به نام خدا

افزایش تست‌پذیری برنامه با اعمال ریفکتورینگ خودکار روی کد برنامه

فهرست مطالب:

# مقدمه

# شرح پروژه

## تولید کلاس دیاگرام

## محاسبه پیچیدگی و تست‌پذیری کد

## بازسازی الگو کارخانه(factory pattern)

## بازسازی الگو تزریق(injection pattern)

# شرح آزمایش‌ها

# نتیجه گیری

# کارهای آینده

# 1. مقدمه:

هدف از این پروژه ارائه روشی برای تایین میزان آزمون پذیری برنامه‌ها می‌باشد. برای این منظور باید بتوان با تحلیل خود برنامه میزان وابستگی بین کلاس ها را مشخص کرد. مسلما با افزایش میزان وابستگی آزمون واحد (unit test) کلاس ها پیچیده تر می‌گردد. در واقع این وابستگی موجب می‌شود که ردیابی خطا در داخل برنامه پیچیده شود چرا که با مشاهده خطا در یک متد در داخل یک کلاس به سادگی مشخص نمی‌شود که آیا علت خطا در آن متد است یا در متدهایی که به آن ها وابستگی وجود دارد. می‌توان با استفاده از الگوهایی مانند الگوی تزریق(injection pattern) و الگو کارخانه (factory pattern) و به کار‌گیری راهکارهای بازسازی(refactoring code) این گونه اشکالات یا در اصطلاح بوی بد کد(code smell) را از میان برداشت. در این پروژه با استفاده از یک مولد کامپایلر و برای برنامه‌های جاوا کد پایتون برای مراحل زیر ایجاد می‌گردد :

1. ایجاد ابزاری برای استخراج مدل ارتباطی کلاس‌ها از متن پروژه های جاوا
2. ایجاد ابزاری برای تحلیل مدل وابستگی کلاس‌ها و محاسبه میزان آزمون پذیری کد بر اساس میزان وابستگی کلاس‌ها به یک دیگر
3. ایجاد ابزاری برای اعمال الگوی تزریق بر کلاس‌های جاوا و در واقع تولید کلاس رابط(interface) برای برقراری ارتباط با هر کلاس
4. پیاده سازی الگوی کارخانه با بررسی چگونگی ارتباط بین کلاس‌ها

# 2. شرح پروژه:

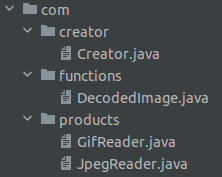
2.1. تولید کلاس دیاگرام:

در این قسمت مدل(گراف) ارتباطی کلاس‌ها از متن پروژه جاوا استخراج می‌شود.

برای درک بهتر نحوه انجام مراحل این قسمت به مثال زیر توجه کنید:

این پروژه جاوا برای مدیریت فایل‌های تصویری و خواندن آن‌ها طراحی شده است.

این مثال یک پروژه جاوا ساده با 4 کلاس جاوا است که ساختار فایل‌های در شکل زیر نشان داده شده است:



کلاس Creator.java:

در این کلاس ابتدا فرمت فایل ورودی تشخیص داده شده است و متناسب با فرمت آن فایل تصویری در یک ساختار if statement شی مناسب برای خواندن آن فایل(reader) را می‌سازد و در آخر با استفاده از reader فایل ورودی را decode می‌کند و مقدار decode شده آن را چاپ می‌کند.

**package com.creator**;  
**import com.functions.DecodedImage**;  
**import com.products.\***;  
  
**public class** Creator {  
 **public static void** main(**String**[] args) {  
 DecodedImage decodedImage **= new** DecodedImage();  
 String image **=** args[***0***];  
 String format **=** image.substring(image.indexOf(**'.'**) **+ *1***, (image.length()));  
 **if** (format.equals(**"gif"**)) {  
 GifReader reader **= new** GifReader(image);  
 }  
 **if** (format.equals(**"jpeg"**)) {  
 JpegReader reader **= new** JpegReader(image);  
 }  
 **assert** reader **!= *null***;  
 decodedImage **=** reader.getDecodeImage();  
 System.out.println(decodedImage.toString());  
 }  
}

کلاس DecodedImage.java:

در این کلاس مقدار یک عکس را می‌گیرد و آن را ابتدا decode کرده و سپس به string تبدیل می‌کند.

