**Лабораторна робота № 1-2**

**Main**

import java.util.ArrayList;  
import java.util.List;  
import java.util.Scanner;  
  
public class Main {  
  
 public static void main(String[] args) throws Exception {  
  
 Scanner scanner = new Scanner(System.*in*);  
  
 Release release = new Release();  
 release.AddCommand(new Controller());  
 release.RunCommand();  
  
 new ProxxyPattern().execute(  
 scanner.nextDouble(), scanner.nextDouble(), scanner.nextDouble(), scanner.nextDouble()  
 );  
 }  
  
 static class Release {  
  
 List <Command> list = new ArrayList<>();  
 Scanner scanner = new Scanner(System.*in*);  
  
 void AddCommand(Command newCommand) {  
 list.add(newCommand);  
 }  
  
 void RunCommand() {  
 System.*out*.println("Enter data (1, 1, 3, 4): ");  
 list.forEach(command -> {  
 try {  
 command.execute(  
 scanner.nextDouble(),scanner.nextDouble(),scanner.nextDouble(),scanner.nextDouble()  
 );  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 });  
 }  
 }  
}

**Point\_Class**

import java.util.Objects;  
  
public class Point\_Class implements Virtual {  
 public double x;  
 public double y;  
  
 public Point\_Class(double x, double y) {  
 this.x = x;  
 this.y = y;  
 }  
  
 public Point\_Class() {  
 x = 0;  
 y = 0;  
 }  
  
 @AnnotationReflectable(name="Getting x")  
 public double getX() {  
 return x;  
 }  
  
 @AnnotationReflectable(name="Getting y")  
 public double getY() {  
 return y;  
 }  
  
 public void setX(double x) {  
 this.x = x;  
 }  
  
 public void setY(double y) {  
 this.y = y;  
 }  
  
 @Override  
 public String toString() {  
 return "point {x = " + x + "; y = " + y + '}';  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Point\_Class point = (Point\_Class) o;  
 return Double.*compare*(point.x, x) == 0 && Double.*compare*(point.y, y) == 0;  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(x, y);  
 }  
  
 @Override  
 public void moving(double x, double y) {  
 this.x += x;  
 this.y += y;  
 }  
}

**Rectangular**

import java.util.Objects;  
  
public class Rectangular extends Point\_Class implements Virtual {  
  
 private double height;  
 private double width;  
  
 public Rectangular(double x, double y, double height, double width) {  
 super(x, y);  
 this.height = height;  
 this.width = width;  
 }  
  
 public Rectangular(double height, double width) {  
 super();  
 this.height = height;  
 this.width = width;  
 }  
  
 public Rectangular() {  
 super();  
 this.height = 0;  
 this.width = 0;  
 }  
  
 public Rectangular(Rectangular r) {  
 super(r.getX(), r.getY());  
 this.width = r.getWidth();  
 this.height = r.getHeight();  
 }  
  
 @AnnotationReflectable(name="Getting Height")  
 public double getHeight() {  
 return height;  
 }  
  
 public void setHeight(double height) {  
 this.height = height;  
 }  
  
 @AnnotationReflectable(name="Getting Width")  
 public double getWidth() {  
 return width;  
 }  
  
 public void setWidth(double width) {  
 this.width = width;  
 }  
  
 @AnnotationReflectable(name="Getting Area")  
 public double area() {  
 return width \* height;  
 }  
  
 @AnnotationReflectable(name="Getting Perimeter")  
 public double perimeter() {  
 return 2 \* (width + height);  
 }  
  
 public boolean isEmpty() {  
 return (width <= 0.0) || (height <= 0.0);  
 }  
  
 public boolean contains(double x, double y) {  
 double x0 = getX();  
 double y0 = getY();  
 return (x >= x0 && y >= y0 && x < x0 + getWidth() && y < y0 + getHeight());  
 }  
  
 @Override  
 public void moving(double x, double y) {  
 super.moving(x, y);  
 }  
  
 @Override  
 public String toString() {  
 return "Rectangular {height = " + height + ", width = " + width + "}";  
 }  
  
 @Override  
 public boolean equals(Object obj) {  
 if (this == obj) return true;  
 if (obj == null || getClass() != obj.getClass() || !super.equals(obj)) return false;  
 Rectangular that = (Rectangular) obj;  
 return Double.*compare*(that.height, height) == 0 && Double.*compare*(that.width, width) == 0;  
 }  
}

