Service locator pattern

According to Wikipedia, The **service locator pattern** is a [design pattern](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) used in software development to encapsulate the processes involved in obtaining a service with a strong [abstraction layer](https://en.wikipedia.org/wiki/Abstraction_layer). This pattern uses a central [registry](https://en.wikipedia.org/wiki/Domain_name_registry) known as the "service locator", which on request returns the information necessary to perform a certain task.

<https://www.tutorialspoint.com/design_pattern/service_locator_pattern.htm>

The service locator design pattern is used when we want to locate various services using JNDI lookup. Considering high cost of looking up JNDI for a service, Service Locator pattern makes use of caching technique. For the first time a service is required, Service Locator looks up in JNDI and caches the service object. Further lookup or same service via Service Locator is done in its cache which improves the performance of application to great extent. Following are the entities of this type of design pattern.

* **Service** - Actual Service which will process the request. Reference of such service is to be looked upon in JNDI server.
* **Context / Initial Context** - JNDI Context carries the reference to service used for lookup purpose.
* **Service Locator** - Service Locator is a single point of contact to get services by JNDI lookup caching the services.
* **Cache** - Cache to store references of services to reuse them
* **Client** - Client is the object that invokes the services via ServiceLocator.

# Implementation

## Step 1

Create Service interface.

*Service.java*

public interface Service {

public String getName();

public void execute();

}

## Step 2

Create concrete services.

*Service1.java*

public class Service1 implements Service {

public void execute(){

System.out.println("Executing Service1");

}

@Override

public String getName() {

return "Service1";

}

}

*Service2.java*

public class Service2 implements Service {

public void execute(){

System.out.println("Executing Service2");

}

@Override

public String getName() {

return "Service2";

}

}

## Step 3

Create InitialContext for JNDI lookup

*InitialContext.java*

public class InitialContext {

public Object lookup(String jndiName){

if(jndiName.equalsIgnoreCase("SERVICE1")){

System.out.println("Looking up and creating a new Service1 object");

return new Service1();

}

else if (jndiName.equalsIgnoreCase("SERVICE2")){

System.out.println("Looking up and creating a new Service2 object");

return new Service2();

}

return null;

}

}

## Step 4

Create Cache

*Cache.java*

import java.util.ArrayList;

import java.util.List;

public class Cache {

private List<Service> services;

public Cache(){

services = new ArrayList<Service>();

}

public Service getService(String serviceName){

for (Service service : services) {

if(service.getName().equalsIgnoreCase(serviceName)){

System.out.println("Returning cached " + serviceName + " object");

return service;

}

}

return null;

}

public void addService(Service newService){

boolean exists = false;

for (Service service : services) {

if(service.getName().equalsIgnoreCase(newService.getName())){

exists = true;

}

}

if(!exists){

services.add(newService);

}

}

}

## Step 5

Create Service Locator

*ServiceLocator.java*

public class ServiceLocator {

private static Cache cache;

static {

cache = new Cache();

}

public static Service getService(String jndiName){

Service service = cache.getService(jndiName);

if(service != null){

return service;

}

InitialContext context = new InitialContext();

Service service1 = (Service)context.lookup(jndiName);

cache.addService(service1);

return service1;

}

}

## Step 6

Use the *ServiceLocator* to demonstrate Service Locator Design Pattern.

*ServiceLocatorPatternDemo.java*

public class ServiceLocatorPatternDemo {

public static void main(String[] args) {

Service service = ServiceLocator.getService("Service1");

service.execute();

service = ServiceLocator.getService("Service2");

service.execute();

service = ServiceLocator.getService("Service1");

service.execute();

service = ServiceLocator.getService("Service2");

service.execute();

}

}

## Step 7

Verify the output.

