**BUILDER**

class Pizza {

private dough: string = '';

private salsa: string = '';

private topping: string = '';

public setDough(dough: string): void {

this.dough = dough;

}

public setSauce(sauce: string): void {

this.salsa = sauce;

}

public setTopping(topping: string): void {

this.topping = topping;

}

}

abstract class PizzaBuilder {

protected pizza: Pizza;

public getPizza(): Pizza {

return this.pizza;

}

public createNewPizzaProduct(): void {

this.pizza = new Pizza();

}

public abstract buildDough(): void;

public abstract buildSauce(): void;

public abstract buildTopping(): void;

}

class HawaiianPizzaBuilder extends PizzaBuilder {

public buildDough(): void {

this.pizza.setDough("cross");

}

public buildSauce(): void {

this.pizza.setSauce("mild");

}

public buildTopping(): void {

this.pizza.setTopping("ham+pineapple");

}

}

class SpicyPizzaBuilder extends PizzaBuilder {

public buildDough(): void {

this.pizza.setDough("pan baked");

}

public buildSauce(): void {

this.pizza.setSauce("hot");

}

public buildTopping(): void {

this.pizza.setTopping("pepperoni+salami");

}

}

class Waiter {

private pizzaBuilder: PizzaBuilder;

public setPizzaBuilder(pb: PizzaBuilder): void {

this.pizzaBuilder = pb;

}

public getPizza(): Pizza {

return this.pizzaBuilder.getPizza();

}

public constructPizza(): void {

this.pizzaBuilder.createNewPizzaProduct();

this.pizzaBuilder.buildDough();

this.pizzaBuilder.buildSauce();

this.pizzaBuilder.buildTopping();

}

}

let waiter = new Waiter();

let hawaiianPizzabuilder: PizzaBuilder = new HawaiianPizzaBuilder();

let spicyPizzaBuilder: PizzaBuilder = new SpicyPizzaBuilder();

waiter.setPizzaBuilder(hawaiianPizzabuilder);

waiter.constructPizza();

let pizza: Pizza = waiter.getPizza();

**FACTORY**

class DecodedImage {

private image: string;

constructor(image: string) {

this.image = image;

}

public toString(): string {

return this.image + ": is decoded";

}

}

interface ImageReader {

getDecodeImage(): DecodedImage;

}

class GifReader implements ImageReader {

private decodedImage: DecodedImage;

constructor(image: string) {

this.decodedImage = new DecodedImage(image);

}

public getDecodeImage(): DecodedImage {

return this.decodedImage;

}

}

class JpegReader implements ImageReader {

private decodedImage: DecodedImage;

constructor(image: string) {

this.decodedImage = new DecodedImage(image);

}

public getDecodeImage(): DecodedImage {

return this.decodedImage;

}

}

let args: string[] = process.argv.slice(2);

let decodedImage: DecodedImage;

let reader: ImageReader = null;

let image: string = args[0];

let format: string = image.substring(image.indexOf('.') + 1, (image.length));

if (format === "gif")

reader = new GifReader(image);

if (format === "jpeg")

reader = new JpegReader(image);

if (reader != null)

throw new Error("COMPLETAR");

decodedImage = reader.getDecodeImage();

console.log(decodedImage);

**SINGLETON**

class Singleton {

private static readonly instance: Singleton = new Singleton();

private constructor() {}

static getInstance(): Singleton {

return Singleton.instance;

}

}

**COMPOSITE**

interface Component {

traverse(): void;

}

class Primitive implements Component {

private value: number;

public constructor(val: number) {

this.value = val;

}

public traverse(): void {

console.log(this.value + " ");

}

}

abstract class Composite implements Component {

private children: Component[] = new Array[9];

private total: number = 0;

private value: number;

public constructor(val: number) {

this.value = val;

}

public add(c: Component): void {

this.children[this.total++] = c;

}

public traverse(): void {

console.log(this.value + " ");

for (let i = 0; i < this.total; i++)

this.children[i].traverse();

}

}

class Row extends Composite {

public constructor(val: number) {

super(val);

}

public traverse(): void {

console.log("Row");

super.traverse();