**package com.functions**;  
  
**class** DecodedImage {  
 **private** String image;  
  
 **public** DecodedImage(String image) {  
 this.image **=** image;  
 }  
  
 **public** String toString() {  
 **return** image **+ ": is decoded"**;  
 }  
}

کلاس GifReader.java:

در این کلاس یک عکس با فرمت git را ورودی می‌گیرد و شی decodedImage آن را تولید کرده و با استفاده از متد getDecodedImage شی decodedImage آن را برمی‌گرداند.

**package com.products**;  
**import com.functions.DecodedImage**;  
  
**class** GifReader {  
 **private** DecodedImage decodedImage;  
  
 **public** GifReader(String image, **int** size) {  
 this.decodedImage **= new** DecodedImage(image);  
 }  
  
 **public** DecodedImage getDecodeImage() {  
 **return** decodedImage;  
 }  
}

کلاس JpegReader.java:

در این کلاس یک عکس با فرمت jpeg را ورودی می‌گیرد و شی decodedImage آن را تولید کرده و با استفاده از متد getDecodedImage شی decodedImage آن را برمی‌گرداند.

**package com.products**;  
**import com.functions.DecodedImage**;  
  
**class** JpegReader{  
 **private** DecodedImage decodedImage;  
  
 **public** JpegReader(String image, **int** size) {  
 decodedImage **= new** DecodedImage(image);  
 }  
  
 **public** DecodedImage getDecodeImage(**int** x, **int** y) {  
 **return** decodedImage;  
 }  
}

نحوه پیاده‌سازی class diagram:

**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 java\_project\_address = config.projects\_info[**'10\_water-simulator'**][**'path'**]  
 base\_dirs = config.projects\_info[**'10\_water-simulator'**][**'base\_dirs'**]  
 files = File.find\_all\_file(java\_project\_address, **'java'**)  
 index\_dic = File.indexing\_files\_directory(files, **'class\_index.json'**, base\_dirs)  
 cd = ClassDiagram(java\_project\_address, base\_dirs, index\_dic)  
 cd.make\_class\_diagram()  
  
 cd.show(cd.class\_diagram\_graph)  
 *#cd.load('class\_diagram.gml')*  
cd.save(**'class\_diagram.gml'**)  
  
 cd.set\_stereotypes()  
 cd.save(**'class\_diagram.gml'**)  
 cd.show(cd.class\_diagram\_graph)

توضیح قسمت‌های مختلف:

* java\_project\_address : در واقع مسیر اصلی پروژه جاوا است.

مقدار این متغیر برای پروژه مثال زده شده:

benchmarks/javaproject

* base\_dirs : هر پروژه ممکن است چندین root داشته باشد و به مسیر هر یک از آن rootها base\_dir گفته می‌شود. در واقع گراف کلاس دیاگرام هر base\_dir به صورت کاملا جدا از بقیه base\_dirها است ولی اکثر پروژه‌های یک base\_dir بیشتر ندارند.

مقدار این متغیر برای پروژه مثال زده شده:

['benchmarks/javaproject/']

* Files : این متغیر شامل لیستی از فایل‌ها با فرمت .java در پروژه است.

مقدار این متغیر برای پروژه مثال زده شده:

[  
 **'benchmarks/javaproject/com/functions/DecodedImage.java'**,  
 **'benchmarks/javaproject/com/products/JpegReader.java'**,  
 **'benchmarks/javaproject/com/products/GifReader.java'**,  
 **'benchmarks/javaproject/com/creator/Creator.java'**  
]

# شرح تئوری کار:

الگوی کارخانه(factory pattern):

