ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Phạm Ngọc Quý

XÂY DỰNG CÔNG CỤ PHÁT HIỆN SỰ TUÂN THỦ MẪU THIẾT 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

ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Phạm Ngọc Quý

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

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

Cán bộ hướng dẫn: PGS. TS. Phạm Ngọc Hùng
CN. Bùi Quang Cường

HÀ NỘI - 2019

VIETNAM NATIONAL UNIVERSITY, HA NOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Pham Ngoc Quy

BUILDING TOOL FOR DETECTING DESIGN PATTERNS ADHERENCE OF JAVA PROJECTS

BACHELOR'S THESIS
Major: Information Technology

Supervisor: Assoc. Prof., Dr. Pham Ngoc Hung
BS. Bui Quang Cuong

HANOI - 2019

LỜI CẢM ƠN

Lời đầu tiên, tôi xin được gửi lời cảm ơn chân thành và sâu sắc nhất tới PGS. TS. Phạm Ngọc Hùng - người thầy đã trực tiếp hướng dẫn, chia sẻ những kiến thức, kinh nghiệm quý báu trong suốt quá trình tập, nghiên cứu và thực hiện khóa luật tốt nghiệp này. Người đã đem tại những thay đổi cách nhìn trong tôi về cuộc sống, học tập và đam mê giúp tôi trưởng thành hơn, vững tin hơn trong con đường sắp tới.

Tôi xin gửi lời cảm ơn tới các thành viên trong tập thể K60CLC - những người đã đồng hành cùng tôi trong suốt bốn năm học vừa qua. Chia sẻ, giúp đỡ và động viên tôi trong những quãng thời gian khó khăn nhất. Cùng nhau chúng tôi đi qua thời sinh viên đầy khó khăn nhưng cũng nhiều niềm vui.

Tiếp theo tôi xin gửi lời cảm ơn tập thể cán bộ, thầy cô giảng viên trường Đại học Công Nghệ - Đại học Quốc Gia Hà Nội - những người luôn mang trong mình nhiệt huyết, tận tâm truyền đạt kiến thức, ngọn lửa đam mê tới tôi cũng nhưng toàn thể các bạn viên. Cuối cùng, tôi xin gửi lời cảm ơn sâu sắc nhất tới gia đình nơi sẽ luôn là niềm động lực to lớn cho tôi trong suốt cuộc đời mình.

TÓM TẮT

Mẫu thiết kế là một kỹ thuật trong lập trình hướng đối tượng, mẫu thiết kế bao gồm tập các luật nhằm mô tả cách giải quyết một vấn đề trong thiết kế, có thể là vấn đề lặp lại nhiều lần. Qua thời gian, mẫu thiết kế đã được đúc kết thành những khuôn mẫu chuẩn và được sử dụng thường xuyên trong các dự án phần mềm nói chung, dự án phần mềm sử dụng Java nói riêng. Mặt khác, dự án phần mềm ngày càng trở lên phức tạp sau những giai đoạn nâng cấp bảo trì, hệ quả là sự không nhất quán về mã nguồn so với đặc tả và thiết kế ban đầu. Dẫn tới sự khó khăn cho nhà phát triển trong việc nắm bắt được dự án, thực hiện công các đảm bảo chất lượng mã nguồn. Do đó, điều quan trọng là cần có một công cụ có thể kiểm tra sự tồn tại của mẫu thiết kế trong mã nguồn, từ đó phát hiện sự tuân thủ về những mẫu thiết kế bên trong mã nguồn so với đặc tả và thiết kế của dự án qua từng phiên bản. Khóa luận này đề xuất xây dựng một công cụ phát hiện sự tuân thủ mẫu thiết kế cho các dự án sử dụng Java. Đầu tiên, mã nguồn sẽ được tiền xử lý để sinh cây cấu trúc, mỗi nút trên cây cấc trúc đại diện cho một thành phần của mã nguồn. Từ cây cấu trúc tiến hành phân tích các phụ thuộc đặc trưng giữa các thành phần trong mã nguồn, sau đó đồ thị phụ thuộc được xây dựng từ cây cấu trúc với đỉnh là các thành phần mã nguồn, cạnh thể hiện phụ thuộc giữa hai thành phần mã nguồn. Đồ thị phụ thuộc sẽ là đầu vào cho quá trình phát hiện sự tuân thủ mẫu thiết kế bên trong mã nguồn. Quá trình phát hiện sự tuân thủ mẫu thiết kế bên trong mã nguồn so với đặc tả và thiết kế ban đầu là quá trình kiểm tra sự tồn tại của những mẫu thiết kế cần được áp dụng như đã trình bày trong đặc tả và thiết kế của dự án thực tế có tồn tại trong mã nguồn hay không. Từ đó đưa ra được kết luận về sự tuân thủ mẫu thiết kế bên trong mã nguồn. Cụ thể, quá trình kiểm tra sự tồn tại của mẫu thiết kế trong mã nguồn là quá trình tìm kiếm đồ thị đẳng cấu giữa đồ thị phụ thuộc của mẫu thiết kế với một trong những đồ thị con của đồ thị phụ thuộc của mã nguồn dự án.

Từ khóa: *ứng dụng doanh nghiệp, phân tích mã nguồn, đồ thị đẳng cấu, đảm bảo chất lượng mã nguồn*

ABSTRACT

Design patterns is a technique in object-oriented programming, which includes a set of rules that describe how to solve a problem in the design of the project, which can be a repeated problem. Over time, the design patterns have been drawn like the standard templates and used frequently in software projects in general, Java software projects in particular. On the other hand, software projects become more complicated after maintenance and upgrade stages, resulting in source code inconsistencies compared to the original design. Leads to the difficulty for developers in capturing project, implementing source code quality assurance. Therefore, the important is must have a tool to check the existence of design patterns in the source, from that return the result of detecting design patterns adherence in the project source code compared to the specification and design of the project through each version.This thesis proposes to build a tool for detecting design patterns adherence in the Java projects. First, the source code will be preprocessed to generate the structure tree, each node on the tree represents a component of the source code. From the structure tree, analyze the dependencies between the components in the source code, then the dependency graph of the project built from the structure tree. The dependency graph will be the input to the process of detecting design patterns adherence within the source code. The process of detecting design patterns adherence compared to the original specification and designs of the project is the detection process the exists of design patterns need to apply in project source code like in the original specification and design of the project. From that, show out the resulting of the design patterns adherence in the project source code.Specifically, the process of checking the existence of design patterns in source code is the process of examining the isomorphism graph between the dependency graph of the design patterns and the sub-graph of the project source code.

Keywords: *enterprise application, source code analyzing,graph-isomorphism,source code quality assurance*

LỜI CAM ĐOAN

Tôi xin cam đoan rằng những nghiên cứu của tối về phương pháp kiểm tra sự tuân thủ mẫu thiết kế cho các dự án sử dụng Java được trình bày trong khóa luận này là của tôi và chưa từng được nộp như một báo cáo khóa luận tại trường Đại học Công Nghệ - Đại học Quốc Gia Hà Nội. Những gì tôi viết trong báo cáo này không sao chép từ các tài liệu, không sử dụng các kết quả của người khác mà không trích dẫn cụ thể. Tôi xin cam đoan công cụ kiểm tra sự tuân thủ mẫu thiết kế cho các dự án sử dụng Java là do tôi tự phát triển, không sao chép mã nguồn của người khác. Nếu sai tôi hoàn toàn chịu trách nhiệm theo quy định của trường Đại Học Công Nghệ - Đại Học Quốc Gia Hà Nội.

Hà Nội, ngày ... tháng ... năm 2019

Sinh viên

Phạm Ngọc Quý

Mục lục

Danh sách bảng **vii**

Danh sách ký hiệu, chữ viết tắt **viii**

Danh sách hình vẽ **ix**

Chương 1 Đặt vấn đề **1**

Chương 2 Kiến thức cơ sở **4**

2.1 Đảm bảo chất lượng mã nguồn 4

2.2 Mẫu thiết kế 5

2.3 Kiểm tra sự tuân thủ mẫu thiết kế 6

2.3.1 Các phương pháp kiểm tra sự tuân thủ mẫu thiết kế 6

2.3.2 Tổng quan quy trình kiểm tra sự tuân thủ mẫu thiết kế 7

2.4 Bộ công cụ JCIA-VT 7

2.5 Công cụ phân tích Java 8

Chương 3 Phương pháp kiểm tra sự tuân thủ mẫu thiết kế cho dự án sử dụng Java **10**

3.1 Tổng quan phương pháp 10

3.2 Tiền xử lý mã nguồn Java 12

3.2.1 Xây dựng cây cấu trúc 12

3.2.2 Xác định thuộc tính cho mỗi nút trên cây cấu trúc 13
3.3 Phân tích cấu trúc mã nguồn Java 17
3.3.1 Phân tích phụ thuộc giữa các thành phần trong mã nguồn . . . 17
3.3.2 Xây dựng đồ thị phụ thuộc từ cây cấu trúc 20
3.3.3 Ví dụ minh họa 22
3.4 Kiểm tra sự tuân thủ mẫu thiết kế bên trong mã nguồn 23
3.4.1 Xây dựng đồ thị đầu vào cho giải thuật VF2 25
3.4.2 Kiếm tra sự tuân thủ mẫu thiết kế 25
Chương 4 Công cụ và thực nghiệm **29**
4.1 Kiến trúc và cài đặt công cụ 29
4.1.1 Tổng quan bộ công cụ JCIA-VT 29
4.1.2 Kiến trúc và cài đặt công cụ kiểm tra sự tuân thủ mẫu thiết kế bên trong mã nguồn 31
4.2 Triển khai 33
4.2.1 Giao diện tải lên mẫu thiết kế 33
4.2.2 Giao diện phát hiện mẫu thiết kế bên trong mã nguồn 34
4.3 Thực nghiệm 37
4.3.1 Thảo luận 41
Chương 5 Kết luận **43**

Danh sách bảng

3.1 Thuộc tính trên mỗi nút 15
3.2 Các loại đỉnh của đồ thị phụ thuộc 21
3.3 Các loại phụ thuộc Java 21

Danh sách chữ viết tắt

API Application Programming Interface

AST Abstract Syntax Tree

JP Java Parser

Danh sách hình vẽ

2.1 Quy trình kiểm tra sự tồn tại của mẫu thiết kế 7

3.1 Quá trình kiểm tra sự tuân thủ mẫu thiết kế của mã nguồn 11

3.2 Xây dựng cây cấu trúc từ mã nguồn 13

3.3 Phụ thuộc thừa kế của lớp 14

3.4 Các thành phần cơ bản trong class 15

3.5 Abstract Syntax Tree đối với Java Class 16

3.6 Mối quan hệ giữa một Class với một Interface qua phương thức Implements 18

3.7 Mối quan hệ giữa một Class với một Class qua phương thức extends . 18

3.8 Mối quan hệ giữa một Class với một Interface qua phương thức Implements 19

3.9 Mô tả Use dependency 20

3.10 Ví dụ minh họa về đồ thị phụ thuộc 23

3.11 Mô tả thuật toán VF2 26

4.1 Tổng quan kiến trúc JCIA-VT 31

4.2 Kiến trúc của công cụ phát hiện sự tuân thủ mã nguồn 32

4.3 Giao diện tải lên mẫu thiết kế 34

4.4 Giao diện tải lên mã nguồn dự án 35
4.5 Giao diện màn hình lịch sử 35
4.6 Giao diện chính và danh sách tính năng của JCIA-VT 36
4.7 Màn hình tính năng kiểm tra sự tuân thủ mã nguồn 37
4.8 Abstract Factory Class Diagram cho ví dụ 1 38
4.9 Kết quả phân tích ví dụ 1 40
4.10 Abstract Factory Diagram cho ví dụ 2 41

Chương 1

Đặt vấn đề

Hiện nay, những ứng dụng doanh nghiệp được phát triển trong thời gian dài với quy mô lớn và độ phức tạp cao. Trải qua nhiều quá trình nâng cấp và bảo trì, ứng dụng thường thiếu tài liệu đặc tả và thiết kế. Tài liệu gần như duy nhất của các ứng dụng này là mã nguồn. Trong khí đó, các hoạt động bảo trì và nâng cấp diễn ra thường xuyên. Đòi hỏi đội dự án cần thực hiện kiểm thử lại toàn bộ hệ thống để đảm bảo chất lượng ứng dụng cho mỗi phiên bản mới. Điều này là rất khó khăn vì tiêu tốn lượng lớn thời gian và kinh phí. Hệ quả là chúng ta không thể kiểm soát được sự thay đổi của mã nguồn, đảm bảo tính nhất quán giữa mã nguồn thực tế so với đặc tả và thiết kế ban đầu dẫn tới chất lượng mã nguồn ứng dụng không được đảm bảo. Chất lượng mã nguồn không được đảm bảo dẫn tới những rủi ro tiềm tàng cho doanh nghiệp trong quá trình bảo trì và vận hành sản phẩm.

Một trong những khía cạnh quan trọng của đảm bảo chất lượng mã nguồn, đó là việc áp dụng những mẫu thiết kế trong mã nguồn dự án. Mẫu thiết kế là tập các luật nhằm giải quyết các vấn đề liên quan tới thiết kế trong mã nguồn ứng dụng. Áp dụng mẫu thiết kế đem lại khả năng tái sử dụng mã nguồn cao, người phát triển có cái nhìn tổng quan về mã nguồn ứng dụng, dễ dàng nắm bắt mã nguồn đối với những nhà phát triển tham gia vào dự án.