Looking up and creating a new Service1 object

Executing Service1

Looking up and creating a new Service2 object

Executing Service2

Returning cached Service1 object

Executing Service1

Returning cached Service2 object

Executing Service2

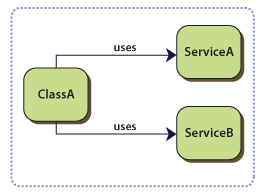
<https://msdn.microsoft.com/en-us/library/ff648968.aspx>

# The Service Locator Pattern

# Context

You have classes with dependencies on services whose concrete types are specified at compile time. In the following example, **ClassA** has compile time dependencies on **ServiceA** and **ServiceB**. The following diagram illustrates this.

Classes with dependencies on services



This situation has the following drawbacks:

* To replace or update the dependencies, you must change your classes' source code and recompile the solution.
* The concrete implementation of the dependencies must be available at compile time.
* Your classes are difficult to test in isolation because they have a direct reference to their dependencies. This means that these dependencies cannot be replaced with stubs or mock objects.
* Your classes contain repetitive code for creating, locating, and managing their dependencies.

The next section describes how to address these issues.

# Objectives

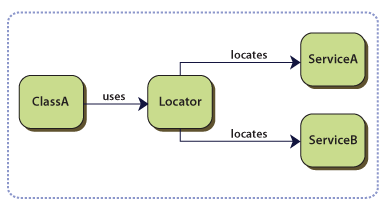
Use the Service Locator pattern to achieve any of the following objectives:

* You want to decouple your classes from their dependencies so that these dependencies can be replaced or updated with little or no change to the classes.
* You want to write logic that depends on classes whose concrete implementation is not known at compile time.
* You want to be able to test your classes in isolation, without the dependencies.
* You do not want the logic that locates and manages the dependencies to be in your classes.
* You want to divide your application into loosely coupled modules that can be independently developed, tested, versioned, and deployed.

# Solution

Create a service locator that contains references to the services and that encapsulates the logic that locates them. In your classes, use the service locator to obtain service instances. The following diagram illustrates how classes use a service locator.

How classes use a service locator



The Service Locator pattern does not describe how to instantiate the services. It describes a way to register services and locate them. Typically, the Service Locator pattern is combined with the Factory pattern and/or the Dependency Injection pattern. This combination allows a service locator to create instances of services.

|  |
| --- |
| **Ff648968.note(en-us,PandP.10).gifNote:** |
| A service locator should be able to locate a service without knowing its concrete type. For example, it might use a string key or a service interface type. This allows you to replace the concrete implementation of the dependency without modifying the classes. |

# Implementation Details

The SharePoint Guidance Library offers an implementation of the Service Locator pattern. The **SharePointServiceLocator** class provides access to a singleton **IServiceLocator** instance and manages that instance. The **SharePointServiceLocator** class includes a default implementation of the interface. This is the **ActivatingServiceLocator** class. This class can both create and locate services.

The Partner Portal application shows how to use the Service Locator to register and locate services such as repositories, logging services, and configuration management services. For more information, see [The SharePoint Service Locator](https://msdn.microsoft.com/en-us/library/ff647574.aspx).

# Considerations

Consider the following points before you use the Service Locator pattern:

* There are more solution elements to manage.
* You must write additional code that adds service references to the service locator before your objects can use it.
* Your classes have a dependency on the service locator.
* The source code is more complex and difficult to understand.
* You can use configuration data to define run-time relationships.
* You must provide implementations of the services. Because the Service Locator pattern decouples service consumers from service providers, it might be necessary to provide additional logic. This logic ensures that the service providers are installed and registered before service consumers try to locate them.

<http://blog.ploeh.dk/2010/11/01/PatternRecognitionAbstractFactoryorServiceLocator/>

# Pattern Recognition: Abstract Factory or Service Locator? by Mark Seemann

It's easy to confuse the [Abstract Factory](http://en.wikipedia.org/wiki/Abstract_factory_pattern) pattern with the [Service Locator](http://blog.ploeh.dk/2010/02/03/ServiceLocatorisanAnti-Pattern) anti-pattern - particularly so when generics or contextual information is involved. However, it's really easy to distinguish between there two, and here's how!

