m) \tag{3.1}$$

Trong đó F_{syn} là tập các luật đánh giá trạng thái dựa trên cấu trúc của đồ thị, F_{sem} đánh giá trạng thái dựa trên các thuộc tính tại đỉnh, cạnh của đồ thị. Tập các luật được định nghĩa cho giải thuật VF2 như sau [5]:

1. Feasibility Rules

   - $F_{syn}(s,n,m) = R_{pred} \wedge R_{succ} \wedge R_{out} \wedge R_{new}$
   - $R_{pred}(s,n,m) \iff$
     $(\forall n \in M_1(s) \cup Pred(G_1,n)')\exists m' \in Pred(G_2,m)|(n',m') \in M(s)) \wedge$
     $(\forall m' \in M_2(s) \cup Pred(G_2,m)\exists n' \in Pred(G_1,n)|(n',m') \in M(s))$
   - $R_{succ}(s,n,m) \iff$
     $(\forall n \in M_1(s) \cup Succ(G_1,n)')\exists m' \in Succ(G_2,m)|(n',m') \in M(s)) \wedge$
     $(\forall m' \in M_2(s) \cup Succ(G_2,m)\exists n' \in Succ(G_1,n)|(n',m') \in M(s))$
   - $R_{in}(s,n,m) \iff$
     $(Card(Succ(G_1,n) \cup T_1^{in}(s)) \geq Card(Succ(G_2,m) \cup T_2^{in}(s))) \wedge$
     $(Card(Pred(G_1,n) \cup T_1^{in}(s)) \geq Card(Pred(G_2,m) \cup T_2^{in}(s)))$
   - $R_{out}(s,n,m) \iff$
     $(Card(Succ(G_1,n) \cup T_1^{out}(s)) \geq Card(Succ(G_2,m) \cup T_2^{out}(s))) \wedge$
     $(Card(Pred(G_1,n) \cup T_1^{out}(s)) \geq Card(Pred(G_2,m) \cup T_2^{out}(s)))$
   - $R_{new}(s,n,m) \iff$
     $Card(\tilde{N}_1(s) \cup Pred(G_1,n)) \geq Card(\tilde{N}_2(s) \cup Pred(G_2,n)) \wedge$
     $Card(\tilde{N}_1(s) \cup Succ(G_1,n)) \geq Card(\tilde{N}_2(s) \cup Succ(G_2,n))$

2. Semantic Feasibility

   - $F_{s,n,m} \iff n \approx m \wedge \forall (n',m') \in M(s), (n,n') \in B_1 \Rightarrow (n,n') \wedge \forall (n',m') \in M(s), (n',n) \in B_1 \Rightarrow (n',m) \approx (m',m)$

Các bước của giải thuật VF2 [5] sẽ được trình bày chi tiết ở Thuật toán 3.

Thuật toán 3: $VF2$

```
PROCEDURE Match(s)
    Input  : an intermediate state s; the initial state s_0 has M(s_0) = ∅
    Output: the mappings between the two graphs
    IF M(s) covers all the node of G2
        return M(s)
    ELSE
        Compute the set P(s) of the paris candidate for inclusion in M(s)
        FOREACH p ∈ P(s)
        IF the feasibility reles successd for the inclusion of p in M(s)
            Compute the state s' obtained by adding p to M(s)
            CALL Match(s')
        END IF
      END FOREACH
      Restore data structures
    END IF
END PROCEDURE Match
```

Nếu mẫu thiết kế có tồn tại trong mã nguồn, kết quả trả về sẽ là một tập ánh xạ giưa toàn bộ đỉnh của đồ thị từ mẫu thiết kế với các đỉnh tưởng ứng trên đồ thị của mã nguồn.

Từ việc giải quyết được bài toán kiểm tra sự tuân thủ của mẫu thiết kế trong mã nguồn, chương tiếp theo khóa luận sẽ trình bày chi tiết cách triển khai phương pháp đã trình bày vào xây dựng công cụ trong thực tế.

Chương 4

Công cụ và thực nghiệm

Trong chương này, khóa luận sẽ trình bày về kiến trúc và cách cài đặt của công cụ cho phương pháp đã được trình bày ở chương 3 bao gồm, kiến trúc tổng quan của bộ công cụ JCIA-VT, kiến trúc của công cụ mà khóa luận xây dựng, phương pháp tích hợp công cụ với JCIA-VT. Thêm vào đó, đưa ra một số ví dụ để tiến hành phân tích thử nghiệm và đánh giá công cụ.