Mặt khác, trong ngữ cảnh mà việc bảo trì nâng cấp ứng dụng ra thường xuyên, tài liệu đặc tả hầu như chỉ có mã nguồn ứng dụng, chi phí cho việc đảm bảo chất lượng

ứng dụng quá lớn và đòi hỏi nhiều thời gian. Hoạt động bảo chất lượng mã nguồn nói chung và hoạt động đảm sự nhất quán của mã nguồn so đặc tả và thiết kế ban đầu về mẫu thiết kế [1] (Design Patterns), chuẩn viết mã (Coding conventions) nói riêng là rất khó khăn.

Phát hiện sự tuân thủ mẫu thiết kế trong mã nguồn được xem là giải pháp có thể giải quyết phần nào vấn đề đảm bảo chất lượng mã nguồn như đã đề cập ở trên. Đối với những nhà phát triển đây là giải pháp giúp họ nắm bắt được tổng quan thiết kế mã nguồn, kiểm tra sự nhất quán giữa mã nguồn so với đặc tả và thiết kế ban đầu, giúp những nhà phát triển mới nhanh chóng hiểu-nắm bắt được dự án [2].

Tính tới tháng 4 năm 2019 Java là ngôn ngữ phổ biến số một thế giới (theo TIOBE [1]. Thật vậy, Java bao gồm nhiều công nghệ phổ biến như Spring, JSF, Struts, Hibernate, v.v. Bởi vậy nó hiện nay được áp dụng nhiều trong phát triển các ứng dụng doanh nghiệp với quy mô lớn, độ phức tạp cao. Cùng với đó, bộ công cụ JCIA-VT đã ra đời nhằm cung cấp các tiện ích về kiểm thử, đảm bảo chất lượng mã nguồn cho ứng dụng sử dụng Java trên các nền tảng như Spring, Hibernate, JSF, v.v.

Khóa luận này sẽ đề xuất giải pháp và xây dựng công cụ phát hiện sự tuân thủ mẫu thiết kế cho dự án sử dụng Java, nhằm khắc phục vấn đề về chất lượng mã nguồn đối với những dự án Java. Cụ thể, mã nguồn dự án sẽ được phân tích nhằm kiểm tra có tuân thủ theo đúng những mẫu thiết kế như trong tài liệu đặc tả và thiết kế ban đầu hay không và những mẫu thiết kế đó sẽ ảnh hưởng tới những thành phần nào khác trong mã nguồn. Công cụ này sẽ được tích hợp vào bộ công cụ JCIA-VT như là một tiện ích.

Các phần còn lại của khóa luận sẽ được trình bày như sau. Chương 2 sẽ trình bày về lý thuyết đảm bảo chất lượng mã nguồn, mẫu thiết kế hướng đối tượng. Ngoài ra sẽ giới thiệu về thư việc Java Parser - là một thư viện hỗ trợ phân tích mã nguồn Java nhanh và mạnh mẽ. Trong chương 3, sẽ trình bày về phương pháp kiểm tra sự tuân thủ mẫu thiết kế bao gồm các phần về phương pháp tiền xử lý mã nguồn Java, phương pháp sinh cây cấu trúc và phân tích các phụ thuộc, phương pháp sinh đồ thị phụ thuộc từ cây cấu trúc, phương pháp kiểm tra sự tồn tại của mã nguồn tiếp cận

[1] https://www.tiobe.com/tiobe-index/

theo hướng sử dụng đồ thị. Chương 4 triển khai và thực nghiệm, bao gồm mô tả kiến trúc công cụ JCIA-VT, mô tả chi tiết kiến trúc công cụ phát hiện sự tuân thủ mẫu thiết kế, mô tả quá trình cài đặt và tích hợp vào bộ công cụ JCIA-VT, tiến hành một số thực nghiệm để đưa ra kết luận về sự chính xác của công cụ. Chương cuối cùng, chương 5 sẽ đưa ra kết luận về những gì mà khóa luận đã thực hiện được bao gồm những gì đã thực hiện được, nhưng hạn chế còn tồn tại. Ngoài ra sẽ trình bày công việc cần thực hiện trong các giai đoạn tiếp theo.

Chương 2

Kiến thức cơ sở

Chương này sẽ trình bày cơ sở lý thuyết về những kiến thức liên quan tới phương pháp mà khóa luận này đề xuất, đồng thời giới thiệu về những công cụ liên quan được đề cập trong khóa luận này.

2.1 Đảm bảo chất lượng mã nguồn

Đảm bảo chất lượng phần mềm luôn được coi là hoạt động khó khăn tiêu tốn nhiều chi phí đồng thời cũng là một trong những yếu tố quyết định tới sự thành công của một dự án, sản phẩm phần mềm. Các sản phẩm phần mềm thường phải thay đổi liên tục nhằm thích nghi với các yếu tố thay đổi đồng thời đáp ứng nhu cầu của người dùng cuối. Khi các thay đổi được thực hiện, tính nhất quán giữa các thành phần sẽ khó được đảm bảo, điều này sẽ gây ra những ảnh hưởng không thể lường trước tới chất lượng phần mềm.

Một trong những khía cạnh quan trọng trong đảm bảo chất lượng phần mềm, đặc biệt là khi phần mềm trải qua các giai đoạn nâng cấp và bảo trì đó là đảm bảo chất lượng mã nguồn (source code quality assurance). Việc đảm bảo chất lượng mã nguồn là vấn đề cốt lõi để đảm bảo tính ổn định của sản phẩm phần mềm bên cạnh những yếu tố như cơ sở hạ tầng, bảo mật, v.v.

Đảm bảo chất lượng mã nguồn là một quá trình bao gồm nhiều hoạt động cần được thực hiện như kiểm tra quy chuẩn của mã nguồn, kiểm tra trùng lặp mã nguồn, kiểm tra sự tuân thủ mẫu thiết kế bên trong mã nguồn, v.v.

2.2 Mẫu thiết kế

Mẫu thiết kế [1] là một kỹ thuật trong lập trình hướng đối tượng, bao gồm tập các luật nhằm mô tả cách giải quyết một vấn đề trong thiết kế. Qua quá trình phát triển của ngôn ngữ lập trình,tập các luật này được nghiên cứu và đúc kết thành các mẫu chuẩn. Mẫu thiết kế không phải là ngôn ngữ lập trình cụ thể nào mà nó được áp dụng trong hầu hết các ngôn ngữ lập trình hướng đối tượng. Áp dụng mẫu thiết kế trong dự án phần mềm một cách hợp lý có thể tiết kiệm thời gian của quá trình viết mã nguồn cho dự án, đồng thời thúc đẩy khả năng tái sử dụng mã nguồn và khả năng bảo trì của dự án từ đó làm giảm tổng chi phí phát triển dự án. Ngoài ra, do các mẫu thiết kế đã được xác định, điều này làm cho mã nguồn dễ hiểu và dễ gỡ lỗi hơn, các thành viên mới trong đội dự án dễ dàng nắm bắt mã nguồn hơn.

Về phân loại mẫu thiết kế, hiện này mẫu thiết kế được chia thành ba loại chính bao gồm:

- Creational Design Patterns: Là những mẫu thiết kế thuộc loại này tập trung vào cung cấp giải pháp để tạo ra các đối tượng mà lô-gic của việc tạo ra đối tượng(Object) đó được che giấu thay vì tạo ra đối tượng bằng cách sử dụng trực tiếp phương thức *New*
- Structural Design Patterns: Những mẫu thiết kế thuộc loại này thường liên quan tới lớp(Class) và các thành phần liên quan của đối tượng. Xoay quanh mục đích là cung cấp các cách khác nhau để tạo ra cấu trúc lớp, ví dụ như sử dụng tính chất kế thừa và thành phần để tạo một đối tượng lớn từ các đối tượng nhỏ.
- Behavioral Design Pattern: Những mẫu thiết kế thuộc loại này hướng tới cung cấp giải pháp cho sự tương tác giữa các đối tượng, chiến lược nhằm giảm giàng

buộc và tăng tính linh hoạt giữa các thành phần trong mã nguồn khiến cho việc mở rộng trở lên dễ dàng.

Tại các doanh nghiệp, trong quá trình phát triển và nâng cấp, bảo trì dự án phần mềm ngoài việc áp dụng những mẫu thiết phổ biến đã được định nghĩa sẵn **còn áp dụng những mẫu thiết kế đặc thù do doanh nghiệp tự định nghĩa** dựa trên các mẫu thiết kế phổ biến để giải quyết những vấn đề cụ thể.

2.3 Kiểm tra sự tuân thủ mẫu thiết kế

Mẫu thiết kế thương xuyên được sử dụng trong những dự án phần mềm nói chung và dự án phần mềm sử dụng Java nói riêng. Mục đích của mẫu thiết kế là để giải quyết những vấn đề liên quan tới thiết kế trong mã nguồn, tăng tính sử dụng dụng lại của mã nguồn, tăng khả năng bảo trì và gỡ lỗi, giải quyết phần nào vấn đề thiếu tài liệu của dự án phần mềm, do mẫu thiết kế là được xác định trước, bởi vậy sẽ giúp cho người đọc dễ dàng nắm bắt được mã nguồn của phần mềm. Do đó, áp dụng mẫu thiết kế làm giảm tổng chi phí phát triển, bảo trì, nâng cấp dự án phần mềm.

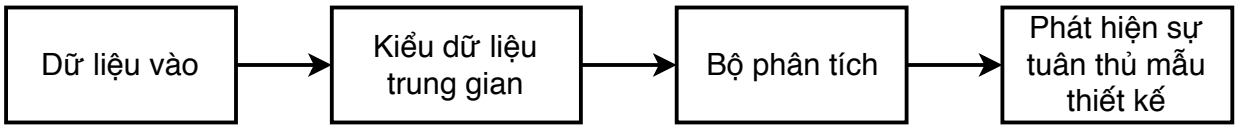
2.3.1 Các phương pháp kiểm tra sự tuân thủ mẫu thiết kế

Qua thời gian có nhiều phương pháp kiểm tra sự tuân thủ mẫu thiết kế được đề xuất. Một số phương pháp dựa trên việc xác định sự tương tự về kiến trúc vi mô giữa mã nguồn với các mẫu thiết kế (micro-architectures similar [3]), một số phương pháp thì cố gắng tiếp cận theo hướng đồ thị [4], ngoài ra một số phương pháp dựa trên việc phân tích thành phần mã nguồn đồng thời áp dụng các phương pháp học máy để khai thác đặc trưng từ những mẫu thiết kế. Những phương pháp dựa trên xác định sự tương tự về kiến trúc vi mô tập trung vào xác định mối quan hệ giữa các thành phần trong mã nguồn ở các mức trừu tượng khác nhau, từ đó mô hình hóa về mã nguồn về những dạng dữ liệu toán học ví dụ như ma trận, nhằm phục vụ việc tính toán sự tương tự giữa mã nguồn đối với mẫu thiết kế. Mặt khác, nhưng phương pháp tiếp cận theo hướng đồ thị sẽ cố gắng xác định nhưng phụ thuộc đặc trưng giữa các

thành phần của mẫu thiết kế đối với từng mã nguồn, nhằm mô hình hóa mẫu thiết kế về dạng đồ thị phụ thuộc thành phần sau đó, áp dụng học thuyết về đồ thị để giải quyết bài toán. Trong những phương pháp trên, phương pháp tiếp cận dựa trên đồ thị là nổi trội nhất, phương pháp này tập trung vào việc phân tích tĩnh cấu trúc của chương trình từ đó xác định nhưng mối quan hệ đặc trưng giữa các đối tượng bên trong mã nguồn và xây dựng đồ thị phụ thuộc.

2.3.2 Tổng quan quy trình kiểm tra sự tuân thủ mẫu thiết kế

Tuy có nhiều phương pháp kiểm tra sự tuân thủ mẫu thiết kế đã được đề xuất, nhưng nhìn chung những phương pháp này đều đi theo một quy trình tương tự nhau. Quá trình kiểm tra sự tuân thủ mẫu thiết kế trong mã nguồn thực chất là quá trình kiểm tra sự tồn tại của mẫu thiết kế bên trong mã nguồn. Hình 2.3.2 mô tả qui trình kiểm tra sự tuân thủ mẫu thiết kế. Hầu hết những phương pháp kiểm tra sự tồn tại mã



Hình 2.1: Quy trình kiểm tra sự tồn tại của mẫu thiết kế

nguồn đề trải qua bốn giai đoạn, xác định kiểu dữ liệu đầu vào, xác định và xây dựng kiểu dữ liệu trung gian, tiến hành phân tích kiểu dữ liệu trung gian, áp dụng một số thuật toán hoặc không để đưa ra sự công nhận về sự tồn tại hay không của mẫu thiết kế trong mã nguồn.

