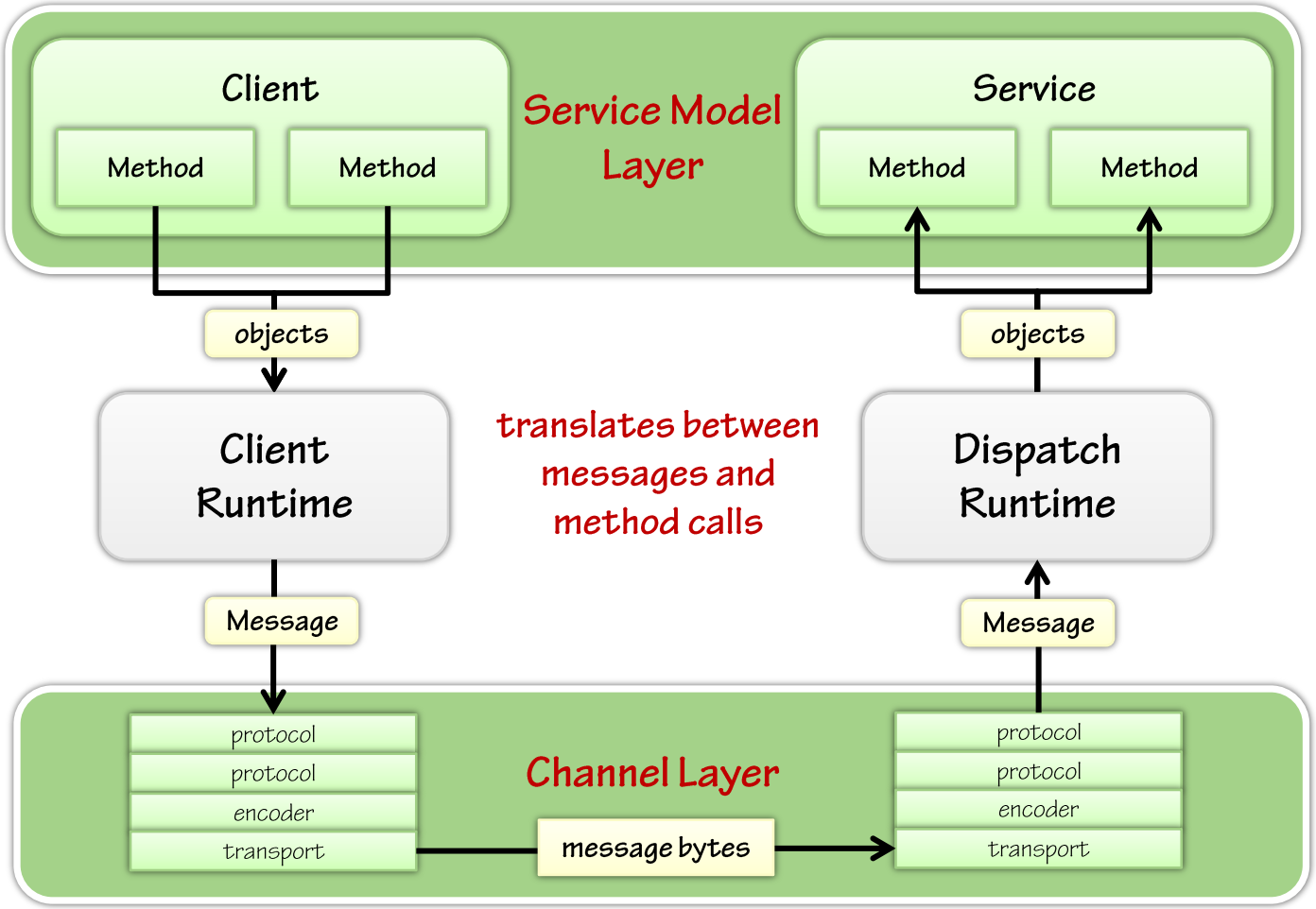
Extensibility

Picking up where WCF leaves off

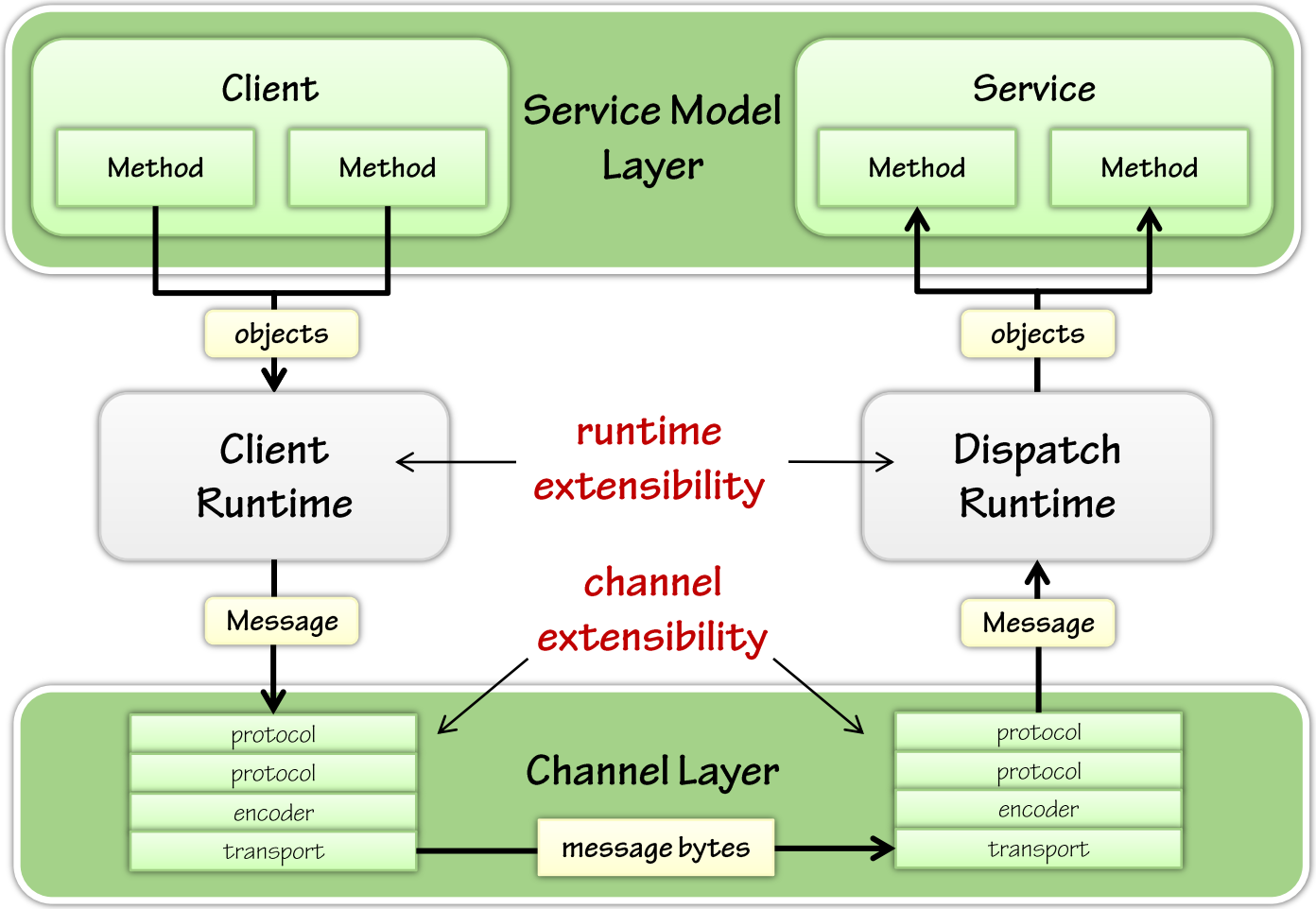
Overview

- **WCF runtime architecture**
- **Dispatch/client runtime interception**
  - Implementing interceptors
- **Applying extensions with behaviors**
  - The four types of behaviors
  - Techniques for applying behaviors
  - Implementing behaviors
- **Sharing state with extension objects**
- **Making it all easy to use**
  - With custom host & factory classes

