CSSE 374: Lab 10-1

The goal of this lab is to explore the expansive possibilities of the **Proxy Pattern.** You will use Java’s dynamic proxy to dynamically create concrete implementations of arbitrarily many interfaces. You will use the generated implementations to describe a regional favorite sports team. To do these, you will provide a **universal factory** that can create objects of a given interface type with default implementation for the common methods such as getter, setters, numerical incrementers, and collection appenders.

# Background

We’ve seen that Maps can be used to separate what changes from what stays the same. Let’s apply them to fix the following major design flaw in the Java language: maintaining getter and setter code.

Consider the following interfaces for the data model of a fantasy football app:

public interface Team{

String getName();

void setName(String name);

int getWins();

void incrementWins();

Roster getOffensiveStartingLineup();

void setOffensiveStartingLineup(Roster r);

… other Team stats here

}

public interface Roster{

String getName();

void setName(String name);

List<Player> getPlayers;

void setPlayers(List<Player>);

void addPlayerToPlayers(Player p);

…. Other roster stats here

}

public interface Player{

String getName();

void setName(String name);

… other player stats here

}

As you can tell, the data model’s interfaces describe getters and setters for important pieces of the problem domain. The nice thing about getters and setters is that we get compile time type checking. One of the major pains of getters/setters is writing and maintaining their code. Whenever the problem domain changes, we have to go back and change both the interface and the implementation to reflect the new getters and setters. This may result in duplicated code.

For example, a Team has a name. Classes that implement Team must implement getName and setName. If we want Roster to also have a name, then we must add getName and setName to its interface and copy/paste the same getName/setName code from Team to all classes that implement Roster. You’ve probably done something similar yourself! We could refactor everything into something Nameable and have everything extend Nameable, but what if we want to extend something else too? As we’ve seen, programming to an implementation via inheritance can have unintended consequences.

We could alternatively implement all of these Teams, Rosters, and Players as straight HashMaps. However, if we use raw Maps to represent the heterogeneous types in the above interfaces, then we bypass compile-time type checking, which can cause headaches when we try to remember what type is returned by a raw HashMap.get and will result in bug-prone code. If we use a class containing a HashMap to implement the above interfaces, we get our type checking back, but we have to write the getters/setters ourselves, so we are back to square one. Thus, what wereallywant is to *generate* implementations of the getter/setter interfaces and back them with the HashMap.

# Feature

Write a class, **UniversalObjectFactory**, with one factory method, **create(Class<?> interfaze)**. As the method signature suggests, it takes an interface as a parameter and generates an instance of a *concrete* *class* that implements all of the listed interfaces’ getters and setters. For example, UniversalObjectFactory.create(Team.class) should return a new concrete instance of the interface Team with all of its interface methods fully implemented. Call create several times with different parameters to generate concrete instances implementing different interfaces.

UniversalObjectFactory must generate implementations of the following interface methods:

* **Getters** (NNN getNnn())**:** get the current value of the specified instance variable. For example, getName() should return the current value of the instance variable Name.
* **Setters** (void setNNN(NNN value)): set the current value of the specified instance variable. For example, setName(“Bob’s team”) should set the instance variable Name to Bob’s team.
* **Incrementers** (void incrementNnn()): increase the value of the instance variable by one.
* **Add** (void addNNNtoNnns(NNN value)): add the value to the specified instance variable of type Collection. For example, addPlayerToPlayers(Player p) should add the given Player to the Collection named Players.
* **toString():** returns a multline string representing this object. The string must recursively list all instance variables in this object, including their field names, types, and toString() values. Use nice formatting and indentation.
* **Anything else**: throw an UnsupportedOperationException.

Demonstrate your universal getter/setter code by building the complete data model for the Indianapolis Colts’ starting lineups in the method Main.main (see the input/output dir for data). Represent their team using the Team object, the offensive and defensive starting lineups using Rosters, and each player using a Player. Print everything about the team (stats, Rosters, and Players) by invoking the Team object’s toString() once and printing the returned value. Program to the Team, Roster, and Player interfaces.

**Do not write any implementations** **for Team, Roster, or Player**; use UniversalObjectFactory to generate them. As well, UniversalObjectFactory and the above methods must be **completely decoupled** from the Team, Roster, and Player interfaces.

# Design (C)

Create a **UML Class Diagram** to describe the complete system. Have your design checked-off by a TA.

# Implementation (B)

Implement the incomplete method, UniversalObjectFactory.create.

# Testing (A)

Complete the Main.main method that build the complete Colts’ Team object and prints the resulting Team object’s toString(). Verify all three starting lines are printed as a result as well as the other stored information. Also write automated JUnit tests for each type of method described in the feature specification.

# Deliverable

Demo your design and lab to a TA.