**Refactoring:**

Refactoring code involves restructuring existing code without changing its external behavior. The goal is to improve readability, maintainability, and sometimes performance, making the code more efficient and easier to understand without altering its functionality.

**Task:**

**Before Refactoring:**

package strategypatterntest;

abstract class Team {

String teamName;

TeamStrategy teamStrategy;

public void playGame() {

System.out.println("Playing game...");

}

public abstract void playInfo();

}

class TeamGermany extends Team {

public TeamGermany() {

teamName = "Germany";

teamStrategy = new AttackStrategy();

}

@Override

public void playInfo() {

System.out.println("Playing info for Germany...");

}

}

class TeamArgentina extends Team {

public TeamArgentina() {

teamName = "Argentina";

teamStrategy = new DefendStrategy();

}

@Override

public void playInfo() {

System.out.println("Playing info for Argentina...");

}

}

interface TeamStrategy {

void play();

}

class AttackStrategy implements TeamStrategy {

@Override

public void play() {

System.out.println("Playing attack strategy...");

}

}

class DefendStrategy implements TeamStrategy {

@Override

public void play() {

System.out.println("Playing defend strategy...");

}

}

public class StrategyPatternTest {

public static void main(String[] args) {

Team team1 = new TeamGermany();

Team team2 = new TeamArgentina();

team1.playGame();

team1.playInfo();

team1.teamStrategy.play();

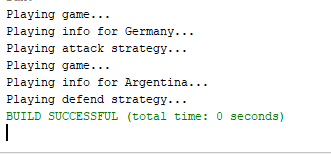
team2.playGame();

team2.playInfo();

team2.teamStrategy.play();

}

}



**After Refactoring:**

package strategypatterntest;

abstract class Team {

String teamName;

TeamStrategy teamStrategy;

public Team(String teamName, TeamStrategy teamStrategy) {

this.teamName = teamName;

this.teamStrategy = teamStrategy;

}

public void playGame() {

System.out.println("Playing game...");

}

public abstract void playInfo();

public void executeStrategy() {

teamStrategy.play();

}

}

class TeamGermany extends Team {

public TeamGermany() {

super("Germany", new AttackStrategy());

}

@Override

public void playInfo() {

System.out.println("Playing info for Germany...");

}

}

class TeamArgentina extends Team {

public TeamArgentina() {

super("Argentina", new DefendStrategy());

}

@Override

public void playInfo() {

System.out.println("Playing info for Argentina...");

}

}

interface TeamStrategy {

void play();

}

class AttackStrategy implements TeamStrategy {

@Override

public void play() {

System.out.println("Playing attack strategy...");

}

}

class DefendStrategy implements TeamStrategy {

@Override

public void play() {

System.out.println("Playing defend strategy...");

}

}

public class StrategyPatternTest {

public static void main(String[] args) {

Team team1 = new TeamGermany();

Team team2 = new TeamArgentina();

team1.playGame();

team1.playInfo();

team1.executeStrategy();

team2.playGame();

team2.playInfo();

team2.executeStrategy();

}

}

I have added a constructor in the Team class for better initialization, renamed the team1.teamStrategy.play() calls to team1.executeStrategy(), and made minor adjustments for consistency and clarity.

