Strategy Pattern

Definition

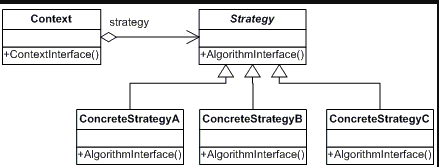
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Explanation

In computer programming, the strategy pattern (also known as the policy pattern) is a software design pattern that enables an algorithm’s behavior to be selected at runtime. The strategy pattern:

1. defines a family of algorithms,
2. encapsulates each algorithm, and
3. makes the algorithms interchangeable within that family.

Strategy lets the algorithm vary independently from clients that use it. For instance, a class that performs validation on incoming data may use a strategy pattern to select a validation algorithm based on the type of data, the source of the data, user choice, or other discriminating factors. These factors are not known for each case until run-time, and may require radically different validation to be performed. The validation strategies, encapsulated separately from the validating object, may be used by other validating objects in different areas of the system (or even different systems) without code duplication.



TypeScript Code

module Strategy {

interface ISortStrategy<T> {

sort<T>(items: Array<T>, comparer: (left: T, right: T) => number);

}

class StandardSortStrategy<T> implements ISortStrategy<T> {

public sort<T>(items: Array<T>, comparer: (left: T, right: T) => number) {

items = items.sort(comparer);

}

}

class InsertionSortStrategy<T> implements ISortStrategy<T> {

public sort<T>(items: Array<T>, comparer: (left: T, right: T) => number) {

var len = items.length;

if (len < 2) {

return;

}

var j = 0;

var previous: T, left: T = items[0];

for (var i = 1; i < len; i++) {

previous = left;

left = items[i];

j = (comparer(previous, left) <= 0) ? ++j : 0;

while (j < i) {

var right = items[j];

if (comparer(left, right) < 0) {

items.splice(i, 1);

items.splice(j, 0, left);

break;

} else {

j++;

}

}

}

}

}

class MergeSortStrategy<T> implements ISortStrategy<T> {

public sort<T>(items: Array<T>, comparer: (left: T, right: T) => number) {

if (items.length < 2) {

return items;

}

var sortedArrays: Array<Array<T>> = items.map((value) => { return [value]; });

var leftArray: Array<T>, rightArray: Array<T>;

while (sortedArrays.length > 1) {

leftArray = sortedArrays.shift();

rightArray = sortedArrays.shift();

var result = new Array<T>();

var left = leftArray.shift();

var right = rightArray.shift();

do {

if (comparer(left, right) <= 0) {

result.push(left);

left = leftArray.shift();

} else {

result.push(right);

right = rightArray.shift();

}

} while (left !== undefined && right !== undefined)

while (left !== undefined) {

result.push(left);

left = leftArray.shift();

}

while (right !== undefined) {

result.push(right);

right = rightArray.shift();

}

sortedArrays.push(result);

}

var sortedItems = sortedArrays[0];

for (var i = 0; i < items.length; i++) {

items[i] = sortedItems.shift();

}

}

}

class SortableList<T>{

private \_items: Array<T> = new Array<T>();

constructor(items: Array<T>) {

this.sortStrategy = new StandardSortStrategy<T>();

this.\_items = this.\_items.concat(items);

}

public add(items: Array<T>) {

this.\_items = this.\_items.concat(items);

}

public get items(): Array<T> {

return this.\_items;

}

public sortStrategy: ISortStrategy<T>;

protected comparer: (left: T, right: T) => number = (left: T, right: T) => { return (left < right) ? -1 : 1; };

public sort() {

this.sortStrategy.sort(this.\_items, this.comparer);

}

}

class SortableNumberList extends SortableList<number>{

constructor(items: Array<number>) {

super(items);

}

}

window.addEventListener("load", function () {

var randomNumbers = new Array<number>();

var lenght = 50;

while (lenght--) {

randomNumbers.push(Math.round(Math.random() \* 1000));

}

var sortableList1 = new SortableNumberList(randomNumbers);

sortableList1.sort();

Output.WriteLine("Sorted using the standard sort strategie:");

Output.WriteLine(sortableList1.items.join("|"));

var sortableList2 = new SortableNumberList(randomNumbers);

sortableList2.sortStrategy = new MergeSortStrategy<number>();

sortableList2.sort();

Output.WriteLine("Sorted using the merge sort strategie:");