}

}

class Column extends Composite {

public constructor(val: number) {

super(val);

}

public traverse(): void {

console.log("Col");

super.traverse();

}

}

let first: Composite = new Row(1);

let second: Composite = new Column(2);

let third: Composite = new Column(3);

let fourth: Composite = new Row(4);

let fifth: Composite = new Row(5);

first.add(second);

first.add(third);

third.add(fourth);

third.add(fifth);

first.add(new Primitive(6));

second.add(new Primitive(7));

third.add(new Primitive(8));

fourth.add(new Primitive(9));

fifth.add(new Primitive(10));

first.traverse();

**ADAPTER**

/\* The OLD \*/

class SquarePeg {

private width: number;

public constructor(width: number) {

this.width = width;

}

public getWidth(): number {

return this.width;

}

public setWidth(width: number): void {

this.width = width;

}

}

/\* The NEW \*/

class RoundHole {

private radius: number;

public constructor(radius: number) {

this.radius = radius;

console.log("RoundHole: max SquarePeg is " + this.radius \* Math.sqrt(2));

}

public getRadius(): number {

return this.radius;

}

}

// Design a "wrapper" class that can "impedance match" the old to the new

class SquarePegAdapter {

// The adapter/wrapper class "has a" instance of the legacy class

private squarePeg: SquarePeg;

public constructor(w: number) {

this.squarePeg = new SquarePeg(w);

}

// Identify the desired interface

public makeFit(roundHole: RoundHole): void {

// The adapter/wrapper class delegates to the legacy object

let amount: number = this.squarePeg.getWidth() - roundHole.getRadius() \* Math.sqrt(2);

console.log("reducing SquarePeg " + this.squarePeg.getWidth() + " by " + ((amount < 0) ? 0 : amount) + " amount");

if (amount > 0) {

this.squarePeg.setWidth(this.squarePeg.getWidth() - amount);

console.log(" width is now " + this.squarePeg.getWidth());

}

}

}

let roundHole: RoundHole = new RoundHole(5);

let squarePegAdapter: SquarePegAdapter;

for (let i = 6; i < 10; i++) {

squarePegAdapter = new SquarePegAdapter(i);

// The client uses (is coupled to) the new interface

squarePegAdapter.makeFit(roundHole);

}

**MEDIATOR**

interface ChatMediator {

sendMessage(msg: string, user: User): void;

addUser(user: User): void;

}

abstract class User {

protected mediator: ChatMediator;

protected name: string;

public constructor(mediator: ChatMediator, name: string) {

this.mediator = mediator;

this.name = name;

}

public abstract send(msg: string): void;

public abstract receive(msg: string): void;

}

class ChatMediatorImpl implements ChatMediator {

private users: Array<User>;

public constructor() {

this.users = new Array<User>();

}

public addUser(user: User): void {

this.users.push(user);

}

public sendMessage(msg: string, user: User): void {

for (let i = 0; i < this.users.length; i++) {

if (this.users[i] != user)

this.users[i].receive(msg);

}

}

}

class UserImpl extends User {

public constructor(med: ChatMediator, name: string) {

super(med, name);

}

public send(msg: string): void {

console.log(this.name + ": Sending Message=" + msg);

this.mediator.sendMessage(msg, this);

}

public receive(msg: string): void {

console.log(this.name + ": Received Message:" + msg);

}

}

let mediator: ChatMediator = new ChatMediatorImpl();

let user1: User = new UserImpl(mediator, "Pankaj");

let user2: User = new UserImpl(mediator, "Lisa");

let user3: User = new UserImpl(mediator, "Saurabh");

let user4: User = new UserImpl(mediator, "David");

mediator.addUser(user1);

mediator.addUser(user2);

mediator.addUser(user3);

mediator.addUser(user4);

user1.send("Hi All");

**STRATEGY**

// 1. Define the interface of the algorithm

interface Strategy {

solve(): void;

}

// 2. Bury implementation

abstract class StrategySolution implements Strategy {

// 3. Template Method

public solve(): void {

this.start();

while (this.nextTry() && !this.isSolution()) {

}

this.stop();

