**ISYS1083/1084 Object Oriented Software Design**

**Tutelab 3 Refactoring**

1. Briefly explain why the following practices are considered bad?

1. Commented out code
2. Large classes
3. Feature Envy
4. Duplicated switch statements
5. Misplaced Responsibilities
6. Inappropriate Intimacy between classes
7. Long parameter list
8. Don’t know the intention of the author :

If it’s a temporary during debugging and forgotten.  code needs to be fixed.

If it’s a function removed from the requirements  should delete the code.

If its experimental code and an alternative implementation was chosen  document the implementation choice and reasons why in--line, maybe package the alternative code elsewhere and delete the code.

It may use functions and variable no longer in use.

It can be retrieved from previous version if necessary

1. Probably has low cohesion and therefore the highly coupled sub functions will cause unnecessary cascading of changes between functions.
2. High coupling : cascading changes between these classes : the envier and the envyee.
3. Every time there is a change to the discrimination values there are multiple places in the code that need to be updated. The more duplication the greater the chance one will be overlooked and a bug introduced.

Solve it with polymorphism.

2. High coupling. A change to one class’s internal implementation will affect the other so there is no encapsulation.
3. This may be a sign that the design is not very OO. Can some of the parameters could be wrapped up into an class?... reducing both complexity and parameter counts.

It is hard for developers to comprehend a long parameter list.

2. How can *Extract Method* be applied to avoid code replication in the code below?

public class CustomerNameChanger

{

public void changeName(CustomerContext, context, String customerID, String name)

{

Customer customer = context.getCustomer(customerID);

If (customer == null)

Throw new Exception (“Customer with ID “ + customerID + “ is not found”);

customer.name = name;

}

}

public class CustomerAddressChanger

{

public void changeAddress(CustomerContext, context, String customerID, String address)

{

Customer customer = context.getCustomer(customerID);

If (customer == null)

Throw new Exception (“Customer with ID “ + customerID + “ is not found”);

customer.address = address;

}

}

**1) Extract into static method :**

**Customer customer = context.getCustomer(customerID); if (customer == null)**

**throw new Exception("Customer with ID " + customerID + " is not found");”**

**2) Extract method into new class “CustomerHelper”**

**3) Copy one line “Customer customer = context.getCustomer(customerID);” into 2nd Changer, overwriting same three lines.**

**4) Helper is not the best place for this code. The Helper actually has code envy. Lets place this method in CustomerContext and remove the context parameter by renaming it to customers.**

**5) Then rename all “customers” to “customersById”.**

**6) Rename “customer” to “result”. (convention inspired from the Eiffel programming language)**

**7) Refactor Exception to be more specific. In CustomerContext class only make it an IndexOutOfBoundsException.**

**8) Extract interface from Customer class for the two pairs of getters and setters. Keep the interface named “Customer” and rename the implementation “CustomerModel”**

**3.** The Circle and Square classes contain common methods to draw and move all such objects. Draw a class diagram showing appropriate classes and interfaces by applying the Extract Interface refactoring that will allow all such objects to be moved or displayed polymorphically.

class Circle {

public Circle(double x, double y, double radius) {

…

}

public void draw(Graphics g) {

}

public void move(int x, int y) {

}

}

class Square {

public Square(double x, double y, double side) {

…

}

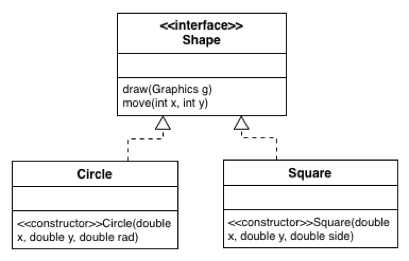
public void draw(Graphics g) {

}

public void move(int x, int y) {

}

}



4. Rewrite the Stack class below using delegation

class MyStack extends Vector {

public void push(Object element) {

insertElementAt(element,0);

}

public Object pop() {

Object result = firstElement();

removeElementAt(0);

return result;

}

}

import java.util.Vector;

/\*\*

\* Created by keith on 8/03/2014.

\*/

class MyStack {

private Vector<Object> vector;

// Construction

MyStack() {

this.vector = new Vector<Object>();

}

// Public API

public void push(Object element) {

vector.insertElementAt(element, 0);

}

public Object pop() {

Object result = vector.firstElement();

vector.removeElementAt(0);

return result;

}

}

**5.** Locate your own Java code from a previous semester (e.g. assignment or exercise) that could benefit from refactoring.

1. Specify sections of the code that could be refactored citing the reasons.
2. Identify at least one specific (Fowler) refactoring technique that can be applied citing the reason.
3. Refactor and rewrite the code according to your answer to b)
4. Explain how the refactored code improves
5. Understanding
6. Maintainability

No answer – Ask you tutor to check it in class.