In the name of God

Auto-refactoring Factory Pattern

Sadegh Jafari

Introduction:

In this project we used compiler and Antlr4 to detect factory-pattern and refactor the java project to have better code.

Refactor manually is very expensive especially when we want to refactor very big project, so we need better and cheap way.

Theories of the work:

**Factory Method** is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Example:

* Imagine that you’re creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the Truck class.
* After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app.
* At present, most of your code is coupled to the Truck class. Adding Ships into the app would require making changes to the entire codebase. Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all of these changes again.
* As a result, you will end up with pretty nasty code, riddled with conditionals that switch the app’s behavior depending on the class of transportation objects.

Solution:

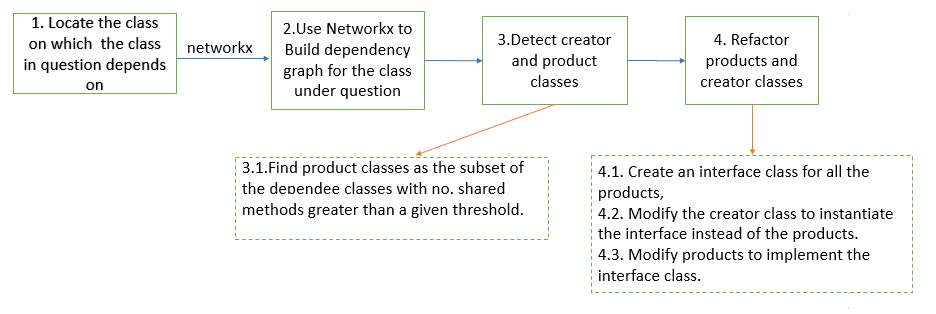
* + The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special factory method. Don’t worry: the objects are still created via the new operator, but it’s being called from within the factory method. Objects returned by a factory method are often referred to as products.

**Pros and Cons**

* You avoid tight coupling between the creator and the concrete products.
* Single Responsibility Principle. You can move the product creation code into one place in the program, making the code easier to support.
* Open/Closed Principle. You can introduce new types of products into the program without breaking existing client code.
* The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when you’re introducing the pattern into an existing hierarchy of creator classes.

Implementation:

* First we find the dependency graph and then use this graph to find creator and products classes.
* Creator is a class that instantiates product classes.
* Products are classes having some common methods with different implementation.
* After finding creator and products classes we start with refactoring.



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

Pseudocode:

**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)