2.4 Bộ công cụ JCIA-VT

JCIA-VT [5] là bộ công cụ phân tích và đảm bảo chất lượng mã nguồn cho các dự án J2EE sử dụng các framework phổ biến như Hibernate, Struts, Spring. Cụ thể, JCIA-VT cung cấp các tiện ích về phân tích ảnh hưởng mã nguồn, phân tích luồng

dữ liệu, phân tích độ phức tạp mã nguồn. Thêm vào đó, ở khóa luận này sẽ xây dựng và tích hợp thêm công cụ phát hiện sự tuân thủ mẫu thiết kế của các dự án sử dụng Java.

JCIA-VT áp dụng phân tích mã nguồn tĩnh để xây dựng một cấu trúc dữ liệu gọi là đồ thị phụ thuộc (Java Dependency Graph - JDG) nhằm thể hiện sự phụ thuộc giữa các thành phần trong mã nguồn dự án J2EE, sau đó sử dụng JDG để dự đoán thành phần của mã nguồn bị ảnh hưởng trong sự thay đổi của một tập hợp các thành phần mã nguồn. Ngoài ra, công cụ còn cung cấp các tính năng về phân tích luồng dữ liệu, phân tích cấu trúc cơ sở dữ liệu, phân tích độ phức tạp của dự án J2EE.

Hiện tại JCIA-VT triển khai dưới dạng ứng dụng Web trên nền trang Java Spring và JavaServer Faces. Người dùng có thể sử dụng mà không cần cài đặt thêm bất cứ phần mềm gì. Hiện nay, công cụ này sẽ tiếp tục được bảo trì và phát triển thêm những tính năng nhằm tạo nên bộ công cụ hoàn chỉnh phục vụ cho công tác phân tích và đảm bảo chất lượng mã nguồn.

2.5 Công cụ phân tích Java

Công cụ phân tích Java (Java Parser - JP) [1] là thư viện mã nguồn mở được chứng nhận bởi JetBrains [2]. JP cung cấp các plug-in hỗ trợ *parse*, *analyze*, *transform*, *generate* đối với mã nguồn Java. Ngoài ra, JP cung cấp những API hỗ trợ phân tích mã nguồn. Người dùng hoàn toàn có thể sử dụng JP để phát triển những công cụ mới. Trong khóa luận này, JP sẽ được sử dụng để hỗ trợ quá trình tiền xử lý mã nguồn trong công cụ mà khóa luận sẽ xây dựng.

Sử dụng JP cho hoạt động phân tích mã nguồn, với đầu vào là mã nguồn và đầu ra sẽ là cây cú pháp trừu tượng tương ứng (Abstract Syntax Tree - AST) tương ứng với mã nguồn.

Đoạn mã nguồn 2.1 là một ví dụ về việc sử dụng API của JP để sinh cây cấu trúc từ mã nguồn.

[1] https://javaparser.org/
[2] https://www.jetbrains.com/

Mã nguồn 2.1: Sử dụng API của JavaParser để sinh cây cú pháp trừu tượng từ mã nguồn

```
CompilationUnit compilationUnit = JavaParser.parse(new
    File(rootNode.getAbsolutePath()));

compilationUnit.accept(new VoidVisitorAdapter<Void>() {
   @Override
   public void visit(ClassOrInterfaceDeclaration n, Void arg) {
      n.accept(new VoidVisitorAdapter<Void>() {
      @Override
      public void visit(MethodDeclaration n, Void arg) {
      }
      @Override
      public void visit(FieldDeclaration n, Void arg) {
      }
   }
```

Hơn thế nữa, JP cũng cung cấp các API để ta thực hiện truy vấn tới những thành phần của một lớp trên cây AST, mối số phương thức mà API của JP cung cấp như:

- `public void visit(FieldDeclaration n, Void arg)`: Duyệt tới thuộc tính của một lớp.
- `public void visit(MethodDeclaration n, Void arg)`: Duyệt tới từng phương thức của một lớp, tại đây những đối tượng thể hiện cho phương cũng cung cấp các API để truy cập tới thành phần nhỏ hơn trong phương thức như tên phương thức, kiểu trả về, v.v.
- `public void visit(ClassOrInterfaceDeclaration n, Void arg)`: Duyệt tới một lớp, đồng thời đối tượng đại diện cho lớp sẽ cung cấp các phương thức truy cập tới kiểu lớp, tên, tính đóng gói.

Chương 3

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

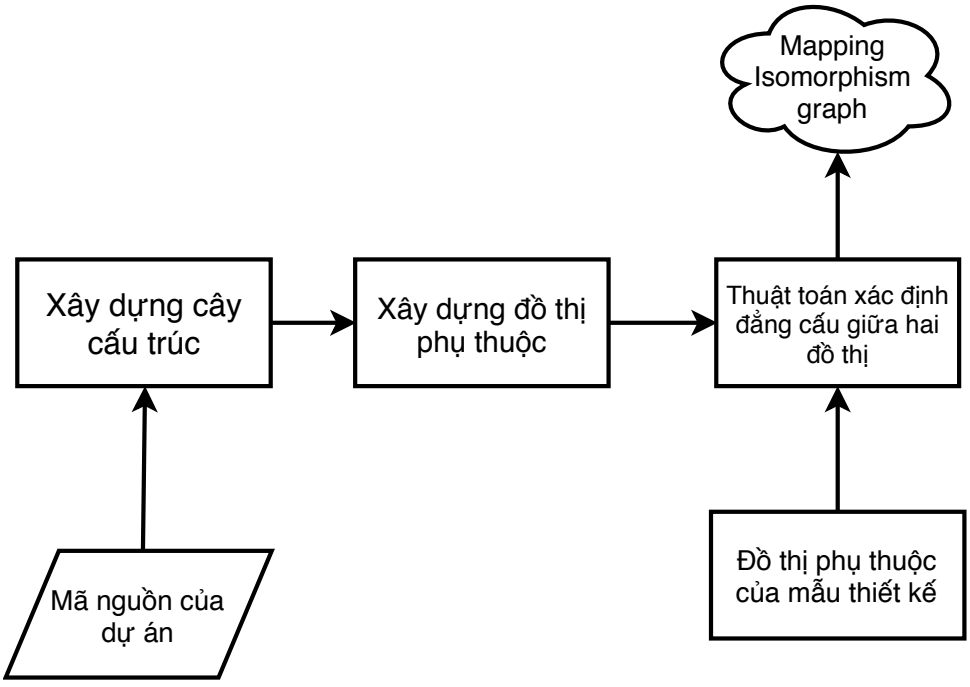
Mẫu thiết kế là tập hợp các luật nhằm mô tả cách giải quyết một vấn đề trong thiết kế có thể là vấn đề lặp lại nhiều lần trong dự án, là một khía cạnh của việc đảm bảo chất lượng mã nguồn, với những dự án công nghệ thông tin nói chung và dự án Java nói riêng. Ở các mẫu thiết kế hướng đối tượng, tập hợp 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 về mặt cấu trúc của mã nguồn. Do đó, phương pháp phát hiện sự tuân thủ mẫu thiết kế cho các dự án sử dụng Java mà khóa luận này đề xuất, sẽ đi theo hướng phân tích cấu trúc của mã nguồn.

3.1 Tổng quan phương pháp

Phương pháp mà khóa luận đề xuất, dựa trên hoạt động phân tích mã nguồn tĩnh của dự án. Ưu điểm của phương pháp phân tích này đó là đầu vào của phương pháp

không nhất thiết là toàn bộ mã nguồn, có thể là một phần của mã nguồn dự án, việc phân tích mã nguồn tĩnh đem lại sự linh hoạt và đơn giản trong quá trình phân tích. Cụ thể, khóa luận sẽ dựa trên phân tích tĩnh cấu trúc mã nguồn để tiền xử lý đầu vào và tiếp cận bài toán kiểm tra sự tuân thủ mẫu thiết kế theo hướng đồ thị. Đầu vào của phương pháp sẽ là mã nguồn, mã nguồn sẽ được tiến hành tiền xử lý và xây dựng cây cấu trúc, cây cấu trúc tiếp tục được phân tích để xác định phụ thuộc giữa các thành phần trên cây. Từ cây cấu trúc với những phụ thuộc đã xác định được, một đồ thị phụ thuộc sẽ được xây dựng từ cây cấu trúc. Đồ thị phụ thuộc sẽ là đầu vào cho hoạt động kiểm tra sự tồn tại của mẫu thiết kế bên trong mã nguồn. Từ đó, đưa ra kết luận về sự tuân thủ đầy đủ mẫu thiết kế của mã nguồn so với đặc tả ban đầu.

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 mẫu 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

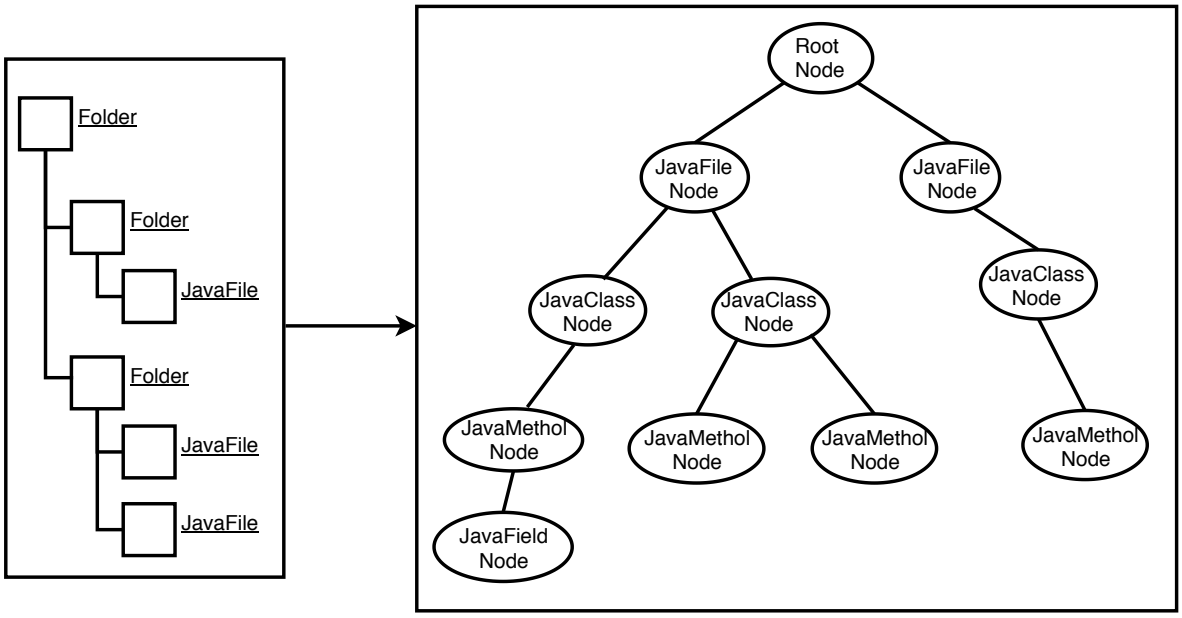
Đối với phương pháp kiểm tra sự tuân thủ mẫu thiết kế mà khóa luận này đề xuất. Việc trực tiếp phân tích mã nguồn đầu vào là một giải pháp không tối ưu do sự phức tạp của mã nguồn, các thành phần không được sử dụng tới của mã nguồn ví dụ như các tệp tin xml, yml, v.v. Do đó, cần có một kiểu dữ liệu tường 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 đề 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 quản lý các đối tượng là dễ dàng nhằm mục đích phục 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* [6]) 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.*

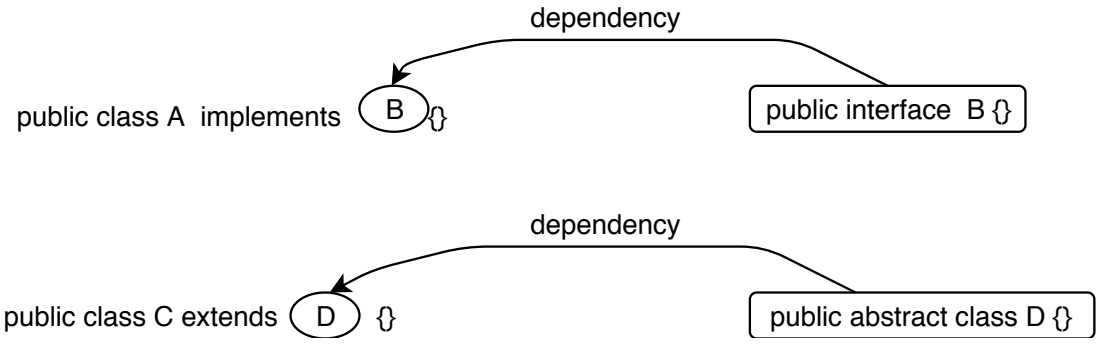
Trong khóa luận này xác định, thành phần của một lớp gồm bốn phần chính: *Class type, Class dependency, Class variables, Method* . Trong đó:

- *Class type*: thể hiện lớp đó 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 một *Method*

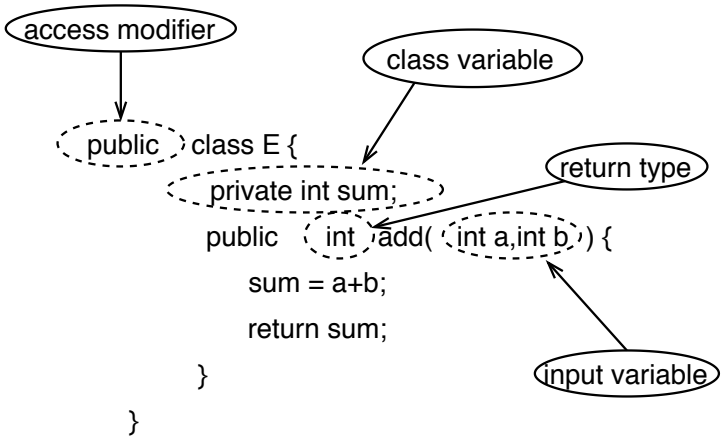
Để rõ ràng hơn về quá trình xác định thuộc tính cho các nút của cây cấu trúc, ta tiến hành xem xét các ví dụ sau.