**Controller**

import java.lang.annotation.Annotation;  
import java.lang.reflect.\*;  
  
import org.apache.commons.lang3.StringUtils;  
  
public class Controller implements Command {  
  
 View view = new View();  
 Model model = Model.*getInstance*();  
  
  
 @Override  
 public void execute(Double x, Double y, Double height, Double width) throws Exception {  
 ContextStrategy contextStrategy = new ContextStrategy(new StrategyPoint());  
 contextStrategy.ActivateAction(x, y, height, width);  
 contextStrategy.strategy = new StrategyRectangular();  
 contextStrategy.ActivateAction(x, y, height, width);  
 }  
  
 class StrategyPoint implements Strategy {  
  
 @Override  
 public void Action(Double x, Double y, Double height, Double width) throws Exception {  
 Point\_Class PointObject = model.Point\_Class(x, y);  
 AnnotationMethodInvoke(PointObject, Point\_Class.class);  
 gettingClassMethods(PointObject);  
 getAnnotations(PointObject);  
 ToString(PointObject);  
 gettingClassModifier(PointObject);  
 getConstructors(PointObject);  
 gettingClassFields(PointObject);  
 NameClass(Point\_Class.class);  
 gettingInterfaces(Point\_Class.class);  
 ProxyObject(PointObject);  
  
 System.*out*.println();  
 System.*out*.println("Iterator: ");  
 Iterator iterator = model.getIterator("Point");  
 while (iterator.hasnext()){  
 System.*out*.println(iterator.next());  
 }  
 }  
 }  
  
 class StrategyRectangular implements Strategy {  
 @Override  
 public void Action(Double x, Double y, Double height, Double width) throws Exception {  
 Rectangular RectangularObject = model.Rectangular(x, y, height, width);  
 AnnotationMethodInvoke(RectangularObject,Rectangular.class);  
 gettingClassMethods(RectangularObject);  
 getAnnotations(RectangularObject);  
 ToString(RectangularObject);  
 gettingClassModifier(RectangularObject);  
 getConstructors(RectangularObject);  
 gettingClassFields(RectangularObject);  
 NameClass(Rectangular.class);  
 gettingInterfaces(Rectangular.class);  
 ProxyObject(RectangularObject);  
  
 System.*out*.println();  
 Iterator iterator = model.getIterator("Rectangular");  
 while (iterator.hasnext()) {  
 System.*out*.println(iterator.next());  
 }  
 }  
 }  
  
 class ContextStrategy {  
  
 Strategy strategy;  
  
 public ContextStrategy(Strategy strategy){  
 this.strategy = strategy;  
 }  
  
 public void ActivateAction(Double x, Double y, Double height, Double width) throws Exception {  
 strategy.Action(x, y, height, width);  
 }  
 }  
  
  
 void ToString (Object object) {  
 view.toStringObj(object);  
 }  
  
 void ProxyObject(Object object){  
 view.proxyView((Virtual) Proxy.*newProxyInstance*(object));  
 }  
  
 double AnnotationMethodInvoke(Object PointObject, Class clazz) {  
 for(Method method: clazz.getDeclaredMethods()) {  
 Annotation annotation = method.getAnnotation(AnnotationReflectable.class);  
 if (annotation != null && annotation.annotationType() == AnnotationReflectable.class) {  
 Object[] params = new Object[method.getParameterTypes().length];  
 try {  
 view.annotationInvokeView((double) method.invoke(PointObject, params));  
 return (double) method.invoke(PointObject, params);  
 } catch (IllegalAccessException | InvocationTargetException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 return 0;  
 }  
  
 String NameClass(Class obj) {  
 view.classNameView(obj.getName());  
 return obj.getName();  
 }  
  
 StringBuffer gettingClassFields(Object object) {  
  
 StringBuffer stringBuffer = new StringBuffer();  
  
 try {  
 for (Field field : object.getClass().getDeclaredFields()) {  
 field.setAccessible(true);  
 stringBuffer.append(object.getClass().getSimpleName() + "." + field.getName() + " = " + field.get(object) +  
 "\ttype: " + field.getType() + "\tmodifier: " + Modifier.*toString*(field.getModifiers()) + "\n");  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
  
 view.fieldsView(stringBuffer);  
 return stringBuffer;  
 }  
  
 StringBuffer gettingClassMethods(Object object) throws Exception {  
 StringBuffer stringBuffer = new StringBuffer();  
 for (Method method : object.getClass().getMethods()) {  
 if (method.isAnnotationPresent(AnnotationReflectable.class)) {  
 method.setAccessible(true);  
 stringBuffer.append(object.getClass().getSimpleName() + "." +  
 method.getName() + "(" + *get\_types*(method) + ") = " + method.invoke(object) + "\n");  
 }  
 }  
 view.methodsView(stringBuffer);  
 return stringBuffer;  
 }  
  