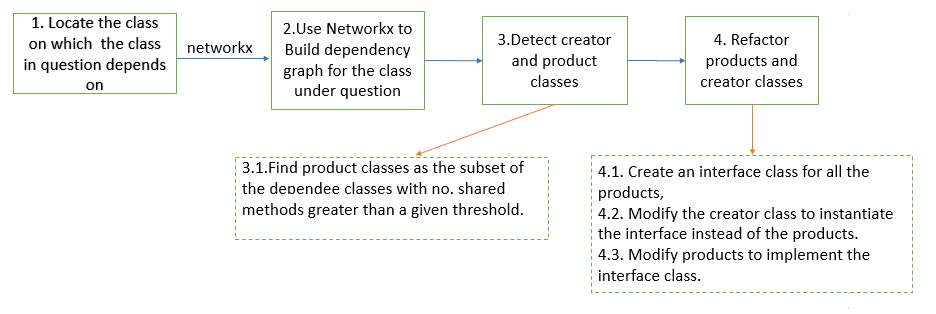
* روش کارخانه یک الگوی طراحی سازنده است که یک interface برای کلاس‌های محصول(products) ایجاد می‌کند و در کلاس سازنده(creator) اشیا محصول از نوع interface ساخته می‌شوند.
* برای فهم بیشتر این الگو به مثال زیر توجه کنید:
  + فرض کنید شما یک برنامه برای خواندن فایل‌های مختلف تصویری نوشته‌اید که در این برنامه فایل‌های تصویری مختلف مثل gif و jpeg خوانده شده و decode می‌شوند.
  + بعد از مدتی برنامه شما معروف می‌شود و نیاز است که انواع مختلف تصویر و فیلم مانند png را نیز بتواند بخواند و decode کند.
  + در این حالت شاید نیاز باشد مقدار زیادی از کد را تغییر دهید تا بتوان این نوع فایل تصویری را نیز پوشش دهد.
  + پس همیشه نیاز دارید کد برنامه خود را برای اضافه کردن یک نوع جدید تصویر تغییر دهید.
* راهکار:
  + الگو کارخانه به شما پیشنهاد می‌کند که به جای فراخانی مستقیم شی(object) مورد نظر از یک کارخانه برای این کار استفاده کنید و این شی کارخانه مسئول تشخیص نوع تصویر مورد نظر است و شی مورد نظر را تولید می‌کند.

مزیت‌ها و عیب‌های این الگو:

* شما از اتصال محکم بین سازنده و محصولات بتنی اجتناب می‌کنید.
* اصل مسئولیت واحد(Single Responsibility Principle): می‌توانید کد ایجاد محصول را به یک مکان در برنامه منتقل کنید تا پشتیبانی از کد آسان‌تر شود.
* اصل باز/بسته(Open/Close Principle): شما می‌توانید انواع جدیدی از محصولات را بدون شکستن کد مشتری موجود به برنامه معرفی کنید.
* ممکن است کد پیچیده‌تر شود زیرا برای پیاده‌سازی الگو باید زیر کلاس‌های جدید زیادی معرفی کنید.

برای اعمال الگو کارخانه(factory pattern) روی پروژه مدیریت فایل‌های تصویری باید مراحل زیر انجام شوند:

شرح کد پروژه:



شکل 1: نمای کلی مراحل انجام شده برای الگوی کارخانه

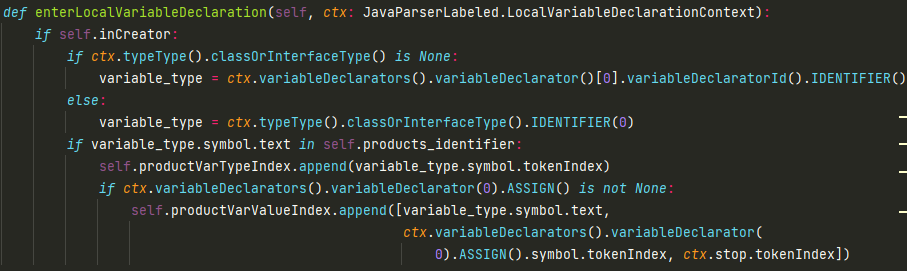
برای خودکار انجام دادن این بازنمایی از 3 بار پیمایش DFS درخت تجزیه استفاده شده است. برای بدست آوردن درخت تجزیه و پیمایش آن به صورت DFS از کتابخانه ANTLR استفاده شده است.

کلاس FixCreatorListener:

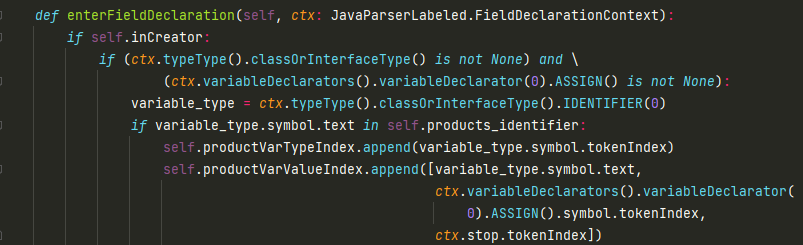
در این کلاس Listener مربوط به بازسازی کلاس creator نوشته شده است که در زیر به موارد مهم آن می‌پردازیم:

. . .