Hình 3.3 mô tả hai loại phụ thuộc kế thừa qua phương thức *Extends* và *Implements*. Trong đó A là một Class thừa kế từ B là một *Interface* qua phương thức *Extends*, C là một class thừa kế D qua phương thức *Implements* với D là một *Abstract Class*. Do đó giữa các lớp A và B , C và D tồn tại phụ thuộc với nhau.



Hình 3.3: Phụ thuộc thừa kế của lớp

Hình 3.4 mô tả thành phần cơ bản của một *Class*. Cụ thể, *Class* E bao gồm các thành phần, *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'. Ngoài ra, nêu một lớp (Class) không định nghĩa gì thêm nó sẽ có hàm khởi tạo mặc định. Khi tạo một đối tượng mới, giá trị của những *Class Variable* sẽ không được khởi tạo.



Hình 3.4: Các thành phần cơ bản trong class

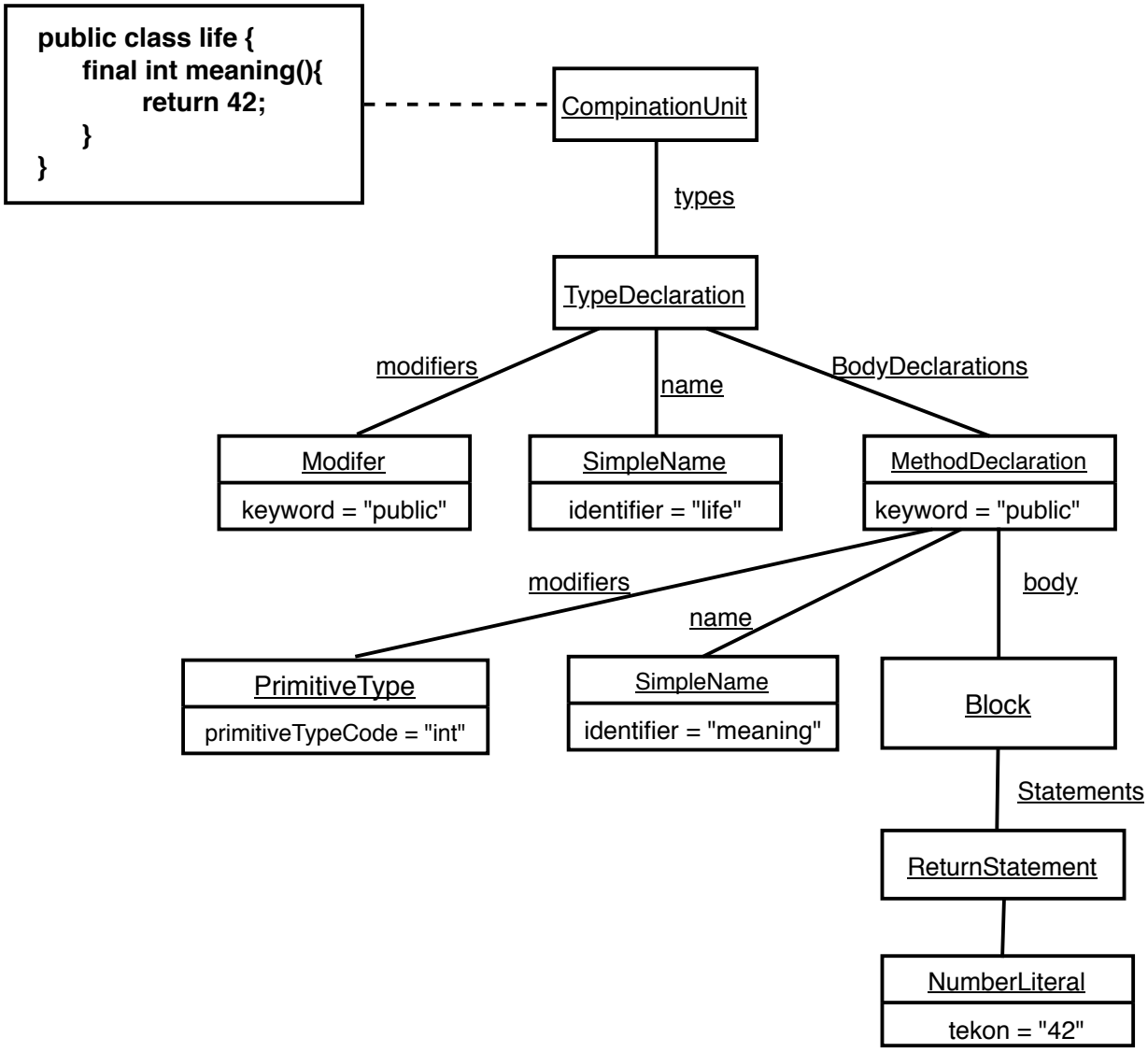
Bảng 3.1 mô tả đầy đủ những thông tin cần xác định cho mỗi loại nút trên cây cấu trúc, nhằm phục vụ cho việc phân tích cấu trúc mã nguồn và xây dựng đồ thị phụ thuộc sẽ được trình bày tại mục **3.2** và **3.3** của chương này.

Bảng 3.1: Thuộc tính trên mỗi nút

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

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 thông qua những API mà JP cung cấp. 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 bởi 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 thức 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 với 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 thức 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 như 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 báo 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 đối 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 liệt kê tất cả trong Bảng 3.2 và Bảng 3.3 [2].

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.

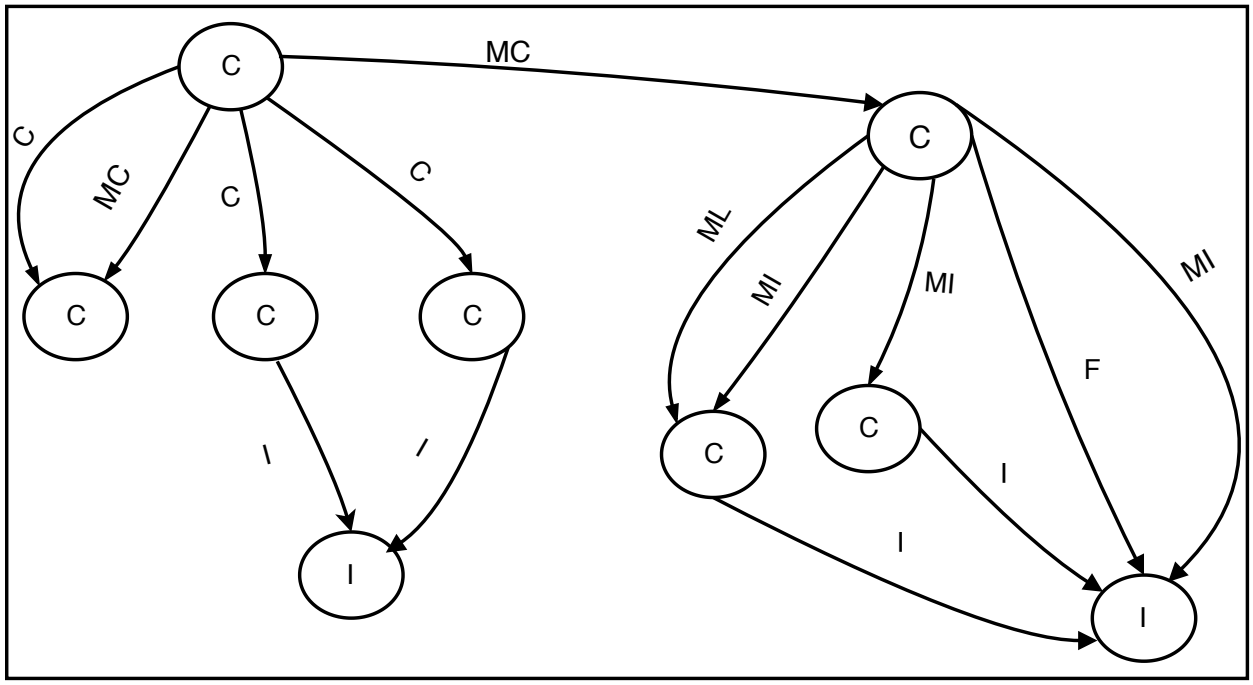
Thuật toán 3.1: *JavaDependencyAnalyze(Root)*

```
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 ∈ 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 đồ thị phụ thuộc sinh ra từ một đoạn mã nguồn. Cụ thể, mã nguồn đầu vào sẽ được tiền xử lý và xây dựng cây cấu trúc, cây cấu trúc sẽ được phân tích nhằm xác định các phụ thuộc giữa các thành phần trong mã nguồn. Ta sẽ được một cây cấu trúc với nút gốc là các thành phần mã nguồn như class, method, v.v. cạnh giữa hai nút lá là tập các phụ thuộc giữa hai nút gốc. Từ cây cấu trúc đã có đầy đủ phụ thuộc giữa các thành phần mã nguồn,

một đồ thị phụ thuộc sẽ được xây dựng. Đồ thị phụ thuộc sẽ có dạng như Hình 3.10. Trong đó, đỉnh của đồ thị biểu 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 cho hoạt động 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 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. Do 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 [8]:

- 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 các 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 ánh 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.

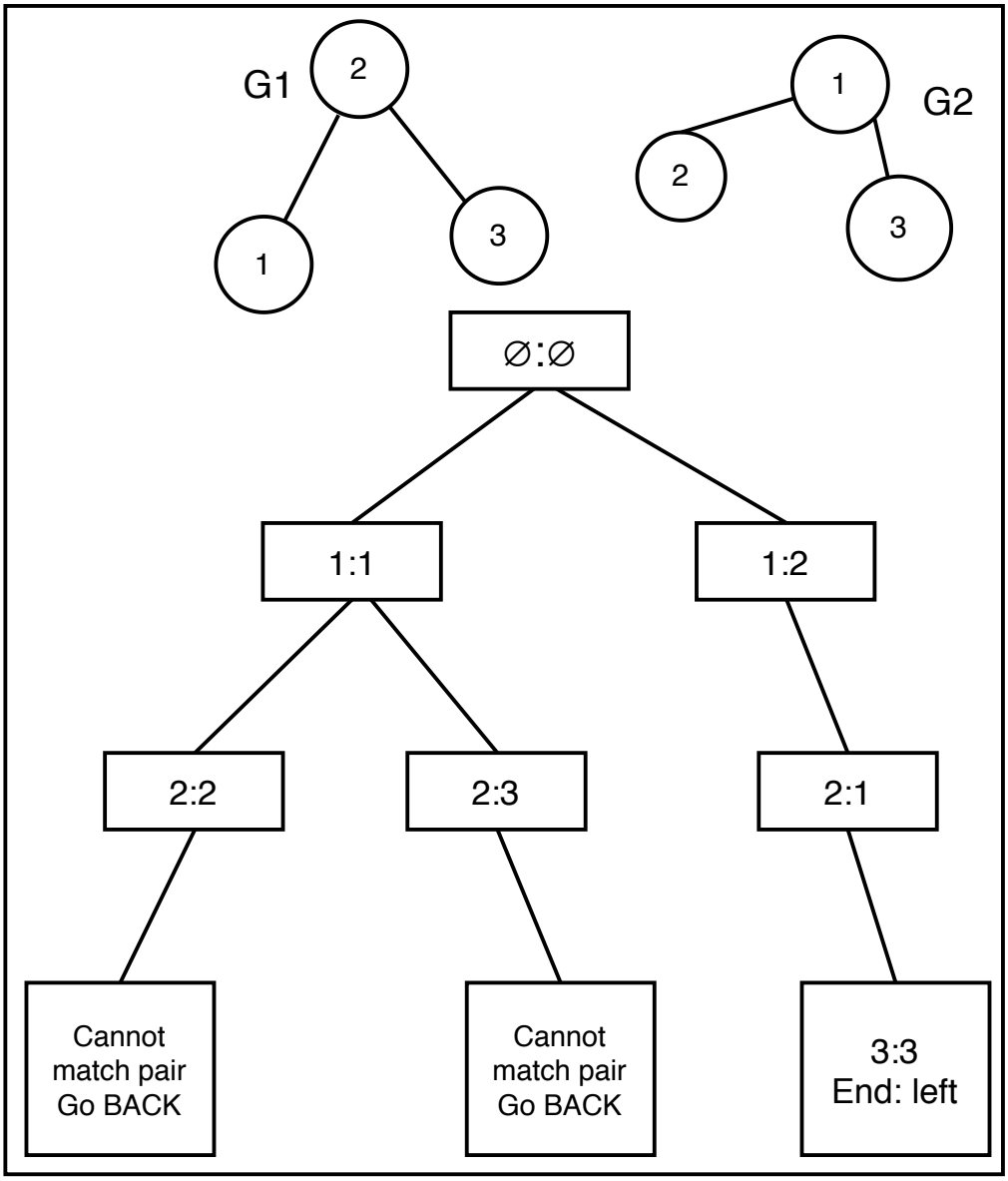
Thuật toán 3.2: Xây dựng đồ thị từ cây cấu trúc

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 $n \in N$ **do**
    D.$add(n.getDepedencies())$;
G.$addNode$(N);
foreach $d \in D$ **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 thực tế là quá trình kiểm tra sự tồn tại của những mẫu thiết kế mà được yêu cầu áp dụng trong mã nguồn theo đặc tả của dự án, những mẫu thiết kế này có tồn tại trong mã nguồn dự án hay không. Quá trình kiểm tra sự tồn tại của mẫu thiết kế trong mã nguồn

chính là quá trình 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 đối 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 [9]:

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 [9] sẽ được trình bày chi tiết ở Thuật toán 3. Trong đó, ta có trạng thái ban đầu là tập $M(s_0) = \emptyset$ và trạng thái s_0. Trong mỗi vòng lặp tính toán, thuật toán sẽ lựa chọn cặp đỉnh dựa trên sự thỏa mãn Feasibility Rules và Semantic Feasibility, mỗi một cặp định được thêm vào tập M sẽ chuyển trạng thái s sang trạng thái s'. Nếu tập M chưa toàn bộ đỉnh của G2 tức là G2 đẳng cấu với một đồ thị con của G1, ngược lại G1 và G2 không tồn tại quan hệ đẳng cấu.

Thuật toán 3.3: $VF2$