4.1 Kiến trúc và cài đặt công cụ

4.1.1 Tổng quan bộ công cụ JCIA-VT

Ở phần này sẽ trình bày về kiến trúc của bộ công cụ JCIA-VT. Kiến trúc của JCIA-VT được mô tả trong Hình 4.1. JCIA-VT được xây dựng dưới dạng một ứng dụng Web dựa trên hai Framework Java là Spring và JavaServer Faces (JSF) bao gồm các mô-đun chính:

Preprocessor: Mô-đun này chịu trách nhiệm nhận dạng loại dữ liệu đầu vào, bao gồm các mô-đun nhỏ đảm nhận các chức năng như nhận dạng nền tảng, công nghệ mà dự án sử dụng, ví dụ như: Java Spring, Java Strust, C#, v.v. Đầu vào là mã nguồn dự án dưới dạng tệp tin nén. Mã nguồn dự án sẽ được giải nén, dựa vào các mô-đun *Language Detector*, *Frameword Detector*, *Configuration Detector* bằng các phân tích những tập tin cấu hình, tập tin mã nguồn để nhận dạng loại ngôn ngữ, công nghệ mà dự án sử dụng.

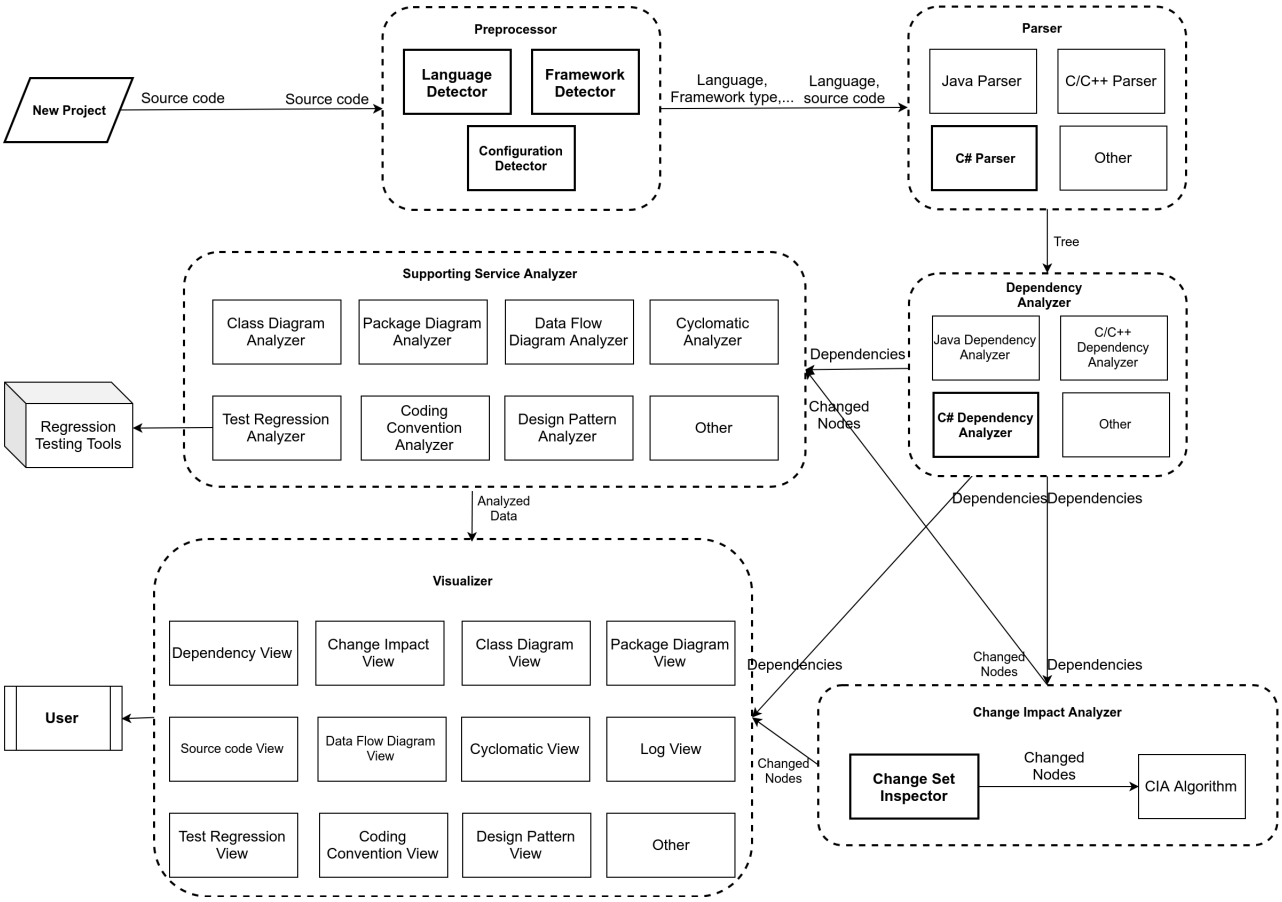
Parser: Mô-đun này nhận đầu vào là toàn bộ mã nguồn dự án. Mã nguồn sẽ được phân tích, mỗi mô-đun nhỏ phụ trách việc phân tích đối với từng loại mã nguồn khác nhau. Ngoài ra, cấu trúc dữ liệu được xây dựng cũng phụ thuộc vào từng loại công nghệ, mã nguồn là được tiến hành xây dựng. Ví dụ: đối với dự án sự dụng mã nguồn Java, cây cấu trúc sẽ được xây dựng.