Output.WriteLine(sortableList2.items.join("|"));

var sortableList3 = new SortableNumberList(randomNumbers);

sortableList3.sortStrategy = new InsertionSortStrategy<number>();

sortableList3.sort();

Output.WriteLine("Sorted using the insertion sort strategie:");

Output.WriteLine(sortableList3.items.join("|"));

});

}

Output

Sorted using the standard sort strategy:  
6|15|17|43|169|213|219|226|237|242|258|261|281|301|333|345|354|364|384|398|431|434|458|482|485|508|533|576|589|616|627|655|659|676|682|684|686|687|723|733|803|822|858|869|877|889|901|931|934|946

Sorted using the merge sort strategy:  
6|15|17|43|169|213|219|226|237|242|258|261|281|301|333|345|354|364|384|398|431|434|458|482|485|508|533|576|589|616|627|655|659|676|682|684|686|687|723|733|803|822|858|869|877|889|901|931|934|946

Sorted using the insertion sort strategy:  
6|15|17|43|169|213|219|226|237|242|258|261|281|301|333|345|354|364|384|398|431|434|458|482|485|508|533|576|589|616|627|655|659|676|682|684|686|687|723|733|803|822|858|869|877|889|901|931|934|946

# Builder Pattern

## Definition

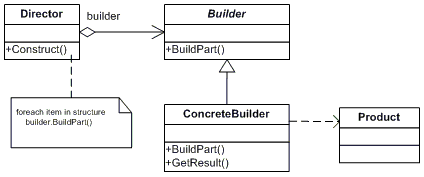
Separate the construction of a complex object from its representation so that the same construction process can create different representations.

## Explanation

The builder pattern is an object creation software design pattern. Unlike the abstract factory pattern and the factory method pattern whose intention is to enable polymorphism, the intention of the builder pattern is to find a solution to the telescoping constructor anti-pattern. The telescoping constructor anti-pattern occurs when the increase of object constructor parameter combination leads to an exponential list of constructors. Instead of using numerous constructors, the builder pattern uses another object, a builder, that receives each initialization parameter step by step and then returns the resulting constructed object at once.

The builder pattern has another benefit. It can be used for objects that contain flat data (html code, SQL query, X.509 certificate…), that is to say, data that can’t be easily edited. This type of data cannot be edited step by step and must be edited at once. The best way to construct such an object is to use a builder class.

Builder often builds a Composite. Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. Sometimes creational patterns are complementary: Builder can use one of the other patterns to implement which components are built. Builders are good candidates for a fluent interface.



## TypeScript Code

module Builder {

class Shop {

private \_vehicleBuilder: VehicleBuilder;

public Construct(vehicleBuilder: VehicleBuilder): void {

this.\_vehicleBuilder = vehicleBuilder;

this.\_vehicleBuilder.BuildFrame();

this.\_vehicleBuilder.BuildEngine();

this.\_vehicleBuilder.BuildWheels();

this.\_vehicleBuilder.BuildDoors();

}

public ShowVehicle(): void {

this.\_vehicleBuilder.vehicle.display();

}

}

class VehicleBuilder {

private \_vehicle: Vehicle = null;

constructor(public vehicleType: VehicleType) {

this.\_vehicle = new Vehicle(vehicleType);

}

public get vehicle(): Vehicle {

return this.\_vehicle;

}

public BuildFrame(): void {

throw new Error("Not implemented.");

}

public BuildEngine(): void {

throw new Error("Not implemented.");

}

public BuildWheels(): void {

throw new Error("Not implemented.");

}

public BuildDoors(): void {

throw new Error("Not implemented.");

}

}

class CarBuilder extends VehicleBuilder {

constructor() {

super(VehicleType.Car);

}

public BuildFrame(): void {

this.vehicle.parts[PartType.Frame] = "Car Frame";

}

public BuildEngine(): void {

this.vehicle.parts[PartType.Engine] = "2500 cc";

}

public BuildWheels(): void {

this.vehicle.parts[PartType.Wheel] = "4";

}

public BuildDoors(): void {

this.vehicle.parts[PartType.Door] = "4";

}

}

class MotorCycleBuilder extends VehicleBuilder {

constructor() {

super(VehicleType.MotorCycle);

}

public BuildFrame(): void {

this.vehicle.parts[PartType.Frame] = "MotorCycle Frame";

}

public BuildEngine(): void {

this.vehicle.parts[PartType.Engine] = "500 cc";