```
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
```

Từ việc giải quyết được bài toán nhỏ kiểm tra sự tồn tại của mẫu thiết kế trong mã nguồn. Ta áp dụng vào hoạt động kiểm tra sự tuân thủ mẫu thiết kế trong mã nguồn dự án bằng cách kiểm tra sự tồn tại của toàn bộ mẫu thiết kế mà dự án yêu cầu từ đó đưa ra được kết quả về sự tuân thủ mẫu thiết kế của dự án.

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 kiểm thử và đảm bảo chất lượng mã nguồn, kiến trúc của công cụ mà khóa luận này xây dựng đồng thời tích hợp vào JCIA-VT. Thêm vào đó, đưa ra một số thực nghiệm 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 sử dụng mã nguồn Java, cây cấu trúc sẽ được xây dựng.

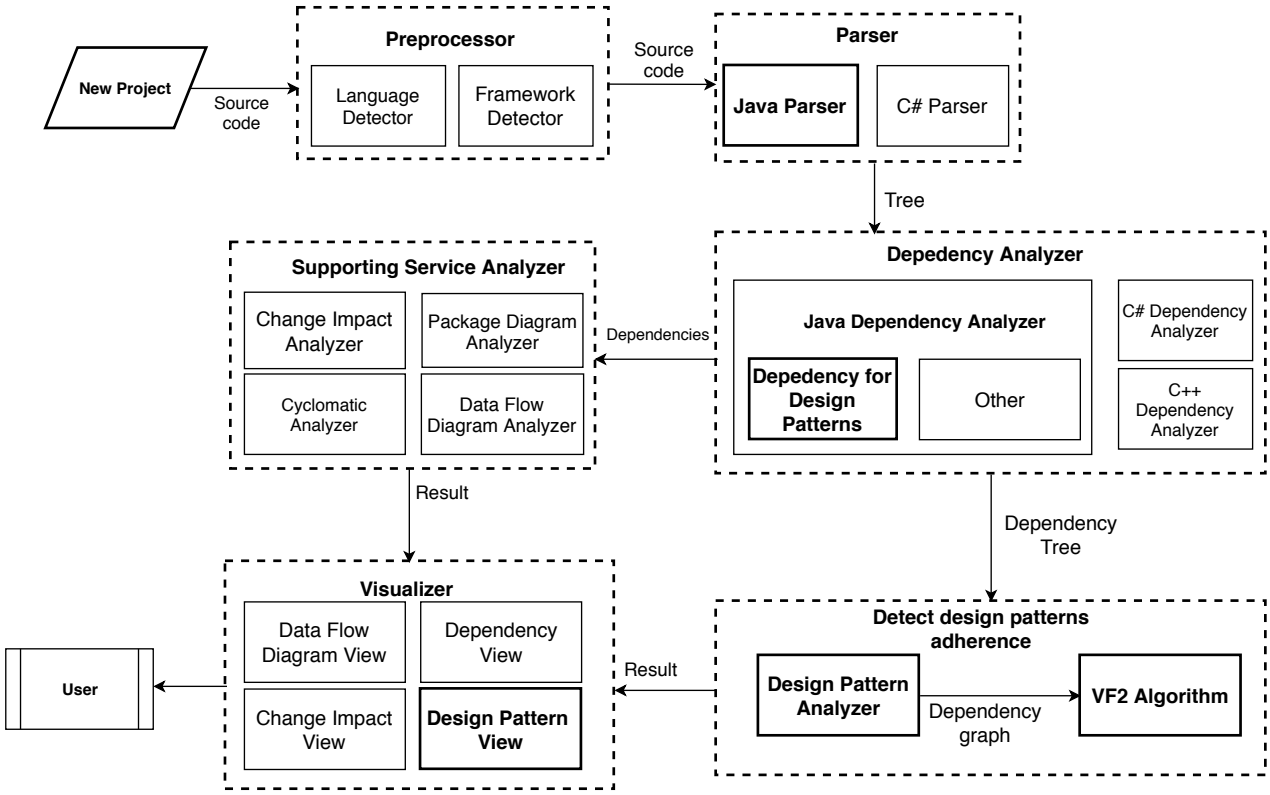
Dependency Analyzer:Bao gồm nhiều mô-đun nhỏ phụ trách phân tích phụ thuộc cho từng loại ngôn ngữ, công nghệ. Đầu vào là mã nguồn đã được phân tích từ mô-đun *Parser*, tùy vào mỗi công nghệ, ngôn ngữ mà dự án sử dụng mà mô-đun nào sẽ được gọi tới. Ví dụ: đối với dự án Java, mô-đun *Java Dependency Analyzer* 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 ra 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 nguồn (Cyclomatic Analyzer) .v.v.

Detect design patterns adherence: Là mô-đun phát hiện sự tuân thủ mẫu thiết kế trong mã nguồn mà khóa luận này xây dựng. Bên cạnh các mô-đun như *Java Parser*, *Dependency for Design Patterns*, *Design Patterns View* đảm nhận vai trò tiền xử lý và phân tích mã nguồn nhằm sinh đồ thị phụ thuộc. Thì mô-đun *Detect design patterns adherence* nhận đầu và là đồ thị phụ thuộc của mã nguồn và mẫu thiết kế, tiến hành kiểm tra sự tồn tại của những mẫu thiết kế bên trong mã nguồn, từ đó trả ra kết quả về sự tuân thủ mẫu thiết kế của mã nguồn so với đặc tả và thiết kế ban đầu.

Visualyzer: Mô-đun thể hiện đầu ra của mã của những tính năng từ JCIA-VT



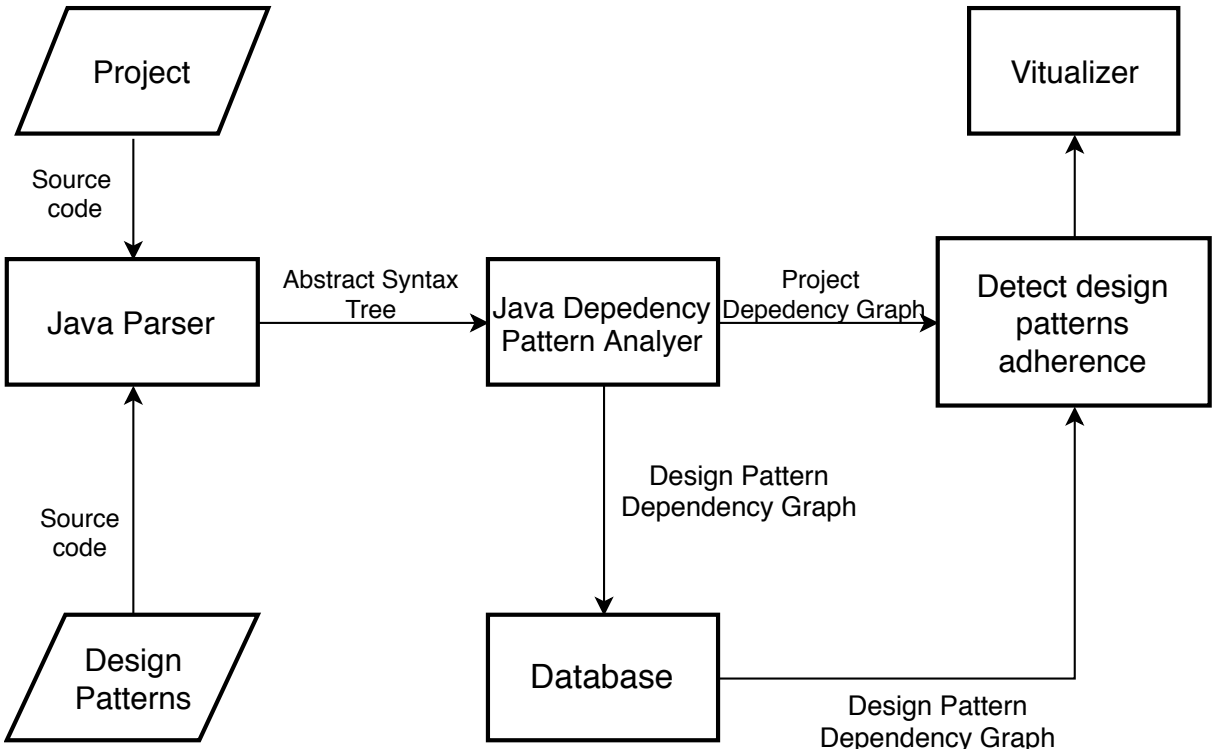
Hình 4.1: Tổng quan kiến trúc JCIA-VT

4.1.2 Kiến trúc và cài đặt công cụ kiểm tra sự tuân thủ mẫu thiết kế bên trong mã nguồn

Phần này sẽ trình bày về kiến trúc chi tiết của mô-đun kiểm tra sự tuân thủ mã nguồn trong bộ công cụ JCIA-VT.

Mô-đun được tích hợp trực tiếp bên trong mã nguồn Back-end của bộ công cụ JCIA-VT. 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:

Kiến trúc của công cụ bao gồm năm mô-đun chính bao gồm: xử lý dữ liệu đầu vào (preprocessor), phân tích mã nguồn (Java Design Pattern Analyzer), cơ sở dữ liệu (Database), phát hiện sự tuân thủ mẫu thiết kế (Detect design patterns adherence),



Hình 4.2: Kiến trúc của công cụ phát hiện sự tuân thủ mã nguồn

mô hình hóa đầu ra (Visualizer). Do quá trình tích hợp với bộ công cụ JCIA-VT mô-đun tiền xử lý dữ liệu đầu vào sẽ được tái sử dụng, ngoài ra những mô-đun khác của công cụ được trình bày sau đây sẽ được xây dựng mới và sẽ được tích hợp thành mội gói bên trong mã nguồn của bộ công cụ JCIA-VT.

- **Java Design Pattern Analyzer**: Là thành phần chịu trách nhiệm phân tích mã nguồn. Đầu vào là mã nguồn Java, đầu ra là đồ thị phụ thuộc của mã nguồn. Đầu tiên cây AST sẽ được sinh ra từ mã nguồn, tiếp đó quá trình xây dựng cây cấu trúc được thực hiện. Từ cây cấu trúc, việc phân tích phụ thuộc được tiến hành.

- **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. Phần này được xây dựng dựa trên gói *core.DAO* của bộ công cụ JCIA-VT

- **Detect design patterns adherence**: Chịu trách nhiệm kiểm tra sự tồn tại của

những 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. Mô-đun tiến hành kiểm tra sự tồn tại của toàn bộ những mẫu thiết kế bên trong mã nguồn, từ đó trả ra kết quả về sự tuân thủ những mẫu thiết kế của mã nguồn so với đặc tả và thiết kế ban đầu. Dữ liệu trả về cho mô-dun Visiualier thông qua giao thức API.

- **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]. Giao tiếp

4.2 Triển khai

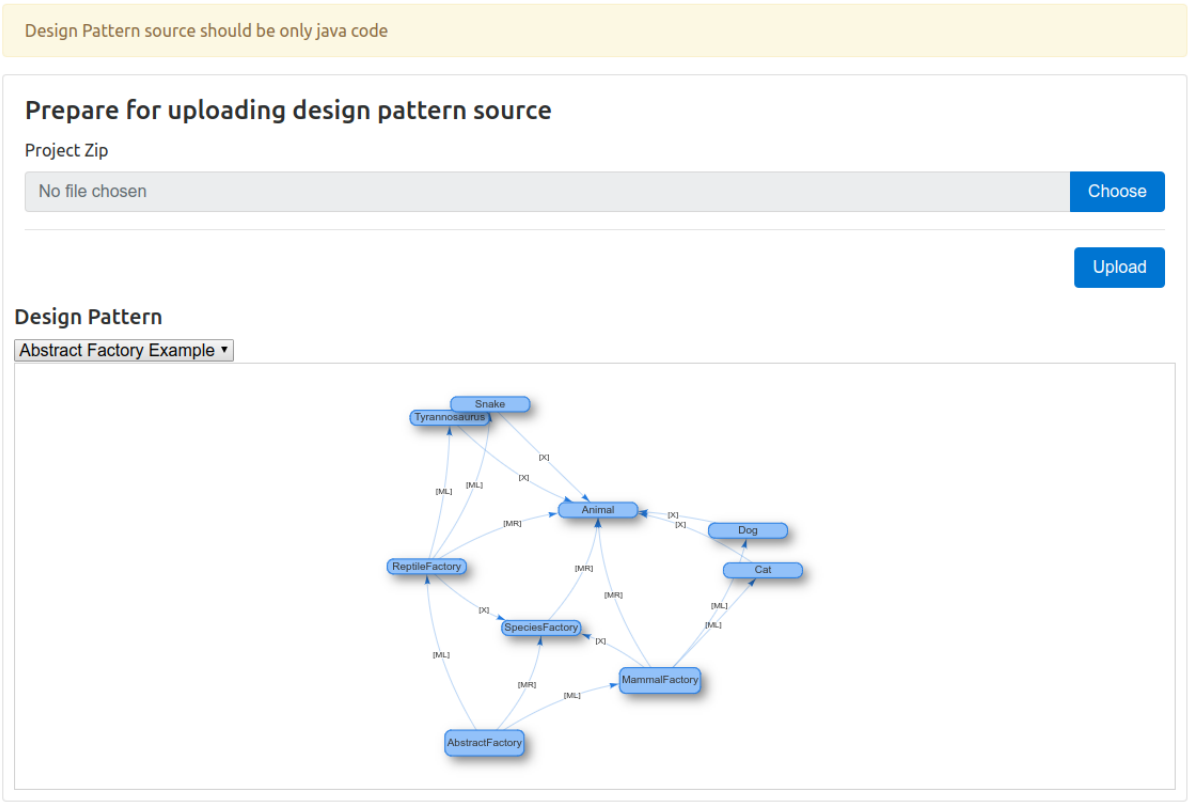
Hiện tại công cụ đã được phát triển và đặt, trên một phiên bản mới của JCIA-VT. Cung cấp những giao diện và tính năng mới liên quan tới việc kiểm tra sự tuân thủ của mẫu thiết kế bên trong mã nguồn.

4.2.1 Giao diện tải lên 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 lên, 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 lên một mẫu thiết kế mới.
Giao diện tải lên mẫu thiết kế cung cấp hai tính năng chính. Thứ nhất, tải lên và quản lý mẫu thiết kế. Khi người dùng tải lên mẫu thiết kế mới, mã nguồn của mẫu thiết kế sẽ được tiến hành xử lý để xây dựng đồ thị phụ thuộc thể hiện cấu trúc của mã nguồn sau đó được lưu trữ trên cơ sở dữ liệu nhằm phục vụ công việc kiểm tra sự tuân thủ mẫu thiết kế. Thứ hai: quản lý và hiển thị đồ thị phụ thuộc của mẫu thiết