 StringBuffer getAnnotations(Object object) throws Exception {  
 StringBuffer stringBuffer = new StringBuffer();  
 for (Method method : object.getClass().getDeclaredMethods()) {  
 if (method.isAnnotationPresent(AnnotationReflectable.class)){  
 for (Annotation annotation : method.getDeclaredAnnotations()) {  
 stringBuffer.append(annotation + "\n"); // System.out.println(Arrays.toString(method.getAnnotations()));  
 }  
 }  
 }  
 view.annotationsView(stringBuffer);  
 return stringBuffer;  
 }  
  
 String gettingClassModifier(Object object) {  
 String modifierName = Modifier.*toString*(object.getClass().getModifiers());  
 view.modifiersView(modifierName);  
 return modifierName;  
 }  
  
 StringBuffer gettingInterfaces(Class clazz) {  
  
 StringBuffer stringBuffer = new StringBuffer();  
  
 for (Class inter : clazz.getInterfaces()) {  
 stringBuffer.append(inter.getName() + " ");  
 }  
  
 view.interfacesView(stringBuffer);  
 return stringBuffer;  
 }  
  
 StringBuffer getConstructors(Object object) {  
  
 StringBuffer stringBuffer = new StringBuffer();  
  
 for (Constructor<?> constructor : object.getClass().getConstructors()) {  
 stringBuffer.append(constructor.getName() + "(" + *get\_types*(constructor) + ")\n");  
 }  
  
 view.constructorsView(stringBuffer);  
 return stringBuffer;  
 }  
  
 public static String get\_types(Constructor<?> constructor) {  
 return StringUtils.*join*(constructor.getParameterTypes(), ", ");  
 }  
  
 public static String get\_types(Method method) {  
 return StringUtils.*join*(method.getParameterTypes(), ", ");  
 }  
}

**Model**

import java.util.ArrayList;  
import java.util.Objects;  
  
public class Model implements Container {  
  
 ArrayList<Point\_Class> ListPoint = new ArrayList<>();  
 ArrayList<Rectangular> ListRectangular = new ArrayList<>();  
  
 private static Model *instance*;  
  
 Factory factory = new Factory();  
  
 private Model() {}  
  
 public static Model getInstance() {  
 return (*instance* == null) ? (*instance* = new Model()) : *instance*;  
 }  
  
 public Point\_Class Point\_Class(double x, double y) {  
 Point\_Class point = (Point\_Class) factory.create("Point", x, y, 1.0, 1.0);  
 ListPoint.add(point);  
 return point;  
 }  
  
 public Rectangular Rectangular(double x, double y, double height, double width) {  
 Rectangular rectangular = (Rectangular) factory.create("Rectangular", x, y, height, width);  
 ListRectangular.add(rectangular);  
 return rectangular;  
 }  
  
 @Override  
 public Iterator getIterator(String name) {  
 return Objects.*equals*(name, "Point") ? new ArrayListPointIterator() : new ArrayListRectangularIterator();  
 }  
  
 class ArrayListPointIterator implements Iterator {  
  
 int index;  
  
 @Override  
 public boolean hasnext() {  
 return index < ListPoint.size();  
 }  
  
 @Override  
 public Object next() {  
 return hasnext() ? ListPoint.get(index++) : null;  
 }  
 }  
  
 class ArrayListRectangularIterator implements Iterator {  
  
 int index;  
  
 @Override  
 public boolean hasnext() {  
 return index < ListRectangular.size();  
 }  
  
 @Override  
 public Object next() {  
 return hasnext() ? ListRectangular.get(index++) : null;  
 }  
 }  
}