}

public BuildWheels(): void {

this.vehicle.parts[PartType.Wheel] = "2";

}

public BuildDoors(): void {

this.vehicle.parts[PartType.Door] = "0";

}

}

class ScooterBuilder extends VehicleBuilder {

constructor() {

super(VehicleType.Scooter);

}

public BuildFrame(): void {

this.vehicle.parts[PartType.Frame] = "Scooter Frame";

}

public BuildEngine(): void {

this.vehicle.parts[PartType.Engine] = "50 cc";

}

public BuildWheels(): void {

this.vehicle.parts[PartType.Wheel] = "2";

}

public BuildDoors(): void {

this.vehicle.parts[PartType.Door] = "0";

}

}

class Vehicle {

constructor(public vehicleType: VehicleType) {

this.vehicleType = vehicleType;

}

private \_parts: {} = {};

public get parts(): {} {

return this.\_parts;

}

public display() {

Output.WriteLine("---------------------------");

Output.WriteLine("Vehicle Type : " + VehicleType[this.vehicleType]);

Output.WriteLine("Frame :" + this.parts[PartType.Frame]);

Output.WriteLine("Engine :" + this.parts[PartType.Engine]);

Output.WriteLine("#Wheels :" + this.parts[PartType.Wheel]);

Output.WriteLine("#Doors :" + this.parts[PartType.Door]);

Output.WriteLine("---------------------------");

}

}

enum VehicleType {

Car,

Scooter,

MotorCycle

}

enum PartType {

Frame,

Engine,

Wheel,

Door

}

window.addEventListener("load", function () {

var shop = new Shop();

shop.Construct(new ScooterBuilder());

shop.ShowVehicle();

shop.Construct(new CarBuilder());

shop.ShowVehicle();

shop.Construct(new MotorCycleBuilder());

shop.ShowVehicle();

});

}

## Output

|  |  |  |
| --- | --- | --- |
| ---------------------------  Vehicle Type : Scooter Frame :Scooter Frame Engine :50 cc #Wheels :2 #Doors :0 --------------------------- | --------------------------- Vehicle Type : Car Frame :Car Frame Engine :2500 cc #Wheels :4 #Doors :4 --------------------------- | --------------------------- Vehicle Type : MotorCycle Frame :MotorCycle Frame Engine :500 cc #Wheels :2 #Doors :0 --------------------------- |

# Singleton Pattern

## Definition

Ensure a class has only one instance and provide a global point of access to it.

## Explanation

In software engineering, the singleton pattern is a design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects. The term comes from the mathematical concept of a singleton.

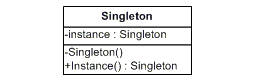
There is criticism of the use of the singleton pattern, as some consider it an anti-pattern, judging that it is overused, introduces unnecessary restrictions in situations where a sole instance of a class is not actually required, and introduces global state into an application.

Common use:

* The Abstract Factory, Builder, and Prototype patterns can use Singletons in their implementation.
* Facade objects are often singletons because only one Facade object is required.
* State objects are often singletons.
* Singletons are often preferred to global variables because: They do not pollute the global namespace (or, in languages with namespaces, their containing namespace) with unnecessary variables.
* They permit lazy allocation and initialization, whereas global variables in many languages will always consume resources.

Implementation of a singleton pattern must satisfy the single instance and global access principles. It requires a mechanism to access the singleton class member without creating a class object and a mechanism to persist the value of class members among class objects. The singleton pattern is implemented by creating a class with a method that creates a new instance of the class if one does not exist. If an instance already exists, it simply returns a reference to that object. To make sure that the object cannot be instantiated any other way, the constructor is made private. Note the distinction between a simple static instance of a class and a singleton: although a singleton can be implemented as a static instance, it can also be lazily constructed, requiring no memory or resources until needed.

The singleton pattern must be carefully constructed in multi-threaded applications. If two threads are to execute the creation method at the same time when a singleton does not yet exist, they both must check for an instance of the singleton and then only one should create the new one. If the programming language has concurrent processing capabilities the method should be constructed to execute as a mutually exclusive operation. The classic solution to this problem is to use mutual exclusion on the class that indicates that the object is being instantiated.