[1] http://visjs.org/

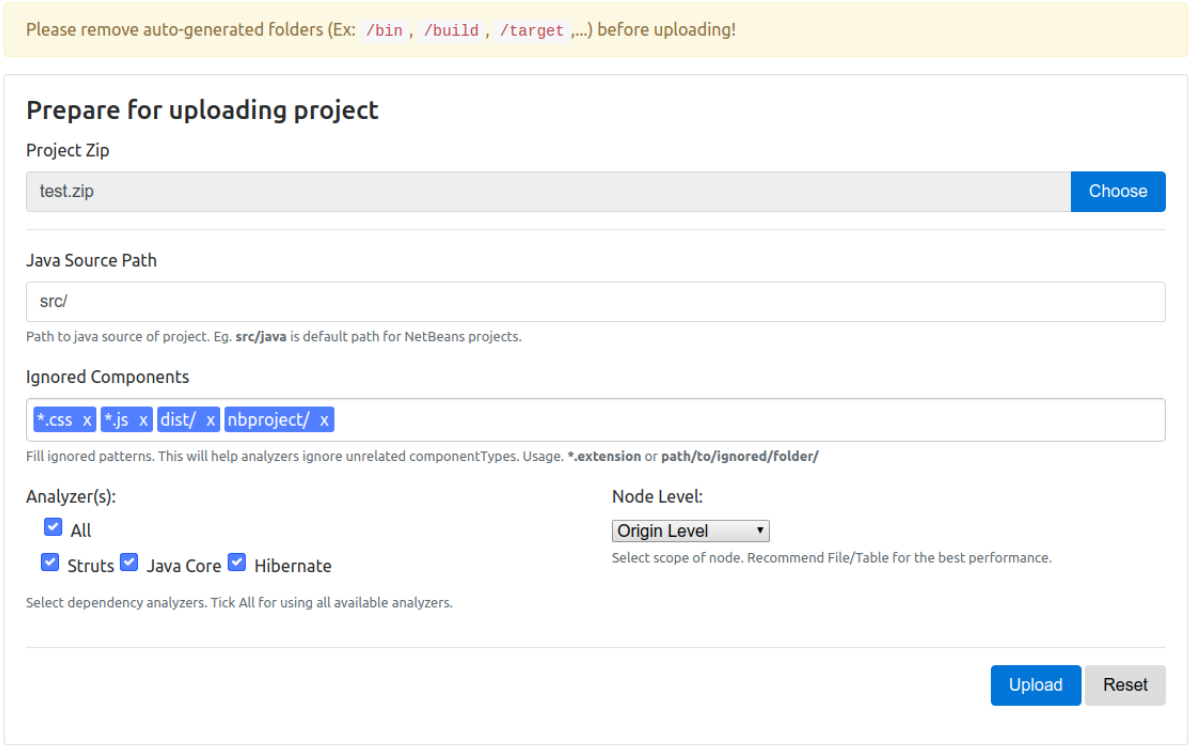


Hình 4.3: Giao diện tải lên mẫu thiết kế

kế, người dùng có thể thêm hoặc xóa mẫu thiết kế, đồng thời giao diện cũng cung cấp cửa sổ hiển thị đồ thị phụ thuộc của mẫu thiết kế.

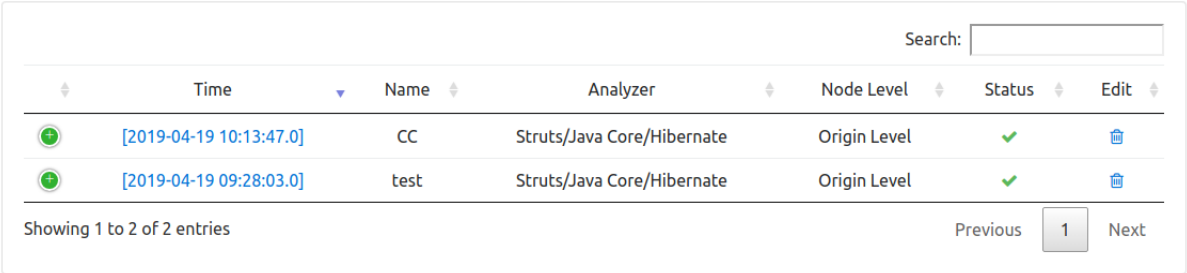
4.2.2 Giao diện phát hiện mẫu thiết kế bên trong mã nguồn

Sau khi tiến hành tải lên những mẫu thiết kế được yêu cầu trong việc phát triển mã nguồn. Ta tiến hành phân tích mã nguồn để kiểm tra sự tuân thủ của mã nguồn đối với những mẫu thiết kế đã được qui định. Ở phân này, ta sẽ trình bày lần lượt từng bước một theo luồng điều khiển, bao gồm quá trình tải liên mã nguồn dự án, quá trình phân tích ảnh hưởng và quá trình kiểm tra sự tuân thủ mẫu thiết kế bên trong mã nguồn của dự án. Đầu tiên, mã nguồn sẽ được tiến hành tải nên, tại giao diện tải lên mã nguồn dự án như Hình 4.4. Cần thêm được dẫn tới thư mục chứa mã nguồn như trong giao diện mô tả.



Hình 4.4: Giao diện tải lên mã nguồn dự án

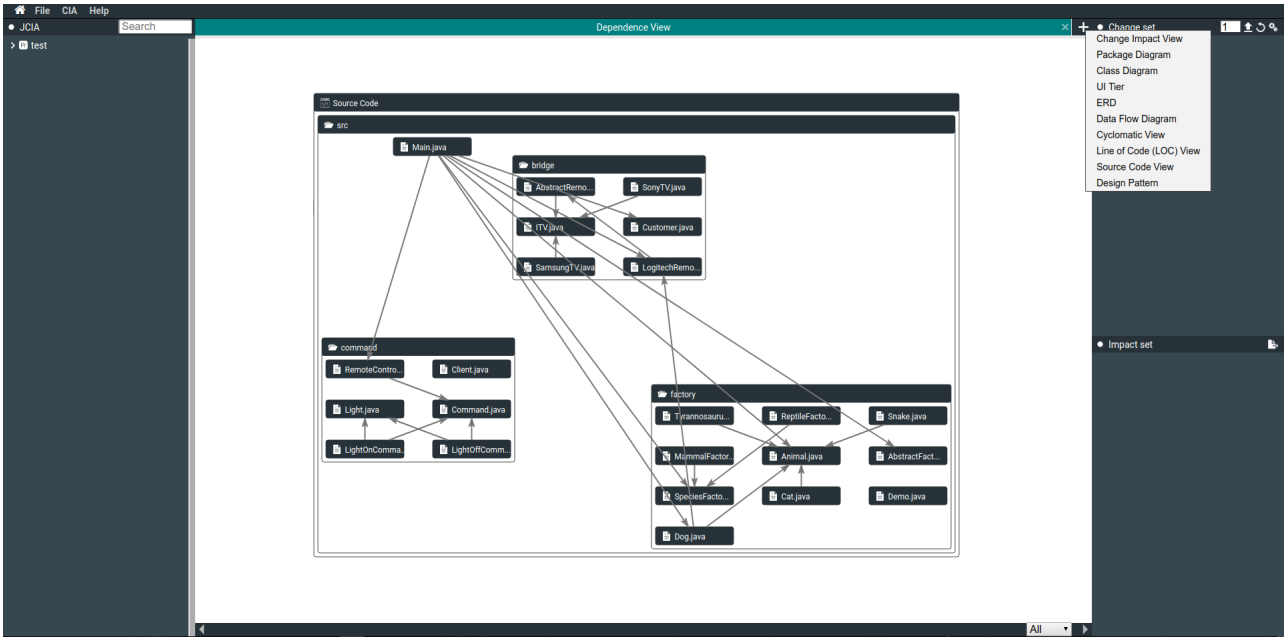
Sau khi nhấn chọn *Upload*, hệ thống chuyển tới giao diện lịch sử và đồng thời tiến hành phân tích mã nguồn dự án. Giao diện lịch sử sẽ được thể hiện như trong Hình 4.5. Bao gồm thông tin về quá trình phân tích mã nguồn, thông tin về mã nguồn của các dự án khác nhau.



Hình 4.5: Giao diện màn hình lịch sử

Quá trình phân tích mã nguồn hoàn thành, giao diện làm việc chính của JCIA-VT sẽ hiển thị khi người dùng chọn một trong các dự án tại danh sách được hiển thị như

trên hình 4.5. Giao diện làm việc chính của JCIA-VT bao gồm nhiều màn hình khác nhau.

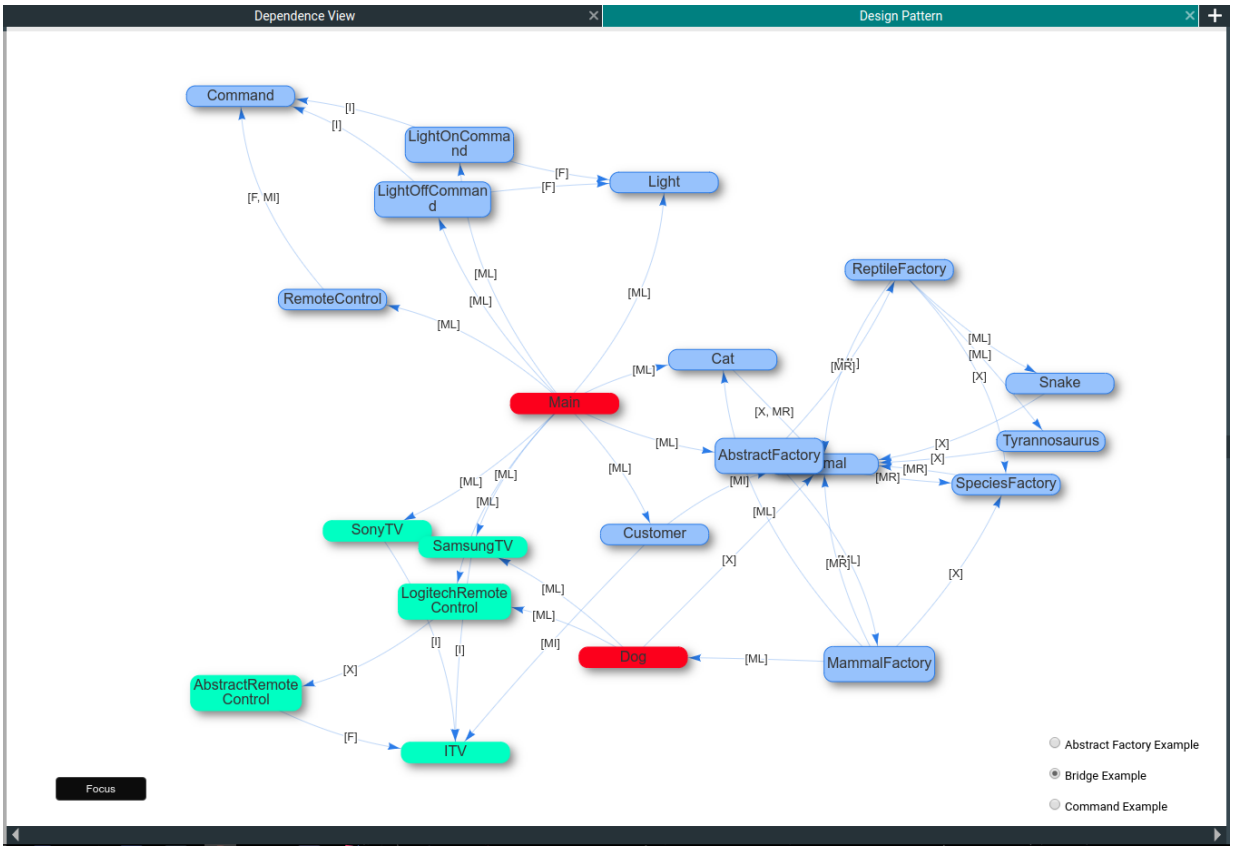


Hình 4.6: Giao diện chính và danh sách tính năng của JCIA-VT

Mỗi màn hình đại diện cho một tính năng, *Dependency view* là màn hình mặc định trong giao diện làm việc, những màn hình làm việc còn lại được cung cấp tại danh mục tùy chọn, thể hiện như trong Hình 4.6

Tiếp theo, từ màn hình chính tại danh mục tính năng, ta lựa chọn *Design Pattern* để chuyển tới màn hình làm việc của tính năng *Kiểm tra sự tuân thủ mẫu thiết kế bên trong mã nguồn*. Hình 4.7 mô tả màn hình làm việc của tính năng *Kiểm tra sự tuân thủ mẫu thiết kế*

Bao gồm, danh sách mẫu thiết kế được sử dụng trong mã nguồn dự án mà công cụ phát hiện được, đồ thị phụ thuộc của mã nguồn dự án. Trong đó, phần màu xanh của đồ thị là phần của một mẫu thiết kế mà công cụ phát hiện mã nguồn có sử dụng mẫu thiết kế đó, phần màu đỏ thể hiện phần mã nguồn liên quan tới mẫu thiết kế đó, tức là khi mẫu thiết kế thay đổi thì sẽ ảnh hưởng tới phần mã nguồn màu đỏ.

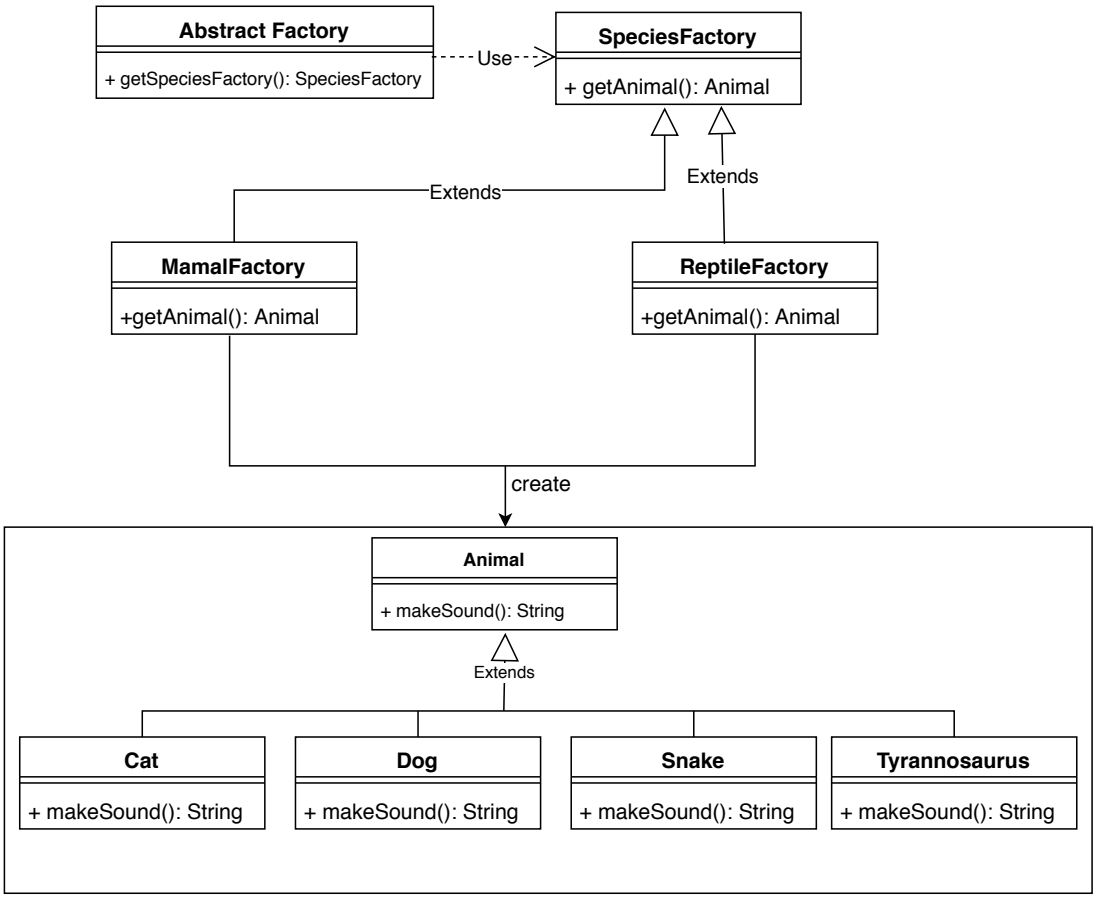


Hình 4.7: Màn hình tính năng kiểm tra sự tuân thủ mã nguồn

4.3 Thực nghiệm

Trong phần này của khóa luận, sẽ tiến hành thực nghiệm công cụ đã được xây dựng. Tập dữ liệu thử nghiệm bao gồm mã nguồn có áp dụng những mẫu thiết kế hướng đối tượng được sử dụng phổ biết như Abstract Factory, Composite, Bridge, Adapter, Proxy v.v. Ta sẽ trình bày một số ví dụ đã được sử dụng để tiến hành thực nghiệm trong quá trình xây dựng công cụ.

Ví dụ 1: Ví dụ này sẽ tiến hành thực nghiệm dựa trên một đoạn mã nguồn sử dụng Java đã được yêu cầu sử dụng ba mẫu thiết kế là Abstract Factory, Bridge, Command trong quá trình phát triển dự án. Hình 4.8 mô tả biểu đồ lớp của mẫu Abstract Factory mà ta sẽ cần áp dụng trong mã nguồn ứng dụng.



Hình 4.8: Abstract Factory Class Diagram cho ví dụ 1

Điều kiện ban đầu: Mã nguồn được yêu cầu áp dụng ba mẫu thiết kế là Abstract Factorym Bridge, Command.

Các bước thực hiện:

`Bước 1`: Triển khai (Implement) ba mẫu thiết kế và thực hiện tải lên JCIA-VT

`Bước 2`: Áp dụng ba mẫu thiết kế vào đoạn mã nguồn sẽ viết, sau đó phân tích đoạn mã nguồn đó với JCIA-VT để kiểm tra sự tuân thủ mẫu thiết kế.

Kết quả mong đợi: Công cụ sẽ kiểm tra sự tuân thủ mẫu thiết kế trong mã nguồn ứng dụng, phát hiện được ba mẫu thiết kế đã được áp dụng.

Thực hiện:

Bước 1: Triển khai ba mẫu thiết kế. Hình 4.8 mô tả cấu trúc của kiểu mẫu thiết kế Abstrac Factory. Ta sẽ viết mã Java triển khai lại mẫu thiết kế như trong biểu đồ lớp

Hình 4.8. Thực hiện như vậy với cả ba mẫu thiết kế, sau đó tải chúng lên cơ sở dữ liệu mẫu thiết kế của bộ công cụ JCIA-VT, mỗi dự án sẽ có những bộ mẫu thiết kế khác nhau.