**View**

public class View {  
  
 public void toStringObj(Object object) {  
 System.*out*.println(object.toString());  
 }  
  
 public void annotationInvokeView(double annotation) {  
 System.*out*.println("Annotation methods invoke " + annotation + "\n");  
 }  
  
 public void methodsView(StringBuffer methods) {  
 System.*out*.println("Methods:\n" + methods + "\n");  
 }  
  
 public void annotationsView(StringBuffer methods) {  
 System.*out*.println("Annotations:\n" + methods + "\n");  
 }  
  
 public void classNameView(String nameClass) {  
 System.*out*.println("Class Name : " + nameClass+ "\n");  
 }  
  
 public void fieldsView(StringBuffer fields) {  
 System.*out*.println("Class Fields:\n" + fields.toString() + "\n");  
 }  
  
 public void modifiersView(String modifierClass) {  
 System.*out*.println("Class Modifier: " + modifierClass + "\n");  
 }  
  
 public void constructorsView(StringBuffer constructors) {  
 System.*out*.println("Constructors:\n" + constructors.toString() + "\n");  
 }  
  
 public void interfacesView(StringBuffer interfaces) {  
 System.*out*.println("Class Interfaces: " + interfaces.toString() + "\n");  
 }  
  
 public void proxyView(Virtual proxy) {  
 System.*out*.println("Proxy: " + proxy.getInterface());  
 }  
}

**Proxy**

import java.lang.reflect.InvocationHandler;  
import java.lang.reflect.Method;  
  
class Proxy implements InvocationHandler {  
 private Object ref;  
 private Proxy(Object ref) {  
 this.ref = ref;  
 }  
  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 if(method.getName().startsWith("get")) {  
 return method.invoke(ref, args);  
 }  
 throw new IllegalAccessException("Not allowed");  
 }  
  
 public static Object newProxyInstance(Object obj) {  
 return java.lang.reflect.Proxy.*newProxyInstance*(  
 obj.getClass().getClassLoader(),  
 obj.getClass().getInterfaces(),  
 new Proxy(obj));  
 }  
}

**ProxyPattern**

import java.util.ArrayList;  
import java.util.Scanner;  
  
public class ProxxyPattern implements Command {  
 @Override  
 public void execute(Double x, Double y, Double height, Double width) throws Exception {  
 new Controller().execute(x < 5 ? 1 : x, y < 5 ? 1 : y, height < 5 ? 1 : height, width < 5 ? 1 : width);  
 }  
}

**Factory**

public class Factory {  
 public Virtual create(String name, Double x, Double y, Double height, Double width) {  
 switch (name) {  
 case "Point": return new Point\_Class(x, y);  
 case "Rectangular": return new Rectangular(x, y, height, width);  
 default: return null;  
 }  
 }  
}

**AnnotationReflectable**

import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
  
@Retention(RetentionPolicy.*RUNTIME*)  
@Target(ElementType.*METHOD*)  
public @interface AnnotationReflectable {  
 public String name();  
}

**Virtual**

public interface Virtual {  
  
 void moving(double x, double y);  
  
 default void moving() {  
 System.*out*.println("moving");  
 }  
  
 double getX();  
  
 double getY();  
  
 void setX(double x);  
  
 void setY(double y);  
  
 double getHeight();  
  
 void setHeight(double height);  
  
 double getWidth();  
  
 void setWidth(double width);  
  
 double area();  
  
 double perimeter();  
  
}

**Command**

public interface Command {  
 void execute(Double x, Double y, Double height, Double width) throws Exception;  
}

**Iterator**

public interface Iterator {  
 boolean hasnext();  
 Object next();  
}  
interface Container {  
 Iterator getIterator(String name);  
}

**Strategy**

public interface Strategy {  
 void Action(Double x, Double y, Double height, Double width) throws Exception;  
}

**Tests**

**Point\_ClassTest**

import org.junit.jupiter.api.AfterEach;  
 import org.junit.jupiter.api.BeforeAll;  
 import org.junit.jupiter.api.BeforeEach;  
 import org.junit.jupiter.api.Test;  
  
 import static org.junit.jupiter.api.Assertions.assertEquals;  
 import static org.junit.jupiter.api.Assertions.assertNotEquals;  
  
  
class Point\_ClassTest {  
  
 Point\_Class TestObject = new Point\_Class();  
  
 @Test  
 void getX() {  
 TestObject.setX(3);  
 assertEquals(TestObject.getX(), 3);  
 assertNotEquals(TestObject.getX(), 5);  
 }  
  
 @Test  
 void getY() {  
 TestObject.setY(6);  
 assertEquals(TestObject.getY(), 6);  
 assertNotEquals(TestObject.getY(), 5);  
 }  
  