## TypeScript Code

module Singleton {

class Singleton {

public counter: number = 0;

private static \_instance: Singleton = null;

constructor() {

if (Singleton.\_instance) {

throw new Error("Error: Instantiation failed: Use Singleton.current instead of new.");

}

Singleton.\_instance = this;

}

public static get current(): Singleton {

if (Singleton.\_instance === null) {

Singleton.\_instance = new Singleton();

}

return Singleton.\_instance;

}

public display() {

Output.WriteLine("The Singleton counter has a value of:" + this.counter);

}

}

window.addEventListener("load", function () {

var singleton1 = Singleton.current;

singleton1.display();

var singleton2 = Singleton.current;

singleton2.counter++;

singleton1.display();

try {

var singleton3 = new Singleton();

} catch (error) {

Output.WriteLine(error.message);

}

singleton2.counter++;

singleton1.display();

});

}

## Output

The Singleton counter has a value of:0  
The Singleton counter has a value of:1  
Error: Instantiation failed: Use Singleton.current instead of new.  
The Singleton counter has a value of:2

# Adapter Pattern

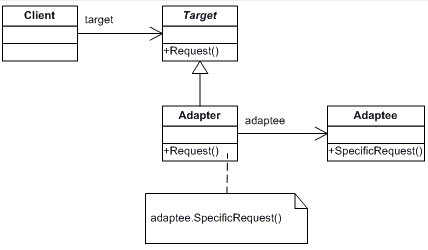
Definition

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

Explanation

An adapter helps two incompatible interfaces to work together. This is the real world definition for an adapter. Interfaces may be incompatible but the inner functionality should suit the need. The Adapter design pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients.

The adapter pattern is useful in situations where an already existing class provides some or all of the services you need but does not use the interface you need. A good real life example is an adapter that converts the interface of a Document Object Model of an XML document into a tree structure that can be displayed.



TypeScript Code

module ThirdpartyLib {

export class StringNewsServer {

public userName: string;

public passWord: string;

public getString(): string {

/\* validate userName and passWord then \*/

return "StringNewsServer.newsItem1;StringNewsServer.newsItem2;StringNewsServer.newsItem3";

}

}

export class ArrayNewsServer {

public url: string;

public getArray(): Array {

/\* use url to fetch data then \*/

return ["ArrayNewsServer.newsItem1", "ArrayNewsServer.newsItem2", "ArrayNewsServer.newsItem3"];

}

}

}

module News {

export interface INewsServerInterface {

getNews(): Array;

}

export class NewsLoader {

public Load(server: INewsServerInterface) {

var news = server.getNews();

news.forEach((value: string) => {

Output.WriteLine(value);

});

}

}

export class StringNewsServerAdapter implements INewsServerInterface {

private newsServer: ThirdpartyLib.StringNewsServer;

constructor() {

this.newsServer = new ThirdpartyLib.StringNewsServer();

this.newsServer.userName = "userName";

this.newsServer.passWord = "passWord";

}

public getNews() {

var items = this.newsServer.getString();

return items.split(";");

}

}

export class ArrayNewsServerAdapter implements INewsServerInterface {

private newsServer: ThirdpartyLib.ArrayNewsServer;

constructor() {

this.newsServer = new ThirdpartyLib.ArrayNewsServer();

this.newsServer.url = "http://mynews.com";

}

public getNews() {

return this.newsServer.getArray();

}

}

window.addEventListener("load", function () {

var newsLoader = new NewsLoader();

var arrayNewsServer = new ArrayNewsServerAdapter();

newsLoader.Load(arrayNewsServer);

var stringNewsServer = new StringNewsServerAdapter();

newsLoader.Load(stringNewsServer);

});

}

Output

ArrayNewsServer.newsItem1  
ArrayNewsServer.newsItem2  
ArrayNewsServer.newsItem3  
StringNewsServer.newsItem1  
StringNewsServer.newsItem2  
StringNewsServer.newsItem3

# Façade Pattern

## Definition

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

## Explanation

The facade pattern (or façade pattern) is a software design pattern commonly used with object-oriented programming. The name is by analogy to an architectural facade.

A facade is an object that provides a simplified interface to a larger body of code, such as a class library. A facade can:

* make a software library easier to use, understand and test, since the facade has convenient methods for common tasks;
* make the library more readable, for the same reason;
* reduce dependencies of outside code on the inner workings of a library, since most code uses the facade, thus allowing more flexibility in developing the system;
* wrap a poorly designed collection of APIs with a single well-designed API (as per task needs).