Bước 2: Sau khi áp dụng ba mẫu thiết kế kể trên vào trong mã nguồn, tiến hành phân tích mã nguồn bằng JCIA-VT, để kiểm tra sự tồn tại của những mẫu thiết kế đó trong mã nguồn và những mẫu thiết kế đó sẽ ảnh hưởng tới những thành phần nào trong mã nguồn. Từ đó ta sẽ đưa ra được kết luật về sự tuân thủ mẫu thiết kế bên trong mã nguồn.

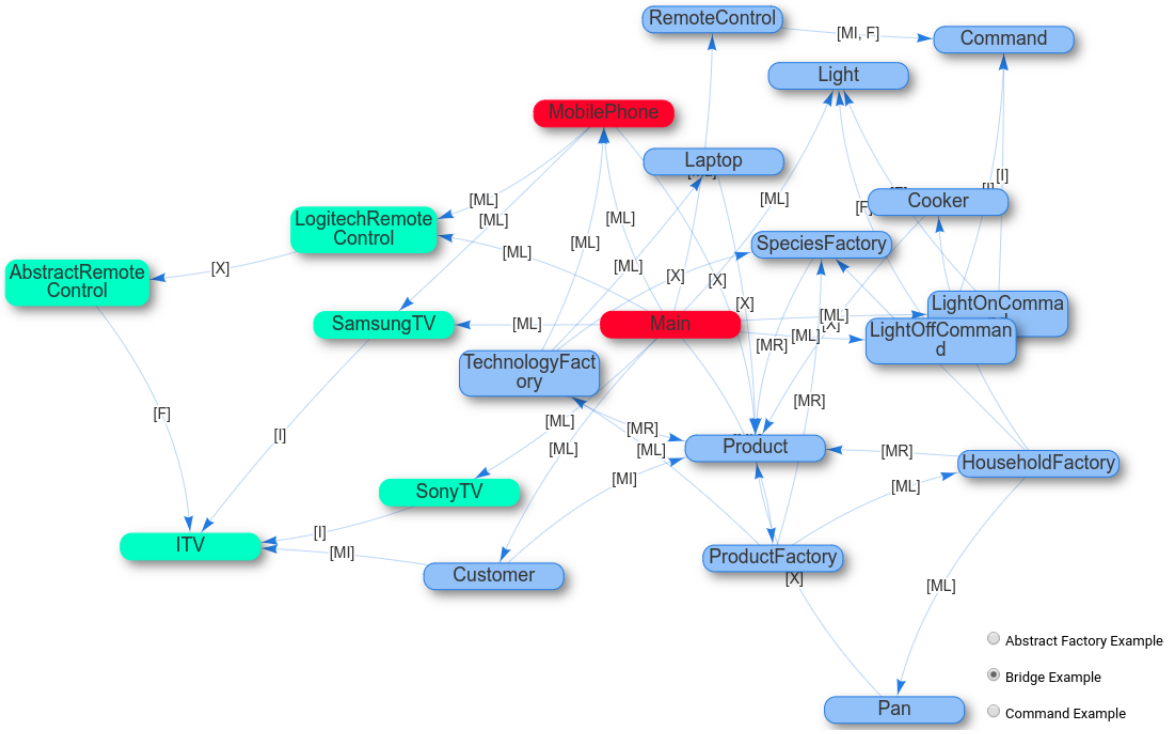
Kết quả: Sau khi tiến hành phân tích, kết quả được trả ra như Hình 4.9. Trong đó, màn hình có hai phần chính bao gồm: danh sách những mẫu thiết kế mã nguồn đã áp dụng theo như đặc tả của dự án, đồ thị phụ thuộc của mã nguồn dự án. Về phần đồ thị phụ thuộc mã nguồn, được chia thành ba màu chính, màu xanh non thể hiện phần mẫu thiết kế mà trong mã nguồn đã tuân thủ như đặc tả của dự án, phần màu đỏ là những thành phần liên quan tới mẫu thiết kế trong quá phát triển mã nguồn, phàn đồ thị màu xanh nước biển thể hiện những thành phần mã nguồn còn lại.

Từ màn hình kết quả ta có thể đưa ra kết luận. Mã nguồn đã hoàn toàn tuân thủ theo đặc tả thiết kế ban đầu. Thể hiện bằng thông qua việc cả ba mẫu thiết kế được yêu cầu áp dụng đề đã tồn tại trong mã nguồn dự án.

Ví dụ 2: Sẽ thực hiện với kịch bản tương tự tại ví dụ 1, nhưng tại mã nguồn cần áp dụng mẫu Abstract Factory, ta đã áp dụng mẫu này khác so với mẫu Abstract Factory đã được tải lên cơ sở dữ liệu mẫu thiết kế của JCIA-VT. Tức là, về tính chất thì mã nguồn đã áp dụng đúng tính chất của mẫu Abstract Factory nhưng về mặt cấu trúc mã nguồn do khác so với cấu trúc mã nguồn đã tải lên cơ sở dữ liệu của JCIA-VT do vậy, công cụ sẽ không phát hiện được

Điều kiện ban đầu: Mã nguồn được yêu cầu áp dụng ba mẫu thiết kế là Abstract Factory, Bridge, Command. Trong đó mẫu Abstract Factory được áp dụng trong mã nguồn sẽ có cấu trúc khác với mẫu Abstract factory được yêu cầu như Hình 4.8. Hình 4.10 là thể hiện cấu trúc mẫu Abstract Factory sẽ được áp dụng trong mã nguồn.

Các bước thực hiện: Các bước thực hiện sẽ tương tự như ví dụ 1

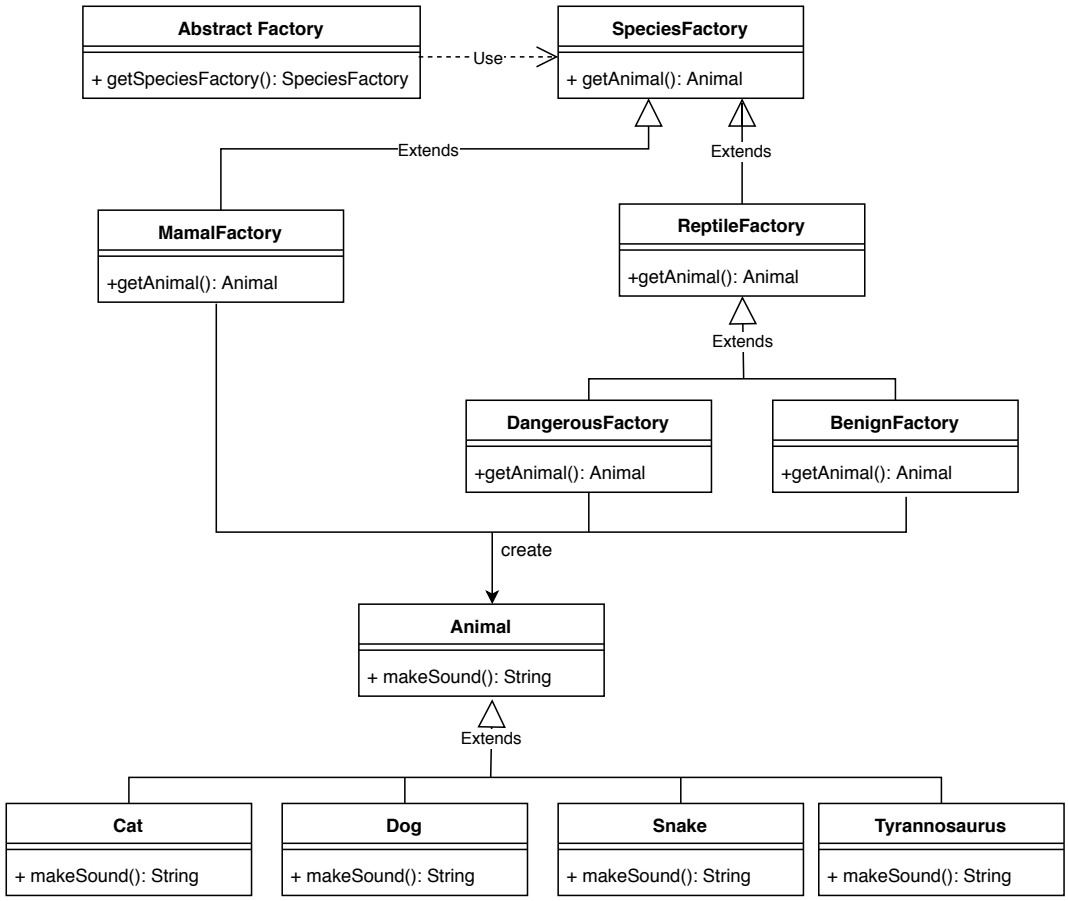


Hình 4.9: Kết quả phân tích ví dụ 1

Kết quả: Mẫu thiết kế Abstrac Factory sẽ không được phát hiện bởi công cụ.

Lý giải: Bản chất của việc phát hiện sự tuân thủ mẫu thiết kế là tiến hành kiểm tra mã nguồn có tồn tại những mẫu thiết kế như tài liệu đặc tả và thiết kế hay không. Phương pháp mà khóa luận này đề xuất tiếp cận theo hướng đồ thị. Tức là, ta có đồ thị phụ thuộc của mẫu thiết kế đã được triển khai trước đó và đồ thị phụ thuộc của toàn bộ mã nguồn ứng dụng. Công việc cần làm đó là tiến hành kiểm tra trong mã nguồn có tồn tại một đồ thị đẳng cấu với đồ thị của mẫu thiết kế hay không. Do vậy triển khai mẫu thiết kế ban đầu là quan trọng sao cho mẫu thiết kế đó là đúng và gọn gàng nhất, khi áp dụng mẫu thiết kế đó vào mã nguồn cần đảm bảo rằng trong mã nguồn áp dụng, mẫu thiết kế đó chỉ có xu hướng mở rộng so với định nghĩa ban đầu.

Trên đây là hai trong số nhiều ví dụ đã được thực hiện để kiểm tra tính đúng đắn của công cụ. Về cơ bản, công cụ hoạt động tốt với một số mẫu thiết kế thuộc *GoF (Gang of Fours) design patters* [1] như Abstract Factory, Builder, Command,... bên cạnh đó còn gặp một số hạn chế về trùng lặp giữa các mẫu thiết kế. Ngoài ra, công cụ còn hỗ trợ kiểm tra những mẫu thiết kế mà đội dự án tự định nghĩa.



Hình 4.10: Abstract Factory Diagram cho ví dụ 2

4.3.1 Thảo luận

Công cụ kiểm tra sự tuân thủ mẫu thiết kế do dự án sử dụng Java trong bộ công cụ JCIA-VT là một giải pháp cho việc đảm bảo chất lượng mã nguồn. Thực nghiệm, đã cho thấy tiềm năng của phương pháp mà khóa luận đề xuất. Với ưu thế là phát triển công cụ trên nền tảng Web cho phép người dùng sử dụng ngay trên những trình duyệt mà không phải cài đặt thêm bất cứ thứ gì. Điều này thúc đẩy khả năng tiếp cận của công cụ tới người dùng.

Đối với những dự án phần mềm, hoạt động bảo trì và nâng cấp diễn ra liên tục. Do đó, cần phải thực hiện kiểm thử lại toàn bộ hệ thống nhằm đảm bảo chất lượng cho từng phiên bản phần mềm điều này là rất khó khăn do tiêu tốn nhiều chi phí và thời gian. Hệ quả là không thể kiểm soát sự thay đổi mã nguồn, mã nguồn không đảm bảo

tính nhất quán so với đặc tả và thiết kế ban đầu. Thêm vào đó, do sự phức tạp của mã nguồn cộng với quá trình bảo trì nâng cấp thường xuyên để có thể nắm bắt được toàn bộ cấu trúc mã nguồn là rất khó khăn, đặc biệt là với những nhà phát triển mới tham gia vào dự án. Với sự ra đời của công cụ mà khóa luận này xây dựng, sẽ đem lại sự trợ giúp đắc lực trong các dự án phần mềm. Bao gồm, kiểm tra sự nhất quán giữa mã nguồn với đặc tả thiết kế ban đầu về mẫu thiết kế qua từng phiên bản của phần mềm, giúp nhà phát triển nắm bắt được toàn bộ cấu trúc mã nguồn.