 /\* Test  
 void setX() {  
 }  
  
 @Test  
 void setY() {  
 }  
 \*/  
  
 @Test  
 void moving() {  
 TestObject.setX(3);  
 TestObject.setY(4);  
 TestObject.moving(1, 3);  
 assertEquals(new Point\_Class(4, 7), TestObject);  
 assertNotEquals(new Point\_Class(3, 4), TestObject);  
 }  
  
 @Test  
 void testToString() {  
  
 TestObject.setX(3);  
 TestObject.setY(4);  
  
 assertEquals("point {x = 3.0; y = 4.0}", TestObject.toString());  
 assertNotEquals("point {x = 3.0; y = 4.0", TestObject.toString());  
 }  
  
}

**Point\_ClassTest**

import org.junit.jupiter.api.BeforeEach;  
 import org.junit.jupiter.api.Test;  
  
 import java.io.ByteArrayOutputStream;  
 import java.io.PrintStream;  
  
 import static org.junit.jupiter.api.Assertions.\*;  
  
  
class RectangularTest {  
  
 Rectangular TestObject = new Rectangular();  
 Rectangular Rectangle = new Rectangular(4, 9, 8.5, 7);  
  
// private final ByteArrayOutputStream output = new ByteArrayOutputStream();  
//  
// @BeforeEach  
// void setUp() {  
// System.setOut(new PrintStream(output));  
// }  
  
 @Test  
 void getHeight() {  
 TestObject.setHeight(7.9);  
 assertEquals(7.9, TestObject.getHeight());  
 assertNotEquals(2.9, TestObject.getHeight());  
 }  
  
 @Test  
 void getWidth() {  
 TestObject.setWidth(5);  
 assertEquals(5, TestObject.getWidth());  
 assertNotEquals(2.9, TestObject.getWidth());  
 }  
  
 /\*  
 @Test  
 void setWidth() {  
 }  
  
  
 @Test  
 void setHeight() {  
 }  
  
  
  
 @Test  
 void area() {  
 TestObject.setX(9);  
 TestObject.setY(4);  
 assertEquals(36, TestObject.area());  
 assertNotEquals(64.9, TestObject.area());  
 }  
  
 \*/  
  
 @Test  
 void area() {  
 assertEquals(59.5, Rectangle.area());  
 assertNotEquals(64.9, Rectangle.area());  
 }  
  
 @Test  
 void perimeter() {  
 assertEquals(31, Rectangle.perimeter());  
 assertNotEquals(69, Rectangle.perimeter());  
 }  
  
 @Test  
 void isEmpty() {  
 assertFalse(Rectangle.isEmpty());  
 assertNotEquals(true, Rectangle.isEmpty());  
 }  
  
 @Test  
 void contains() {  
 assertTrue(Rectangle.contains(5, 12));  
 assertFalse(Rectangle.contains(2, 12));  
 }  
  
 @Test  
 void moving() {  
 Rectangle.moving(1, 3);  
 assertEquals(new Rectangular(5, 12, 8.5, 7), Rectangle);  
 assertNotEquals(new Rectangular(3, 4, 8.5, 7), Rectangle);  
 }  
  
 @Test  
 void testToString() {  
 assertEquals("Rectangular {height = 8.5, width = 7.0}", Rectangle.toString());  
 assertNotEquals("Rectangular {height = 8.5, width = 7}", Rectangle.toString());  
 }  
  
  
}

**MainTest**

import org.junit.jupiter.api.BeforeAll;  
 import org.junit.jupiter.api.Test;  
 import java.util.ArrayList;  
 import java.util.Arrays;  
  
 import static org.junit.jupiter.api.Assertions.assertEquals;  
  
class MainTest {  
  
 Rectangular Rectangle = new Rectangular(5, 6, 7, 8);  
  
// ArrayList<String> Expected\_constructors = new ArrayList<String>(  
// Arrays.asList("Rectangular(class Rectangular)",  
// "Rectangular()", "Rectangular(double, double)", "Rectangular(double, double, double, double)"));  
  
 ArrayList<String> Expected\_methods = new ArrayList<>(Arrays.asList(  
 "Rectangular.area() = 56.0", "Rectangular.getHeight() = 7.0", "Rectangular.getWidth() = 8.0",  
 "Rectangular.getX() = 5.0", "Rectangular.getY() = 6.0", "Rectangular.perimeter() = 30.0")  
 );  
  