The Facade design pattern is often used when a system is very complex or difficult to understand because the system has a large number of interdependent classes or its source code is unavailable. This pattern hides the complexities of the larger system and provides a simpler interface to the client. It typically involves a single wrapper class which contains a set of members required by client. These members access the system on behalf of the facade client and hide the implementation details.

A Facade is used when an easier or simpler interface to an underlying object is desired. Alternatively, an adapter can be used when the wrapper must respect a particular interface and must support polymorphic behavior. A decorator makes it possible to add or alter behavior of an interface at run-time.

**Adapter**

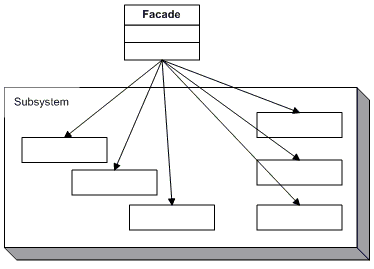
Converts one interface to another so that it matches what the client is expecting.

**Decorator**

Dynamically adds responsibility to the interface by wrapping the original code.

**Facade**

Provides a simplified interface.



## TypeScript Code

module Facade {

class Customer {

constructor(public name: string) {

}

}

class Bank {

public static hasSufficientSavings(customer: Customer, amount: number): boolean {

Output.WriteLine("Check bank for " + customer.name);

return true;

}

}

class Credit {

public static hasGoodCredit(customer: Customer): boolean {

Output.WriteLine("Check credit for " + customer.name);

return true;

}

}

class Loan {

public static hasNoBadLoans(customer: Customer): boolean {

Output.WriteLine("Check loans for " + customer.name);

return true;

}

}

class Mortgage {

public static IsEligible(customer: Customer, amount: number): boolean {

Output.WriteLine(customer.name + " applies for a loan of " + amount + " dollar.");

var eligible: boolean = true;

if (!Bank.hasSufficientSavings(customer, amount)) {

eligible = false;

}

else if (!Loan.hasNoBadLoans(customer)) {

eligible = false;

}

else if (!Credit.hasGoodCredit(customer)) {

eligible = false;

}

return eligible;

}

}

window.addEventListener("load", function () {

var customer = new Customer("Wesley Bakker");

var elegible: boolean = Mortgage.IsEligible(customer, 1000000);

Output.WriteLine(customer.name + " has been " + (elegible ? "approved" : "rejected"));

});

}

## Output

Wesley Bakker applies for a loan of 1000000 dollar.  
Check bank for Wesley Bakker  
Check loans for Wesley Bakker  
Check credit for Wesley Bakker  
Wesley Bakker has been approved

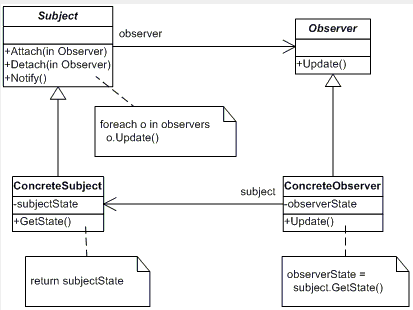
Observer Pattern

Definition

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Explanation

The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems. The Observer pattern is also a key part in the familiar model–view–controller (MVC) architectural pattern. The observer pattern is implemented in numerous programming libraries and systems, including almost all GUI toolkits.

The observer pattern can cause memory leaks, known as the lapsed listener problem, because in basic implementation it requires both explicit registration and explicit deregistration, as in the dispose pattern, because the subject holds strong references to the observers, keeping them alive. This can be prevented by the subject holding weak references to the observers.

TypeScript Code

module Observer {

export interface IEvent<T> {

add(eventHandler: (sender: any, eventArgs: T) => void): void;

remove(eventHandler: (sender: any, eventArgs: T) => void): void;

}

export class Event<T> implements IEvent<T> {

private \_eventHandlers: Array<(sender: any, eventArgs: T) => void>;

constructor() {

this.\_eventHandlers = new Array<(sender: any, eventArgs: T) => void>();

}

public add(eventHandler: (sender: any, eventArgs: T) => void): void {

if (this.\_eventHandlers.indexOf(eventHandler) === -1) {

this.\_eventHandlers.push(eventHandler);

}

}

public remove(eventHandler: (sender: any, eventArgs: T) => void): void {

var i = this.\_eventHandlers.indexOf(eventHandler);

if (i !== -1) {

this.\_eventHandlers.splice(i, 1);