Qua thực nghiệm, ta thấy công cụ hoạt động tốt với những mẫu thiết kế do doanh nghiệp tự định nghĩa, một số những mẫu thiết kế cơ bản thuộc *GoF (Gang of Fours) design patters* [1]. Bên cạnh đó, do thời gian là có hạn, tập phụ thuộc cần được đúc kết và nghiên cứu trong thời gian dài do vậy sự chưa đầy đủ về mặt xác định phụ thuộc giữa các thành phần mã nguồn là không thể tránh khỏi, điều này dẫn tới trong thực nghiệm vẫn còn một số hạn chế về sự trùng lặp giữa các mẫu thiết kế, một số mẫu thiết kế chưa thể được phát hiện.

Những hạn chế này sẽ được khắc phục trong những giai đoạn tiếp theo. Thực nghiệm cũng sẽ được thực hiện lại nhiều lần. Mục tiêu nhằm đem lại độ chính xác tốt hơn cho công cụ và trong tương lai công cụ có thể được đưa vào áp dụng trong thực tế tại các doanh nghiệp. Hiện tại công cụ đang nhận được một số phản hồi tích cực từ phía các chuyên gia tại Trung tâm Công nghệ và Quản lý chất lượng phần mềm Viettel.

Chương 5

Kết luận

Khóa luận này đã đề xuất phương pháp và xây dựng công cụ kiểm tra sự tuân thủ mẫu thiết kế của các dự án sử dụng Java. Ở giai đoạn này, khóa luận đã hoàn thành cơ bản các chức năng chính của công cụ. Phương pháp sẽ xây dựng một luồng xử lý mã nguồn Java, mã nguồn sẽ được phân tích thành cây cấu trúc, từ cây cấu trúc sẽ được tiếp tục phân tích để tìm những phụ thuộc tồn tại giữa các thành phần mã nguồn, sau đó tiền hành xây dựng đồ thị phụ thuộc từ cây cấu trúc. Mỗi khi người dùng muốn kiểm tra sự tuân thủ của mẫu thiết kế theo đặc tả của dự án, chỉ cần thực hiện tải lên cơ sở dữ liệu mẫu thiết kế của JCIA-VT những mẫu thiết kế mà mã nguồn sẽ áp dụng theo như đặc tả ban đầu, sau đó tiến hành hành tải lên mã nguồn dự án để phát hiện sự tuân thủ mẫu thiết kế của mã nguồn dự án đối với đặc tả ban đầu.

Bên cạnh đó, khóa luận cũng tiến hành tích hợp công cụ vào bộ công cụ quản lý và đảm bảo chất lượng phần mềm JCIA-VT được triển khai trên nền tảng ứng dụng Web. Điều này đem lại nhiều thuận lợi về phía người dùng. Cụ thể, người dùng đã có thể tiến hành phân tích dự án, phát hiện sự tuân thủ mẫu thiết kế bên trong mã nguồn, không chỉ là những mẫu thiết kế phổ biến đã được định nghĩa sẵn mà còn hỗ trợ những mẫu thiết kế đặc thù ở từng doanh nghiệp khác nhau. Cùng với các tiện ích quản lý phiên bản đã có của bộ JCIA-VT. Đem lại một tập các tiện ích phục vụ cho hoạt động quản lý phiên bản mã nguồn phần mềm.

Dựa trên những gì đã thực hiện được của khóa luận này. Mặc dù công cụ còn tồn tại nhiều hạn chế và tính ứng dụng thực tế chưa cao. Tuy nhiên đã đem lại một nền tảng tốt, có thể tiếp tục hoàn thiện. Do đó, ở giai đoạn tiếp theo, phụ thuộc giữa các thành phần mã nguồn sẽ được tiếp tục nghiên cứu và thêm vào bộ phân tích của công cụ, nhằm cải thiện độ tin cậy của công cụ cũng như tăng khả năng áp dụng vào thực tế. Ngoài ra, tiếp tục phát triển thêm tính năng phát hiện thành phần mã nguồn áp dụng sai mẫu thiết kế nhằm tăng tính toàn diện của công cụ. Cụ thể, công cụ sẽ chỉ ra những thành phần mã nguồn áp dụng sai mẫu thiết kế, đồng thời đề xuất phương án sửa chữa nhằm khắc phục sai sót về cấu trúc mẫu thiết kế của mã nguồn. Nhằm xây dựng một bộ công cụ toàn diện phục vụ cho hoạt động quản lý và đảm bảo chất lượng phần mềm.

Tài liệu tham khảo

[1] Design Patterns: Elements of Reusable Object-Oriented Software by ErichGamma, RichardHelm, RalphJohnson, and JohnVlissides (the GangOfFour)

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

[3] Y. G. Gueheneuc, P-MARt : Pattern-like Micro Architecture Repository, Proceedings of the 1 st EuroPLoP Focus Group on Pattern Repositories (EPFPR), 2007.

[4] Dong, J., Sun, Y., Zhao, Y., Design Pattern Detection by Template Matching, Proceedings of the 2008 ACM Symposium on Applied Computing, Fortaleza, Brazil, 2008, pp. 765-769.

[5] A Toolchain for Source Code Quality Assurance of Java EE Applications Bui, Quang Cuong and Dinh, Tien Loc and Luu, Anh Viet and Nguyen, Viet Hoa and Pham, Ngoc Quy and Pham, Ngoc Hung (2018) A Toolchain for Source Code Quality Assurance of Java EE Applications. Technical Report. Vietnam National University Hanoi

[6] 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.

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

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

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