Dependency Analyzer:Bao gồm nhiều mô-đun nhỏ phụ trách phân tích phụ thuộc cho từng loại ngôn ngữ, công nghệ. Đầu vào là mã nguồn đã được phân tích từ mô-đun *Parser*, tùy vào mỗi công nghệ, ngôn ngữ mà dự án sử dụng mà mô-đun nào sẽ được gọi tới. Ví dụ: đối với dự án Java, mô-đun *Java Dependency Analyzer* sẽ được gọi tới, tiến hành phân tích các phụ thuộc tồn tại bên trong mã nguồn trả về *cây phụ thuộc*.

Change Impact Analyzer: Là mô-đun phân tích thay đổi cho mã nguồn, đầu vào là một tập thay đổi, đầu ta là tập mã nguồn sẽ bị ảnh hưởng. Đây là một trong những

mô-đun đảm nhiệm chức năng chính cho JCIA-VT. Quá trình phân tích ảnh hưởng được thực hiện dựa trên giải thuật WAVE-CIA.

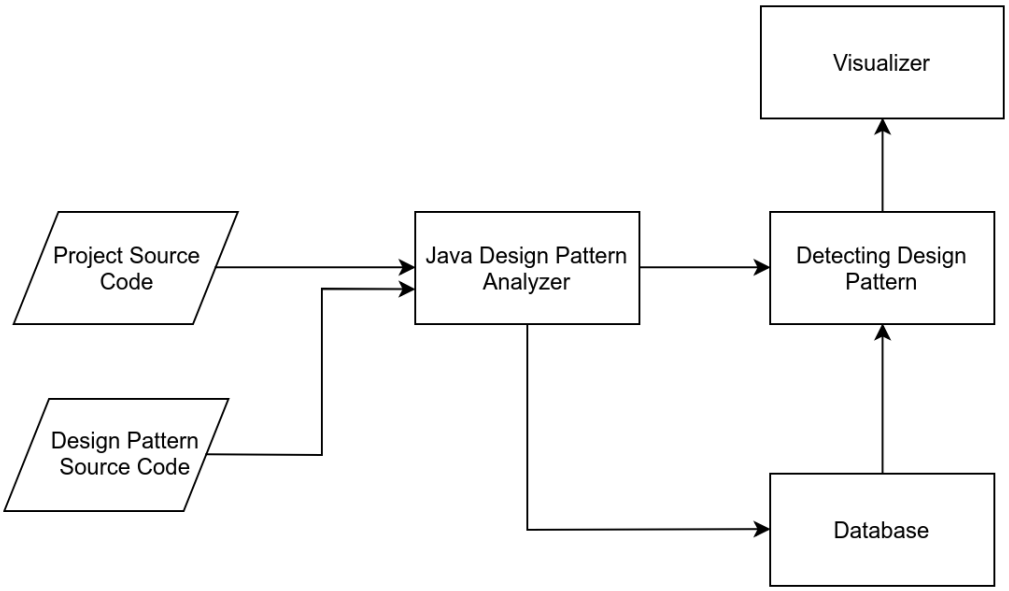
Supporting Service Analyzer: Ngoài chức năng *Phân tích ảnh hưởng*, JCIA-VT còn cung cấp thêm nhiều tính năng, phục vụ cho quá trình kiểm thử hồi quy (Regession Testing Tools). Bao gồm các mô-đun cung cấp các tính năng như: phân tích luồng dữ liệu (Data Flow Diagram Analyzer), phát hiện sự tuân thủ mẫu thiết kế (Detect Design Pattern), phân tích độ phức tạp của mã nguồ (Cyclomatic Analyzer) .v.v.
Visualyzer: Mô-đun thể hiện đầu ra của mã của những tính năng từ JCIA-VT