}

}

public raise(sender: any, e: T): void {

for (var i = 0, j = this.\_eventHandlers.length; i < j; i++) {

try {

this.\_eventHandlers[i](sender, e);

}

catch(ex) { /\*Errors in eventhandlers should not prevent other handlers to be called.\*/ }

}

}

}

export class PropertyChangedEventArgs {

private \_propertyName: string;

private \_before: any;

private \_after: any;

constructor(propertyName: string, before?: any, after?: any) {

this.\_propertyName = propertyName;

this.\_before = before;

this.\_after = after;

}

public get propertyName(): string {

return this.\_propertyName;

}

public get before(): any {

return this.\_before;

}

public get after(): any {

return this.\_after;

}

}

export interface INotifyPropertyChanged {

propertyChanged: IEvent<PropertyChangedEventArgs>;

}

export class ObservableObject implements INotifyPropertyChanged {

private \_propertyChanged: Event<PropertyChangedEventArgs>;

constructor() {

this.\_propertyChanged = new Event<PropertyChangedEventArgs>();

}

public get propertyChanged(): IEvent<PropertyChangedEventArgs> {

return this.\_propertyChanged;

}

protected onPropertyChanged(eventArgs: PropertyChangedEventArgs) {

this.\_propertyChanged.raise(this, eventArgs);

}

}

export class Employee extends ObservableObject {

private \_name: string;

private \_role: string;

constructor(name: string, role: string) {

super();

this.\_name = name;

this.\_role = role;

}

public get name(): string {

return this.\_name;

}

public set name(value: string) {

if (this.\_name !== value) {

var eventArgs = new PropertyChangedEventArgs("name", this.\_name, value);

this.\_name = value;

this.onPropertyChanged(eventArgs);

}

}

public get role(): string {

return this.\_role;

}

public set role(value: string) {

if (this.\_role !== value) {

var eventArgs = new PropertyChangedEventArgs("role", this.\_role, value);

this.\_role = value;

this.onPropertyChanged(eventArgs);

}

}

private \_isDirty: boolean = false;

public get isDirty(): boolean {

return this.\_isDirty;

}

protected onPropertyChanged(eventArgs: PropertyChangedEventArgs) {

this.\_isDirty = true;

super.onPropertyChanged(eventArgs);

}

}

window.addEventListener("load", function () {

var employee = new Employee("Wesley", "Developer");

employee.propertyChanged.add(function (sender: any, e: PropertyChangedEventArgs) {

Output.WriteLine("The property '" + e.propertyName + "' " +

"changed from '" + e.before + "' " +

"to '" + e.after + "'.");

});

employee.name = "Norbert";

employee.role = "Lead Developer";

employee.name = "Jos";

employee.role = "Junior Developer";

});

}

Output

The property 'name' changed from 'Wesley' to 'Norbert'.  
The property 'role' changed from 'Developer' to 'Lead Developer'.  
The property 'name' changed from 'Norbert' to 'Jos'.  
The property 'role' changed from 'Lead Developer' to 'Junior Developer'.

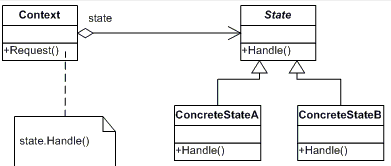
State Pattern

Definition

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Explanation

This pattern is used in computer programming to encapsulate varying behavior for the same routine based on an object’s state object. This can be a cleaner way for an object to change its behavior at runtime without resorting to large monolithic conditional statements.



TypeScript Code

module State {

interface IMusicPlayerState {

start();

stop();

pauze();

}

class StoppedState implements IMusicPlayerState {

constructor(private player: MusicPlayer) {

}

public start() {

Output.WriteLine("Start playing.");

this.player.\_state = new StartedState(this.player);

}

public stop() {

Output.WriteLine("Already stopped!");

}

public pauze() {

Output.WriteLine("I'm not started!");

}

}

class StartedState implements IMusicPlayerState {

constructor(private player: MusicPlayer) {

}

public start() {

Output.WriteLine("Already started!");

}

public stop() {

Output.WriteLine("Stopped playing.");

this.player.\_state = new StoppedState(this.player);

}

public pauze() {

Output.WriteLine("Pauzed playing.");

this.player.\_state = new PauzedState(this.player);

}

}

class PauzedState implements IMusicPlayerState {

constructor(private player: MusicPlayer) {

}

public start() {

Output.WriteLine("Continue playing.");

this.player.\_state = new StartedState(this.player);