Here are both (anti-)patterns in condensed form opposite each other:

|  |  |
| --- | --- |
| Abstract Factory | Service Locator |
| public interface IFactory<T> {     T Create(object context); } | public interface IServiceLocator {     T Create<T>(object context); } |

For these examples I chose to demonstrate both as generic interfaces that take some kind of contextual information (*context*) as input.

In this example the context can be any object, but we could also have considered a more strongly typed *context* parameter. Other variations include more than one method parameter, or, in the degenerate case, no parameters at all.

Both interfaces have a simple Create method that returns the generic type T, so it's easy to confuse the two. However, even for generic types, it's easy to tell one from the other:

An **Abstract Factory** is a generic type, and the return type of the Create method is determined by the type of the factory itself. In other words, a constructed type can only return instances of a single type.

A **Service Locator**, on the other hand, is a non-generic interface with a generic method. The Create method of a single Service Locator can return instances of an infinite number of types.

Even simpler:

An **Abstract Factory** is a generic type with a non-generic Create method; a **Service Locator** is a non-generic type with a generic Create method.

The name of the method, the number of parameters, and other circumstances may vary. The types may not be generic, or may be base classes instead of interfaces, but at the heart of it, the question is whether you can ask for an arbitrary type from the service, or only a single, static type.

<http://paul-m-jones.com/archives/4800>

The differences between factories, registries, and containers (both service locators and dependency injection containers) can be subtle and hard to grasp at first. Here’s how I keep them straight:

* a Factory creates and returns an object, but does not retain the created object for future use;
* a Registry retains an object instance by name for repeated use, but does not create that object;
* a Service Locator contains a named object (a “service”) and creates it if it  
  has a definition for that service; the creation logic might itself be a Factory, and the retention logic might itself be a Registry.

By way of example, I have put together an [example service locator implementation](https://github.com/pmjones/Pmjones/blob/master/service-locator.php)along with an embedded test case. It has only two methods: set() and get():

* The set() method takes an object name as its first argument, and  
  a factory (in this case, a [callable](http://php.net/manual/en/language.types.callable.php)). The factories are retained in a registry (in this case, a plain-old PHP array).
* The get() method looks in a registry of instances for the requested object by name and returns it. If the object is not there, it uses the factory for that object name and retains it in the instance registry.

<http://stackoverflow.com/questions/8325619/whats-the-difference-between-the-service-locator-and-the-factory-design-pattern>

The main difference between Service Locator and Abstract Factory is that Abstract Factory suppose a new object be instantiated an returned at each requested and Service Locator needs to be configured with an object instance and every time the same instance will be returned.

<https://www.tutorialspoint.com/design_pattern/business_delegate_pattern.htm>

# Business Delegate Pattern

Business Delegate Pattern is used to decouple presentation tier and business tier. It is basically use to reduce communication or remote lookup functionality to business tier code in presentation tier code. In business tier we have following entities.

* **Client** - Presentation tier code may be JSP, servlet or UI java code.
* **Business Delegate** - A single entry point class for client entities to provide access to Business Service methods.
* **LookUp Service** - Lookup service object is responsible to get relative business implementation and provide business object access to business delegate object.
* **Business Service** - Business Service interface. Concrete classes implement this business service to provide actual business implementation logic.

## Implementation

## Step 1

Create BusinessService Interface.

*BusinessService.java*

public interface BusinessService {

public void doProcessing();

}

## Step 2

Create concrete Service classes.

*EJBService.java*

public class EJBService implements BusinessService {

@Override

public void doProcessing() {

System.out.println("Processing task by invoking EJB Service");

}

}

*JMSService.java*

public class JMSService implements BusinessService {

@Override

public void doProcessing() {

System.out.println("Processing task by invoking JMS Service");

}

}

## Step 3

Create Business Lookup Service.

*BusinessLookUp.java*

public class BusinessLookUp {

public BusinessService getBusinessService(String serviceType){

if(serviceType.equalsIgnoreCase("EJB")){

return new EJBService();

}

else {

return new JMSService();

}

}

}

## Step 4

Create Business Delegate.