Hình 4.1: Tổng quan kiến trúc JCIA-VT

4.1.2 Kiến trúc chi tiết của công cụ kiểm tra sự tuân tủ mẫu thiết kế bên trong mã nguồn

Phần này sẽ trình bày chi tiết về kiến trúc chi tiết của mô-đun kiểm tra sự tuân thủ mã nguồn trong bộ công cụ JCIA-VT. Mô-đun có kiến trúc chi tiết được mô tả như trong hình 4.2, kiến trúc này được thiết kế dựa trên phương pháp đã mô tả ở phần 3 của khóa luận. Bao gồm bốn thành phần chính:



Hình 4.2: Kiến trúc

Java Design Pattern Analyzer: Là thanh phần chịu trách nhiệm phân tích mã nguồn. Đầu vào là mã nguồn Java, đầu ra là đồ thị phụ thuộc của mã nguồn. Mã nguồn sẽ được phân tích và tiến hành xây dựng đồ thị phụ thuộc như đã trình bày ở chưởng 3. Phần này gồm các gói chính:

- *ddp.dom*: Định cấu trúc các nút trên cây cấu của mã nguồn
- *ddp.parser*: Phân tích mã nguỗn, xây dựng cây cấu trúc từ mã nguồn
- *ddp.analyzer*: Phân tích cây cấu trúc, thực hiện gắn phụ thuộc giữa các nút trên cây. Những phụ thuộc cần phân tích đã được định nghĩa tại chương 3 mục 3.3.

- *ddp.graph*: Xây dựng đồ thị phụ thuộc, với đầu vào là cây cấu trúc của mã nguồn dự án.

Database: Quản lý việc lưu trữ cây cấu trúc của mã nguồn cũng như những mẫu thiết kế đã được định nghĩa của dự án. Sự dụng hệ quản trị cơ sở dữ liệu H2.

Detecting Design Pattern: Chịu trách nhiệm kiểm tra sự tồn tại của mẫu thiết kế bên trong mã nguồn. Sử dụng giải thuật VF2 như đã được trình bày tại chương 3 mục 3.4. Với đầu vào là cây cấu trúc của mã nguồn cùng với cây cấu trúc của mẫu thiết kế đã được định nghĩa.

Visualizer: Cung cấp giao diện thể hiện kết quả của quá trình kiểm tra sự tồn tại của mẫu thiết kế bên trong mã nguồn. Kết quả thể hiện dưới dạng đồ thị được triển khai bằng Javascript dựa trên thư việc Visjs [1].