}

abstract start(): void;

abstract nextTry(): boolean;

abstract isSolution(): boolean;

abstract stop(): void;

}

class FOO extends StrategySolution {

private state: number = 1;

public start(): void {

console.log("Start ");

}

public stop(): void {

console.log("Stop");

}

public nextTry(): boolean {

console.log("NextTry-" + this.state++ + " ");

return true;

}

public isSolution(): boolean {

console.log("IsSolution-" + (this.state == 3) + " ");

return (this.state == 3);

}

}

// 2. Bury implementation

abstract class StrategySearch implements Strategy {

// 3. Template Method

public solve(): void {

while (true) {

this.preProcess();

if (this.search())

break;

this.postProcess();

}

}

abstract preProcess(): void;

abstract search(): boolean;

abstract postProcess(): void;

}

class BAR extends StrategySearch {

private state: number = 1;

public preProcess(): void {

console.log("PreProcess ");

}

public postProcess(): void {

console.log("PostProcess ");

}

public search(): boolean {

console.log("Search-" + this.state++ + " ");

return this.state == 3 ? true : false;

}

}

// 4. Clients couple strictly to the interface

function execute(strategy: Strategy): void {

strategy.solve();

}

let algorithms: Strategy[] = [new FOO(), new BAR()];

for (let i = 0; i < algorithms.length; i++) {

execute(algorithms[i]);

}

**FACADE**

// 1. Subsystem

class PointCartesian {

private x: number

private y: number;

public constructor(x: number, y: number) {

this.x = x;

this.y = y;

}

public move(x: number, y: number): void {

this.x += x;

this.y += y;

}

public toString(): string {

return "(" + this.x + "," + this.y + ")";

}

public getX(): number {

return this.x;

}

public getY(): number {

return this.y;

}

}

// 1. Subsystem

class PointPolar {

private radius: number

private angle: number;

public constructor(radius: number, angle: number) {

this.radius = radius;

this.angle = angle;

}

public rotate(angle: number): void {

this.angle += angle % 360;

}

public toString(): string {

return "[" + this.radius + "@" + this.angle + "]";

}

}

// 1. Desired interface: move(), rotate()

class Point {

// 2. Design a "wrapper" class

private pointCartesian: PointCartesian;

public constructor(x: number, y: number) {

this.pointCartesian = new PointCartesian(x, y);

}

public toString(): string {

return this.pointCartesian.toString();

}

// 4. Wrapper maps

public move(x: number, y: number): void {

this.pointCartesian.move(x, y);

}

public rotate(angle: number, o: Point): void {

let x = this.pointCartesian.getX() - o.pointCartesian.getX();

let y = this.pointCartesian.getY() - o.pointCartesian.getY();

let pointPolar: PointPolar = new PointPolar(Math.sqrt(x \* x + y \* y), Math.atan2(y, x) \* 180 / Math.PI);

// 4. Wrapper maps

pointPolar.rotate(angle);

console.log(" PointPolar is " + pointPolar);

let str = pointPolar.toString();

let i = str.indexOf('@');

let r = Number(str.substring(1, i));

let a = Number(str.substring(i + 1, str.length - 1));

this.pointCartesian = new PointCartesian(r \* Math.cos(a \* Math.PI / 180) + o.pointCartesian.getX(),

r \* Math.sin(a \* Math.PI / 180) + o.pointCartesian.getY());

}

}

class Line {

private o: Point;

private e: Point;

public constructor(ori: Point, end: Point) {

this.o = ori;

this.e = end;

}

public move(x: number, y: number): void {

this.o.move(x, y);

this.e.move(x, y);

}

public rotate(angle: number): void {

this.e.rotate(angle, this.o);

}

public toString(): string {

return "origin is " + this.o + ", end is " + this.e;

}

}

// 3. Client uses the Facade

let lineA = new Line(new Point(2, 4), new Point(5, 7));

lineA.move(-2, -4);

console.log("after move: " + lineA);

lineA.rotate(45);

console.log("after rotate: " + lineA);

let lineB = new Line(new Point(2, 1), new Point(2.866, 1.5));

lineB.rotate(30);

console.log("30 degrees to 60 degrees: " + lineB);