// ArrayList<String> Expected\_fields = new ArrayList<>(Arrays.asList(  
// "Rectangular.height = 7.0 type: double modifier: private", "Rectangular.width = 8.0 type: double modifier: private)"  
// ));  
//  
  
 @BeforeAll  
 static void setUp() {  
 }  
  
 @Test  
 void getting\_class\_methods() {  
 try {  
 assertEquals(Main.*getting\_class\_methods*(Rectangle), Expected\_methods);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 @Test  
 void check\_invocation() {  
 try {  
 Main.*check\_invocation*(Rectangle);  
 } catch (Exception e) {  
 assertEquals("java.lang.RuntimeException: Object immutable", e.toString());  
 }  
 }  
}

**ProxyTest**

import org.junit.jupiter.api.Test;  
  
 import static org.junit.jupiter.api.Assertions.\*;  
  
class ProxyTest {  
  
 Point\_Class PointObject = new Point\_Class(1.0,1.0);  
 Rectangular RectangularObject = new Rectangular(2.0, 2.0, 10, 15);  
  
 static Class *reflected* = Rectangular.class;  
  
 @Test  
 void invoke() {  
 Virtual proxy = (Virtual) Proxy.newProxyInstance(PointObject);  
 assertEquals("Getter", proxy.getInterface());  
 assertNotEquals("Not allowed", proxy.getInterface());  
 assertNotEquals("Not allowed", proxy.getInterface());  
 }  
}

**ProxyTest**

import org.apache.commons.lang3.StringUtils;  
 import org.junit.jupiter.api.Test;  
 import org.mockito.\*;  
  
 import java.lang.annotation.Annotation;  
 import java.lang.reflect.Constructor;  
 import java.lang.reflect.Field;  
 import java.lang.reflect.Method;  
 import java.lang.reflect.Modifier;  
  
 import static org.junit.jupiter.api.Assertions.assertEquals;  
 import static org.junit.jupiter.api.Assertions.assertNotEquals;  
  
class ControllerTest {  
  
 Controller controller = Mockito.mock(Controller.class);  
 Point\_Class point = Mockito.mock(Point\_Class.class);  
 Rectangular rectangular = Mockito.mock(Rectangular.class);  
 Rectangular rectangularObject = new Rectangular(9, 9, 9, 9);  
 Point\_Class pointObject = new Point\_Class(9, 9);  
  
  
 StringBuffer kek = new StringBuffer("kek");  
 StringBuffer kekekek = new StringBuffer("mda");  
  
// @Test  
// void proxyObject() {  
// Mockito.doThrow(new IllegalAccessException()).when(controller).ProxyObject(point);  
// controller.ProxyObject(point);  
// }  
  
 @Test  
 void annotationMethodInvoke() {  
 Point\_Class point = new Point\_Class(1, 1);  
 Mockito.when(controller.AnnotationMethodInvoke(point, Point\_Class.class)).thenReturn(1.);  
 assertEquals(controller.AnnotationMethodInvoke(point, Point\_Class.class), 1.);  
 controller.AnnotationMethodInvoke(rectangular, Rectangular.class);  
 Mockito.verify(controller).AnnotationMethodInvoke(rectangular, Rectangular.class);  
 }  
  
 @Test  
 void nameClass() {  
 Mockito.when(controller.NameClass(Point\_Class.class)).thenReturn("P");  
 assertEquals(controller.NameClass(Point\_Class.class), "P");  
 assertNotEquals(controller.NameClass(Point\_Class.class), "R");  
 controller.NameClass(Rectangular.class);  
 Mockito.verify(controller).NameClass(Rectangular.class);  
 }  
  
 @Test  
 void gettingClassFields() {  
 Mockito.when(controller.gettingClassFields(rectangularObject)).thenReturn(kek);  
 assertEquals(controller.gettingClassFields(rectangularObject), kek);  
 }  
  
 @Test  
 void gettingClassMethods() throws Exception {  
 Mockito.when(controller.gettingClassMethods(rectangularObject)).thenReturn(kek);  
 assertEquals(controller.gettingClassMethods(rectangularObject), kek);  
 }  
  
 @Test  
 void getAnnotations() throws Exception {  
 Mockito.when(controller.getAnnotations(rectangular)).thenReturn(kek);  
 assertEquals(controller.getAnnotations(rectangular), kek);  
 }  
  
 @Test  
 void gettingClassModifier() {  
 Mockito.when(controller.gettingClassModifier(rectangular)).thenReturn("KEK");  
 assertEquals(controller.gettingClassModifier(rectangular), "KEK");  
 }  
  