}

public stop() {

Output.WriteLine("Stop playing!");

this.player.\_state = new StoppedState(this.player);

}

public pauze() {

Output.WriteLine("Already pauzed.");

}

}

export class MusicPlayer {

public \_state: IMusicPlayerState;

constructor() {

this.\_state = new StoppedState(this);

}

public start() {

this.\_state.start();

}

public stop() {

this.\_state.stop();

}

public pauze() {

this.\_state.pauze();

}

}

}

window.addEventListener("load", function () {

var musicPlayer = new State.MusicPlayer();

musicPlayer.stop();

musicPlayer.start();

musicPlayer.start();

musicPlayer.pauze();

musicPlayer.start();

musicPlayer.stop();

});

Output

Already stopped!  
Start playing.  
Already started!   
Pauzed playing.  
Continue playing.  
Stopped playing.

Factory Method Pattern

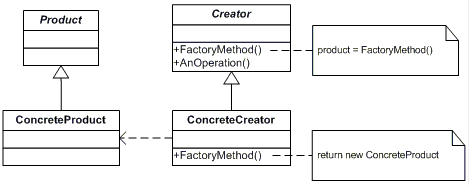
Definition

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Explanation

Creating an object often requires complex processes not appropriate to include within a composing object. The object’s creation may lead to a significant duplication of code, may require information not accessible to the composing object, may not provide a sufficient level of abstraction, or may otherwise not be part of the composing object’s concerns. The factory method design pattern handles these problems by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created.

The factory method pattern relies on inheritance, as object creation is delegated to subclasses that implement the factory method to create objects.



TypeScript Code

module FactoryMethod {

class Page {

private \_type: string;

constructor(type: string) {

this.\_type = type;

}

public get Type(): string {

return this.\_type;

}

}

class SkillsPage extends Page {

constructor() {

super("SkillsPage");

}

}

class EducationPage extends Page {

constructor() {

super("EducationPage");

}

}

class ExperiencePage extends Page {

constructor() {

super("ExperiencePage");

}

}

class IntroductionPage extends Page {

constructor() {

super("IntroductionPage");

}

}

class ResultsPage extends Page {

constructor() {

super("ResultsPage");

}

}

class ConclusionPage extends Page {

constructor() {

super("ConclusionPage");

}

}

class SummaryPage extends Page {

constructor() {

super("SummaryPage");

}

}

class BibliographyPage extends Page {

constructor() {

super("BibliographyPage");

}

}

class PageFactory {

private \_pages: Array = new Array();

private \_type: string;

constructor(type: string) {

this.\_type = type;

this.createPages();

}

public get type(): string {

return this.\_type;

}

public get pages(): Array {

return this.\_pages;

}

public createPages(): void {

throw new Error("Method not implemented");

}

}

class Resume extends PageFactory {

constructor() {

super("Resume");

}

public createPages(): void {

this.pages.push(new SkillsPage());

this.pages.push(new EducationPage());

this.pages.push(new ExperiencePage());

}

}

class Report extends PageFactory {

constructor() {

super("Report");

}

public createPages(): void {

this.pages.push(new IntroductionPage());

this.pages.push(new ResultsPage());

this.pages.push(new ConclusionPage());

this.pages.push(new SummaryPage());

this.pages.push(new BibliographyPage());

}

}

window.addEventListener("load", function () {

var factories: Array = new Array(new Resume(), new Report());

factories.forEach((factory: PageFactory) => {

Output.WriteLine("The " + factory.type + " contains the following pages:");

factory.pages.forEach((page: Page) => {

Output.WriteLine("--" + page.Type);

});

});

});

}

Output

The Resume contains the following pages:  
--SkillsPage  
--EducationPage  
--ExperiencePage  
The Report contains the following pages:  
--IntroductionPage  
--ResultsPage  
--ConclusionPage  
--SummaryPage  
--BibliographyPage

# ****Object Pool**** Design Pattern

### **Intent**

Object pooling can offer a significant performance boost; it is most effective in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instantiations in use at any one time is low.

### **Problem**

Object pools (otherwise known as resource pools) are used to manage the object caching. A client with access to a Object pool can avoid creating a new Objects by simply asking the pool for one that has already been instantiated instead. Generally the pool will be a growing pool, i.e. the pool itself will create new objects if the pool is empty, or we can have a pool, which restricts the number of objects created.