* در کد زیر تمام متغیرهای محلی که نوع آن‌ها برابر یکی از productها باشد را در productVarValueIndex ذخیره می‌کند.



* کد زیر نیز مانند قسمت بالا عمل می‌کند فقط تنها فرق آن این است که متغیرهای کلاس را بررسی می‌کند.



* در کد زیر در کلاس سازنده(creator) از نوع interface ساخته شده به جای نوع(type) کلاس‌های محصول استفاده می‌شود.



* برای فهم بهتر نقش این کلاس در این برنامه به مثال زیر توجه کنید:

+import simulator.Interface34;

public class SuiteGUI extends JFrame {

private Function[] availableFunctions = new Function[7];

- private Linear linear = new Linear();

+ private Interface34 linear = new Linear();

- private Random random1 = new Random();

+ private Interface34 random1 = new Random();

- private Random random2 = new Random();

+ private Interface34 random2 = new Random();

- private Discrete discrete = new Discrete();

+ private Interface34 discrete = new Discrete();

- private Raise raise = new Raise();

+ private Interface34 raise = new Raise();

- private MetDataFunction metData = new MetDataFunction();

+ private Interface34 metData = new MetDataFunction();

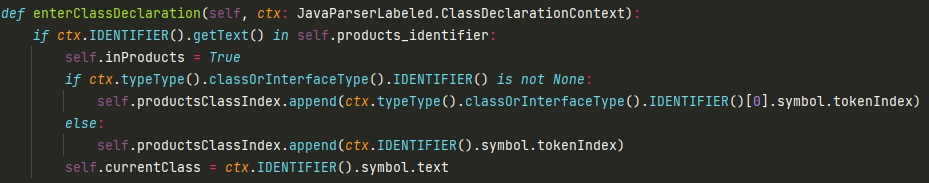
- private Raise price = new Raise();

+ private Interface34 price = new Raise();

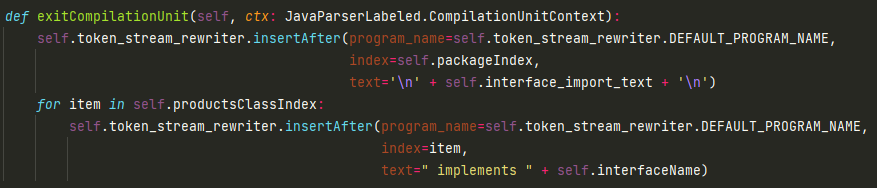
* در مثال بالا Linear, Random, Discrete, Raise, MetDataFunction کلاس‌های محصول(product) هستند که همگی به Interface34 تبدیل شده‌اند.
* در مثال بالا Interface34 در ابتدا کد هم به صورت خودکار Import شده است.

کلاس FixProductListener:

* در کد زیر سعی می‌شود مکانی که قرار است عبارت “implement interface\_name” اضافه شود یافته را به لیست productsClassIndex اضافه کند.



* در این قسمت عبارت import برای interface ایجاد شده به ابتدا فایل اضافه می‌شود و همچنین عبارت “implement interface\_name” نیز به کلاس محصول(product) اضافه می‌شود.



* برای فهم بهتر نقش این کلاس به مثال زیر توجه کنید:

package simulator.SA.gui;

+import simulator.SA.Interface26;

+

import jade.gui.GuiEvent;

import java.awt.BorderLayout;

import java.awt.Dimension;

-public class DemandCurveParameterFrame extends JInternalFrame {

+public class DemandCurveParameterFrame extends JInternalFrame implements Interface26 {

private static final long serialVersionUID = 1208279781892420981L;

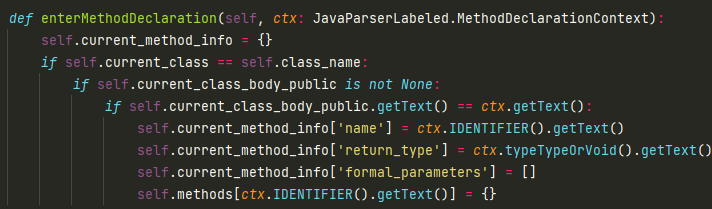
private Logger log = Logger.getLogger(DemandCurveParameterFrame.class);

private boolean flagFunctionSelected = false;

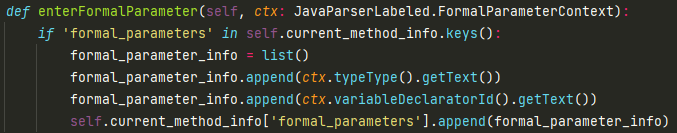
* در مثال بالا ابتدا عبارت import interface ساخته شده(Interface26) اضافه می‌شود و سپس عبارت implements Interface26 نیز به کلاس DemandCurveParameterFrame اضافه می‌شود.