*BusinessDelegate.java*

public class BusinessDelegate {

private BusinessLookUp lookupService = new BusinessLookUp();

private BusinessService businessService;

private String serviceType;

public void setServiceType(String serviceType){

this.serviceType = serviceType;

}

public void doTask(){

businessService = lookupService.getBusinessService(serviceType);

businessService.doProcessing();

}

}

## Step 5

Create Client.

*Client.java*

public class Client {

BusinessDelegate businessService;

public Client(BusinessDelegate businessService){

this.businessService = businessService;

}

public void doTask(){

businessService.doTask();

}

}

## Step 6

Use BusinessDelegate and Client classes to demonstrate Business Delegate pattern.

*BusinessDelegatePatternDemo.java*

public class BusinessDelegatePatternDemo {

public static void main(String[] args) {

BusinessDelegate businessDelegate = new BusinessDelegate();

businessDelegate.setServiceType("EJB");

Client client = new Client(businessDelegate);

client.doTask();

businessDelegate.setServiceType("JMS");

client.doTask();

}

}

## Step 7

Verify the output.

Processing task by invoking EJB Service

Processing task by invoking JMS Service

<http://stackoverflow.com/questions/2502772/java-design-pattern-business-delegate>

The *business delegate pattern* tries to decouple the clients from the business services. To achieve this you need:

* **business delegate** that is the object used by clients to request for services;
* **lookup service** is a bridge used by **business delegate** to search for services, it encapsulates the search algorithm according to the request made by the delegate;
* **business service** is the actual service that is offered to clients, usually an EJB or similar J2EE concepts.

The Business Delegate acts as a client-side business abstraction; it provides an abstraction for, and thus hides, the implementation of the business services.

Using a Business Delegate reduces the coupling between presentation-tier clients and the system's Business services

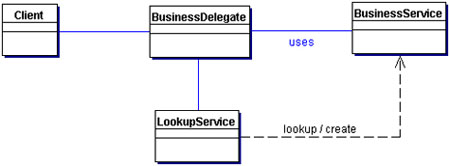
A Business Delegate uses a component called the Lookup Service. The Lookup Service is responsible for hiding the underlying implementation details of the Business service lookup code.

The Business service is a business-tier component, such as an enterprise bean or a JMS component, that provides the required service to the client. The Business service is used to invoke the business methods on behalf of the client.

Advantages:

Business Delegate reduces coupling between presentation-tier clients and Business services. The Business Delegate hides the underlying implementation details of the Business service.

Structure:

[](http://i.stack.imgur.com/XVVIs.jpg)

<http://stackoverflow.com/questions/7168714/what-is-the-purpose-of-a-delegation-pattern>

Delegation is not exactly a 'design pattern' in the sense used in the GoF book. It is useful in a number of scenarios, and is a base for other patterns

* when you want to perform some additional actions before/after you delegate (that's the Decorator pattern, but it's based on delegation). For example, Collections.synchronizedList(..) creates a new collection that delegates to the original one, but has its methods synchronized.
* when you have incompatible interfaces and you want to adapt one to the other (the adapter pattern). You get the original object and delegate to it from methods that conform to the desired interface. For example, there's the EnumerationIterator class, that adapts enumerations to the Iterator interface. The class has a hasNext() method which delegates to enumeration.hasMoreElements()
* when you want to hide some complexity from the user of your class, you can have methods that delegate to different actual workers. For example, a Car can have start(), openWindow() and brake(), but each of these methods will actually delegate to the engine, el.windows and braking system

Imagine you have the classes Car and Engine:

public class Car {

private Engine engine = new Engine(); //or inject it externally

public void start() {

engine.start();

}

}

In this example the Car delegates to the underlying Engine. The user of the car cannot directly start the engine (unless he is a mechanic). But he can tell the car to start, and the car in turn tells the engine to start.

You'd want to use it whenever you use object composition and you need to use a method of one of the composing objects. In that case you create a method that delegates to it.