 @Test  
 void gettingInterfaces() {  
 Mockito.when(controller.gettingInterfaces(Rectangular.class)).thenReturn(kek);  
 assertEquals(controller.gettingInterfaces(Rectangular.class), kek);  
 }  
  
 @Test  
 void getConstructors() {  
 Mockito.when(controller.getConstructors(rectangularObject)).thenReturn(kek);  
 assertEquals(controller.getConstructors(rectangularObject), kek);  
 assertNotEquals(controller.getConstructors(rectangularObject), kekekek);  
 }  
}

**ControllerTest**

import org.apache.commons.lang3.StringUtils;  
import org.junit.jupiter.api.Test;  
import org.mockito.\*;  
  
import java.lang.annotation.Annotation;  
import java.lang.reflect.Constructor;  
import java.lang.reflect.Field;  
import java.lang.reflect.Method;  
import java.lang.reflect.Modifier;  
  
import static org.junit.jupiter.api.Assertions.*assertEquals*;  
import static org.junit.jupiter.api.Assertions.*assertNotEquals*;  
  
class ControllerTest {  
  
 Controller controller = Mockito.*mock*(Controller.class);  
 Point\_Class point = Mockito.*mock*(Point\_Class.class);  
 Rectangular rectangular = Mockito.*mock*(Rectangular.class);  
 Rectangular rectangularObject = new Rectangular(9, 9, 9, 9);  
 Point\_Class pointObject = new Point\_Class(9, 9);  
  
  
 StringBuffer kek = new StringBuffer("kek");  
 StringBuffer kekekek = new StringBuffer("mda");  
  
// @Test  
// void proxyObject() {  
// Mockito.doThrow(new IllegalAccessException()).when(controller).ProxyObject(point);  
// controller.ProxyObject(point);  
// }  
  
 @Test  
 void annotationMethodInvoke() {  
 Point\_Class point = new Point\_Class(1, 1);  
 Mockito.*when*(controller.AnnotationMethodInvoke(point, Point\_Class.class)).thenReturn(1.);  
 *assertEquals*(controller.AnnotationMethodInvoke(point, Point\_Class.class), 1.);  
 controller.AnnotationMethodInvoke(rectangular, Rectangular.class);  
 Mockito.*verify*(controller).AnnotationMethodInvoke(rectangular, Rectangular.class);  
 }  
  
 @Test  
 void nameClass() {  
 Mockito.*when*(controller.NameClass(Point\_Class.class)).thenReturn("P");  
 *assertEquals*(controller.NameClass(Point\_Class.class), "P");  
 *assertNotEquals*(controller.NameClass(Point\_Class.class), "R");  
 controller.NameClass(Rectangular.class);  
 Mockito.*verify*(controller).NameClass(Rectangular.class);  
 }  
  
 @Test  
 void gettingClassFields() {  
 Mockito.*when*(controller.gettingClassFields(rectangularObject)).thenReturn(kek);  
 *assertEquals*(controller.gettingClassFields(rectangularObject), kek);  
 }  
  
 @Test  
 void gettingClassMethods() throws Exception {  
 Mockito.*when*(controller.gettingClassMethods(rectangularObject)).thenReturn(kek);  
 *assertEquals*(controller.gettingClassMethods(rectangularObject), kek);  
 }  
  
 @Test  
 void getAnnotations() throws Exception {  
 Mockito.*when*(controller.getAnnotations(rectangular)).thenReturn(kek);  
 *assertEquals*(controller.getAnnotations(rectangular), kek);  
 }  
  
 @Test  
 void gettingClassModifier() {  
 Mockito.*when*(controller.gettingClassModifier(rectangular)).thenReturn("KEK");  
 *assertEquals*(controller.gettingClassModifier(rectangular), "KEK");  
 }  
  
 @Test  
 void gettingInterfaces() {  
 Mockito.*when*(controller.gettingInterfaces(Rectangular.class)).thenReturn(kek);  
 *assertEquals*(controller.gettingInterfaces(Rectangular.class), kek);  
 }  
  
 @Test  
 void getConstructors() {  
 Mockito.*when*(controller.getConstructors(rectangularObject)).thenReturn(kek);  
 *assertEquals*(controller.getConstructors(rectangularObject), kek);  
 *assertNotEquals*(controller.getConstructors(rectangularObject), kekekek);  
 }  
}