به نام خدا

افزایش تست‌پذیری برنامه با اعمال ریفکتورینگ خودکار روی کد پروژه

فهرست مطالب:

# مقدمه

# شرح تئوری کار

# شرح کد پروژه

## تولید کلاس دیگرام

## محاسبه پیچیدگی و تست‌پذیری کد

## بازسازی الگو کارخانه(factory pattern)

## بازسازی الگو تزریق(injection pattern)

# شرح آزمایش‌ها

# نتیجه گیری

# کارهای آینده

# مقدمه:

هدف از این پروژه ارائه روشی برای تایین میزان آزمون پذیری برنامه‌ها می‌باشد. برای این منظور می 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 می گردد. در واقع این وابستگی موجب می شود که ردیابی خطا در داخل برنامه پیچیده شود چرا که با مشاهده خطا در یک متد در داخل یک کلاس به سادگی مشخص نمی شود که ایا علت خطا در آن متد است یا در متدهایی که به آن ها وابستگی وجود دارد. می توان با استفاده از الگوهایی مانند الگوی تزریق(injection pattern) و الگو کارخانه (factory pattern) و به کار گیری راهکار های 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 برنامه های جاوا کد پایتون برای مراحل زیر ایجاد می گردد :

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

با استفاده از ابزارهای فوق سعی می کنیم که فرایند TDD را در قالب فرایند جدید پیشنهادی خود به نام TsDD (testability driven development ) بهبود بخشید.

# شرح تئوری کار:

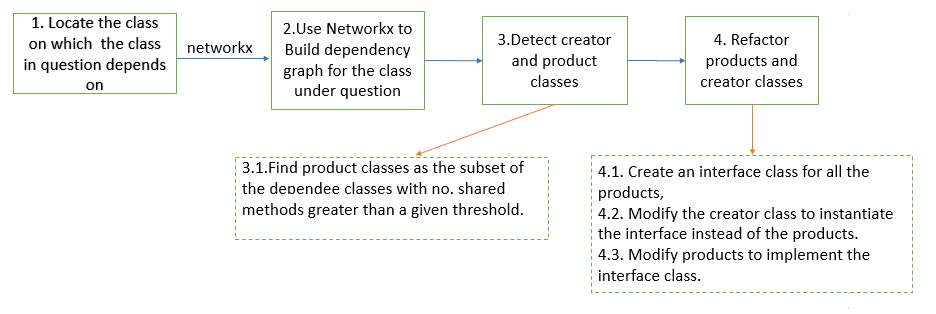
الگوی کارخانه(factory pattern):

* 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 مدیریت حمل و نقل نوشته‌اید. فرض کنید در ابتدا فقط برنامه ما حمل و نقل کامیون را مدیریت می‌کند, بنابراین ما باید در برنامه خود یک کلاس کامیون داشته باشیم.
  + بعد از مدتی برنامه شما معروف می‌شود و روزانه تعداد زیادی درخواست از شرکت‌های دریایی برای اضافه کردن حمل و نقل دریایی به برنامه‌تان دریافت می‌کنید.
  + حال با توجه به این که تمام کد شما بر اساس کامیون نوشته شده است شاید مجبور باشید مقدار زیادی از کد را تغییر بدهید تا بتوانید قابلیت حمل و نقل قایق را نیز اضافه کنید و در آینده اگر نوع دیگری حمل و نقل اضافه شد شاید مجبور باشید دوباره این تغییرات را تکرار کنید.
  + پس شما همیشه با این روند نیاز دارید کد را تغییر دهید با گزاره‌های شرطی متوالی نوع حمل و نقل مورد نظر را تشخیص 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 حمل و نقل 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 می کنید.
* 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. بهترین حالت زمانی است که شما الگو را در یک سلسله مراتب موجود از کلاس‌های سازنده معرفی می‌کنید.