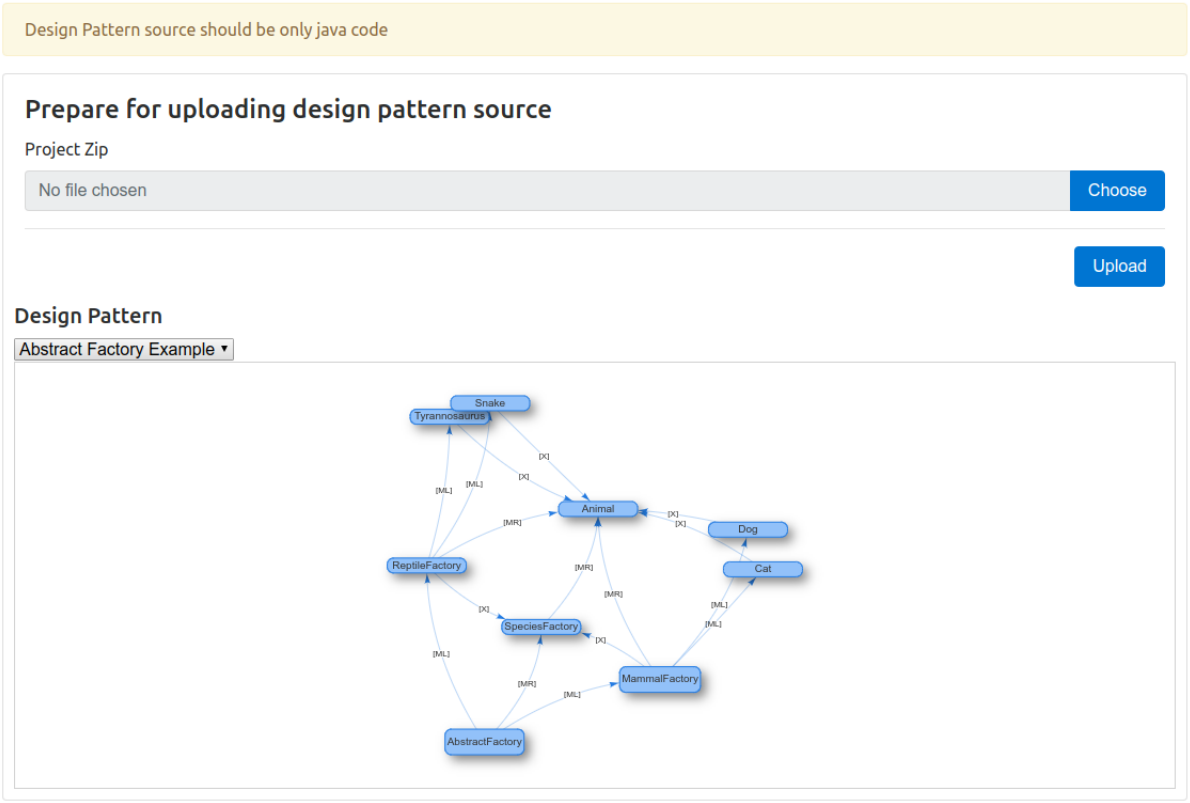
[1]http://visjs.org/

4.2 Triển khai và thử nghiệm

Hiện tại công cụ đã được phát triển và đặt, trên một phiên bản mới của JCIA-VT. Cung cấp những giao diện và tính năng mới liên quan tới việc kiểm tra sự tuân thủ của mẫu thiết kế bên trong mã nguồn.

4.2.1 Giao diện tải nên mẫu thiết kế

Sau khi đăng nhập thành công vào hệ thống, để có thể tiến hành phân tích, kiểm tra sự tuân thủ mẫu thiết kế bên trong mã nguồn. Người dùng cần tải lên những mẫu thiết kế đã được định nghĩa trước mà dự án sẽ được yêu cầu phát triển dựa trên những mẫu thiết kế đó. Mẫu thiết kế tải nên, nên chỉ tồn tại những tập tin Java và cần được đóng gói thành tập tin nén. Hình 4.3 mô tả màn hình tải nên một mẫu thiết kế mới.



Hình 4.3: Màn hình tải lên mẫu thiết kế

Chương 5

Kết luận

Tài liệu tham khảo

Tiếng Việt

Tiếng Anh

[1] Murat Oruc, Fuat Akal, Hayri Server. Detecting Design Patterns in Object-Oriented Design Models By Using Graph Mining Approach. pp 115,2016

[2] Ba Cuong Le, Son Nguyen Van, Duc Anh Nguyen, Ngoc Hung Pham, Hieu Vo Dinh. JCIA: A Tool for Change Impact Analysis of Java EE Applications. Information Systems Design and Intelligent Applications, pp.105-114, 2018.

[3] Nicholas Smith, Danny van Bruggen, Federico Tomassetti JavaParser: Visited Analyse, transform and generate your Java code base

[4] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone,and Mario Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs 10-2004 P1367

[5] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone,and Mario Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs 10-2004 P1368-1369