It is desirable to keep all Reusable objects that are not currently in use in the same object pool so that they can be managed by one coherent policy. To achieve this, the Reusable Pool class is designed to be a singleton class.

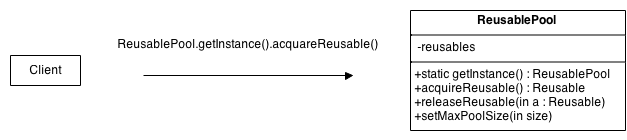
### Discussion

The Object Pool lets others "check out" objects from its pool, when those objects are no longer needed by their processes, they are returned to the pool in order to be reused.

However, we don't want a process to have to wait for a particular object to be released, so the Object Pool also instantiates new objects as they are required, but must also implement a facility to clean up unused objects periodically.

### **Structure**

The general idea for the Connection Pool pattern is that if instances of a class can be reused, you avoid creating instances of the class by reusing them.



* **Reusable** - Instances of classes in this role collaborate with other objects for a limited amount of time, then they are no longer needed for that collaboration.
* **Client** - Instances of classes in this role use Reusable objects.
* **ReusablePool** - Instances of classes in this role manage Reusable objects for use by Client objects.

Usually, it is desirable to keep all Reusable objects that are not currently in use in the same object pool so that they can be managed by one coherent policy. To achieve this, the ReusablePool class is designed to be a singleton class. Its constructor(s) are private, which forces other classes to call its getInstance method to get the one instance of the ReusablePool class.

A Client object calls a ReusablePool object's acquireReusable method when it needs a Reusable object. A ReusablePool object maintains a collection of Reusable objects. It uses the collection of Reusable objects to contain a pool of Reusable objects that are not currently in use.

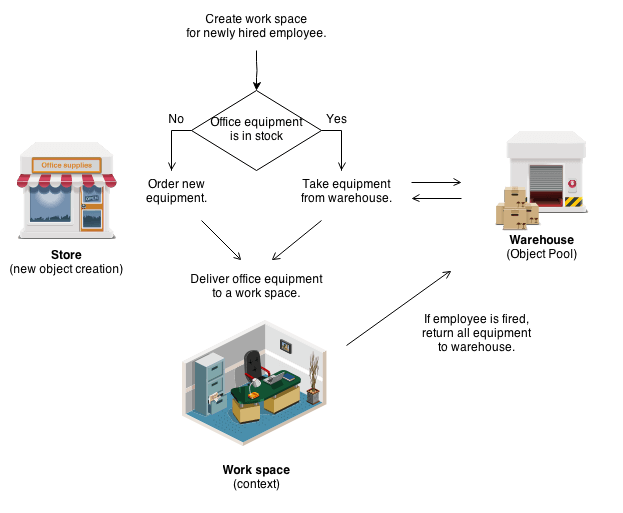
If there are any Reusable objects in the pool when the acquireReusable method is called, it removes a Reusable object from the pool and returns it. If the pool is empty, then the acquireReusable method creates a Reusable object if it can. If the acquireReusable method cannot create a new Reusable object, then it waits until a Reusable object is returned to the collection.

Client objects pass a Reusable object to a ReusablePool object's releaseReusable method when they are finished with the object. The releaseReusable method returns a Reusable object to the pool of Reusable objects that are not in use.

In many applications of the Object Pool pattern, there are reasons for limiting the total number of Reusable objects that may exist. In such cases, the ReusablePool object that creates Reusable objects is responsible for not creating more than a specified maximum number of Reusable objects. If ReusablePool objects are responsible for limiting the number of objects they will create, then the ReusablePool class will have a method for specifying the maximum number of objects to be created. That method is indicated in the above diagram as setMaxPoolSize.

### **Example**

Object pool pattern is similar to an office warehouse. When a new employee is hired, office manager has to prepare a work space for him. She figures whether or not there's a spare equipment in the office warehouse. If so, she uses it. If not, she places an order to purchase new equipment from Amazon. In case if an employee is fired, his equipment is moved to warehouse, where it could be taken when new work place will be needed.



### **Check** **list**

1. Create ObjectPool class with private array of Objects inside
2. Create acquire and release methods in ObjectPool class
3. Make sure that your ObjectPool is Singleton

### **Rules** **of** **thumb**

* The Factory Method pattern can be used to encapsulate the creation logic for objects. However, it does not manage them after their creation, the object pool pattern keeps track of the objects it creates.
* Object Pools are usually implemented as Singletons.