شرح کد پروژه:



شکل 1: نمای کلی مراحل انجام شده برای الگوی کارخانه

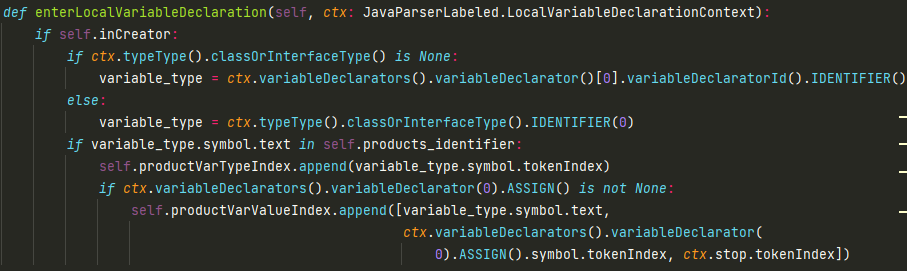
برای خودکار انجام دادن این بازنمایی از 3 بار پیمایش DFS درخت تجزیه استفاده شده است. برای بدست آوردن درخت تجزیه و پیمایش آن به صورت DFS از کتابخانه ANTLR استفاده شده است.

کلاس FixCreatorListener:

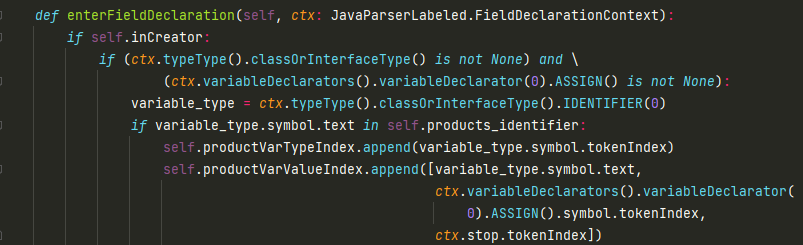
در این کلاس Listener مربوط به بازسازی کلاس creator نوشته شده است که در زیر به موارد مهم آن می‌پردازیم:

. . .

* 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();

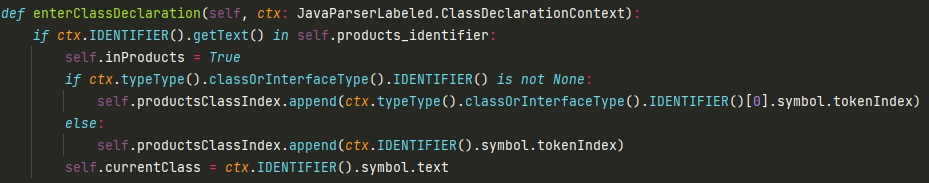
- private Raise price = new Raise();

+ private Interface34 price = new Raise();

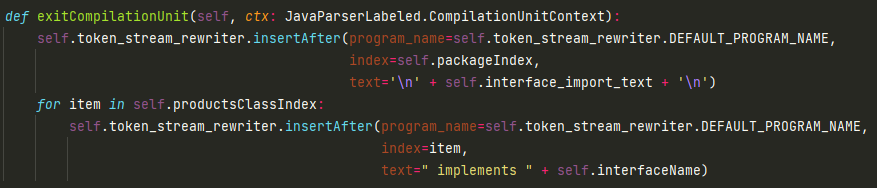
* در مثال بالا Linear, Random, Discrete, Raise, MetDataFunction کلاس‌های محصول(product) هستند که همگی به Interface34 تبدیل شده‌اند.
* در مثال بالا Interface34 در ابتدا کد هم به صورت خودکار Import شده است.

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;

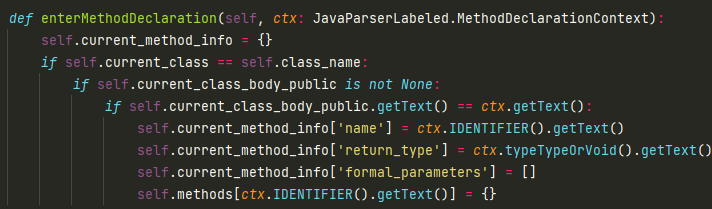
private Logger log = Logger.getLogger(DemandCurveParameterFrame.class);

private boolean flagFunctionSelected = false;

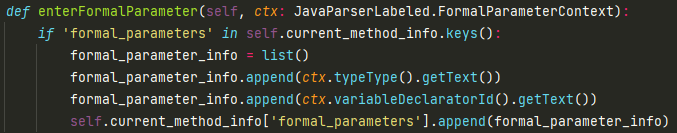
* در مثال بالا ابتدا عبارت import interface ساخته شده(Interface26) اضافه می‌شود و سپس عبارت implements Interface26 نیز به کلاس DemandCurveParameterFrame اضافه می‌شود.

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)