کلاس ProductCreatorDetectorListener:

* در این قسمت نام و نوع خروجی بدست می‌آید و لیست پارامترهای ورودی با یک لیست خالی مقدار دهی اولیه می‌شود.



* در این قسمت پارامترهای ورودی method مورد نظر پیدا شده و به لیست متغیرهای ورودی method اضافه می‌شود.



تعریف حساسیت(sensitivity):

* میزان حساسیت الگوریتم تشخیص الگوی کارخانه(factory pattern) گفته می‌شود که طبق فرمول زیر بدست می‌آید:

No\_common\_methods: تعداد methodهایی که بین کلاس‌هایی که کلاس سازنده(creator) به آن‌ها وابسته است.

No\_methods\_class1: تعداد methodهای کلاس class1

Sensitivity = no\_common\_methods / max(no\_methods\_class1, no\_methods\_class2, . . .)

کلاس Factory:

* در این method کلاس‌های محصول و methodهایی از آن‌ها که مشترک هستند بر اساس پارامتر sensitivity مشخص می‌شوند.



سودو کدهای الگوی کارخانه:

**factory.py\Factory:**

def refactor(sensitivity, class\_diagram):

internal\_nodes = get\_internal\_nodes(class\_diagram)

for node in internal\_nodes:

neighbors = node.neighbors

if len(neighbors) > 1:

neighbors\_methods\_dict = {}

for child\_node in neighbors:

listener = ProductCreatorDetectorListener(child\_node.class\_name)

listener.walk()

neighbors\_methods\_dict[child\_node] = listener.methods

result = find\_products(node, neighbors\_methods\_dict, sensitivity)

if len(result.products) > 1:

interface = create\_interface(result.products\_common\_methods)

fix\_creator(node, interface, products)

for product in products:

fix\_product(product, interface)

def find\_products(node, neighbors\_methods\_dict, sensitivity):

factory\_info = {}

factory\_info.creator = node

candidate\_product\_classes = neighbor\_methods\_dict.get\_classes()

for class1 in candidate\_product\_classes:

products = []

method\_list = class1.methods

no\_class1\_methods = len(class1.methods)

for class2 in candidate\_product\_classes:

no\_class2\_methods = len(class2.methods)

common\_methods = get\_similarity\_of\_two\_list(method\_list, class2.methods)

if len(common\_methods) / max(no\_class1\_methods, no\_class2\_methods) >= sensitivity:

method\_list = common\_methods.copy()

class\_list.append(class2)

factory\_info.products = products

factory\_info.products\_common\_methods = method\_list

return factory\_info

def fix\_product(product, interface):

product.add\_import\_statement(interface.package, interface.name)

product.add\_implement\_statement(product.name, interface.name)

def fix\_creator(creator, interface, products):

creator.add\_import\_statement(interface.package)

for product in products:

creator.replace\_type(product, interface.name)

def add\_import\_statement(package, name):

package = enterPackageDeclaration()

current\_token = package.stop.token

import = enterImportDeclaration()

while import is not None:

current\_token = import.stop.token

import = enterImportDeclartion()

import\_text = ‘import ’ + package + ‘.’ + name + ‘;’

token\_stream\_rewirter.insertAfter(current\_token, import\_text)

def add\_implement\_statement(class\_name, interface\_name):

if enterClassDeclaration().name == class\_name:

class = enterClassDeclaration()

while (enterClassDeclaration() is not None):

if enterClassDeclaration().name == class\_name:

class = enterClassDeclaration()

break

implement\_statement\_token = None

implement\_statement\_text = ‘implement ’ + interface + ‘;’

if class.extend is not None:

implement\_statement\_token = class.extend.token

if class.implement is not None:

implement\_statement\_token = class.implement.token

implement\_statement\_text = ‘ ’ + interface + ‘;’

token\_stream\_rewirter.insertAfter(implement\_statement\_token, implement\_statement\_text)

def replace\_type(product, interface\_name):

dependee = enterClassOrInterfaceType()

while dependee is not None:

if dependee == product.type:

token\_stream\_rewirter.replace(dependee.start.token,

dependee.start.token + 1,

interface\_name)