WCF runtime architecture



WCF extensibility



Channel extensibility overview

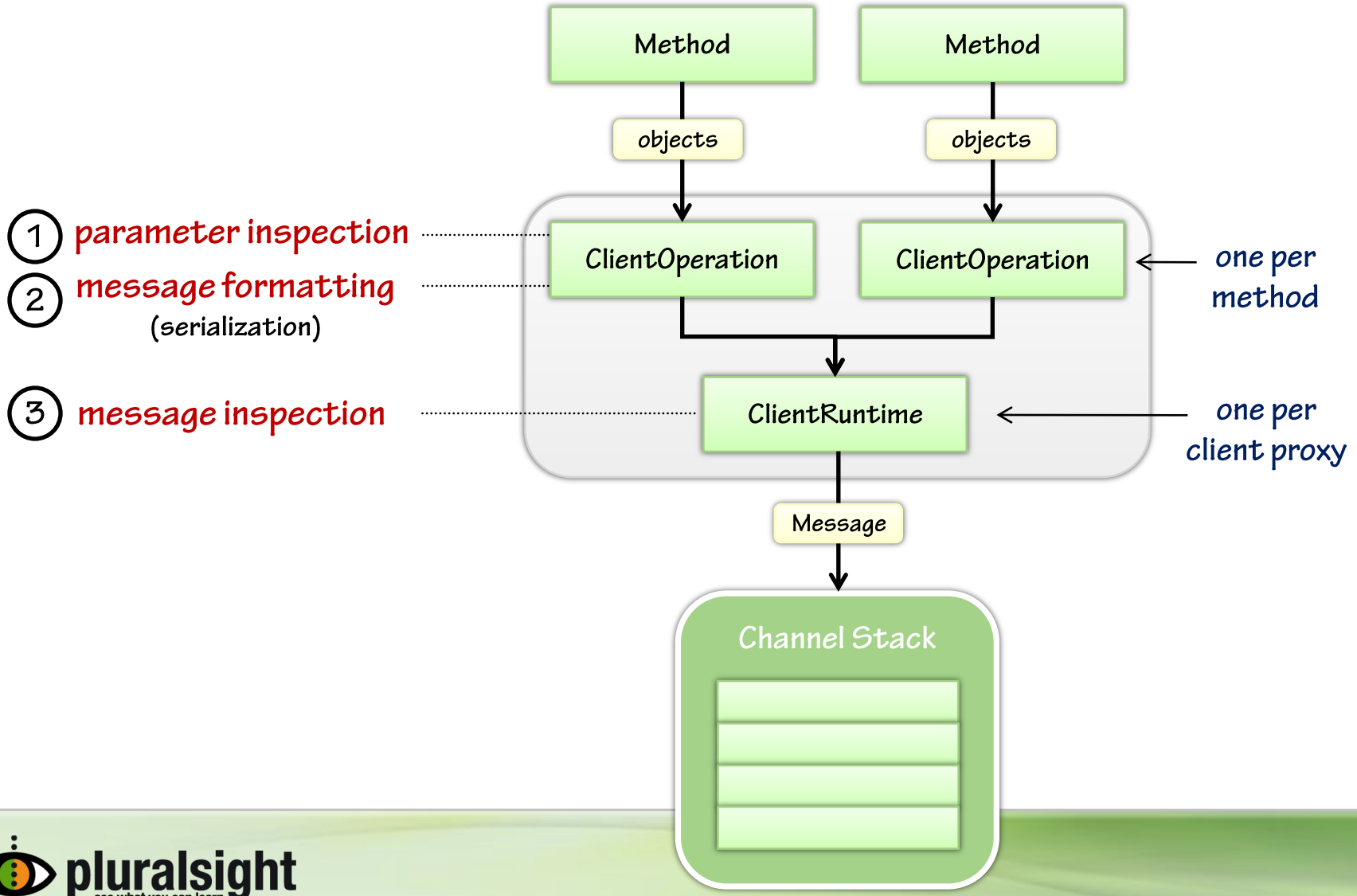
- **The WCF channel layer is completely extensible**
  - Allows for custom message encoders (formats) & protocols
  - Also makes it possible to implement custom transports
- **Using custom channel components directly isn't trivial**
  - Implement a custom binding to make them easy for others to use
  - You may also need to implement metadata extensions
- **WCF channel extensibility is an advanced topic**
  - However, most developers won't need to use these techniques
  - Most needs can be handled in the dispatch/client runtimes

See Extending the Channel Layer (MSDN) and WCF Channels Mini Book

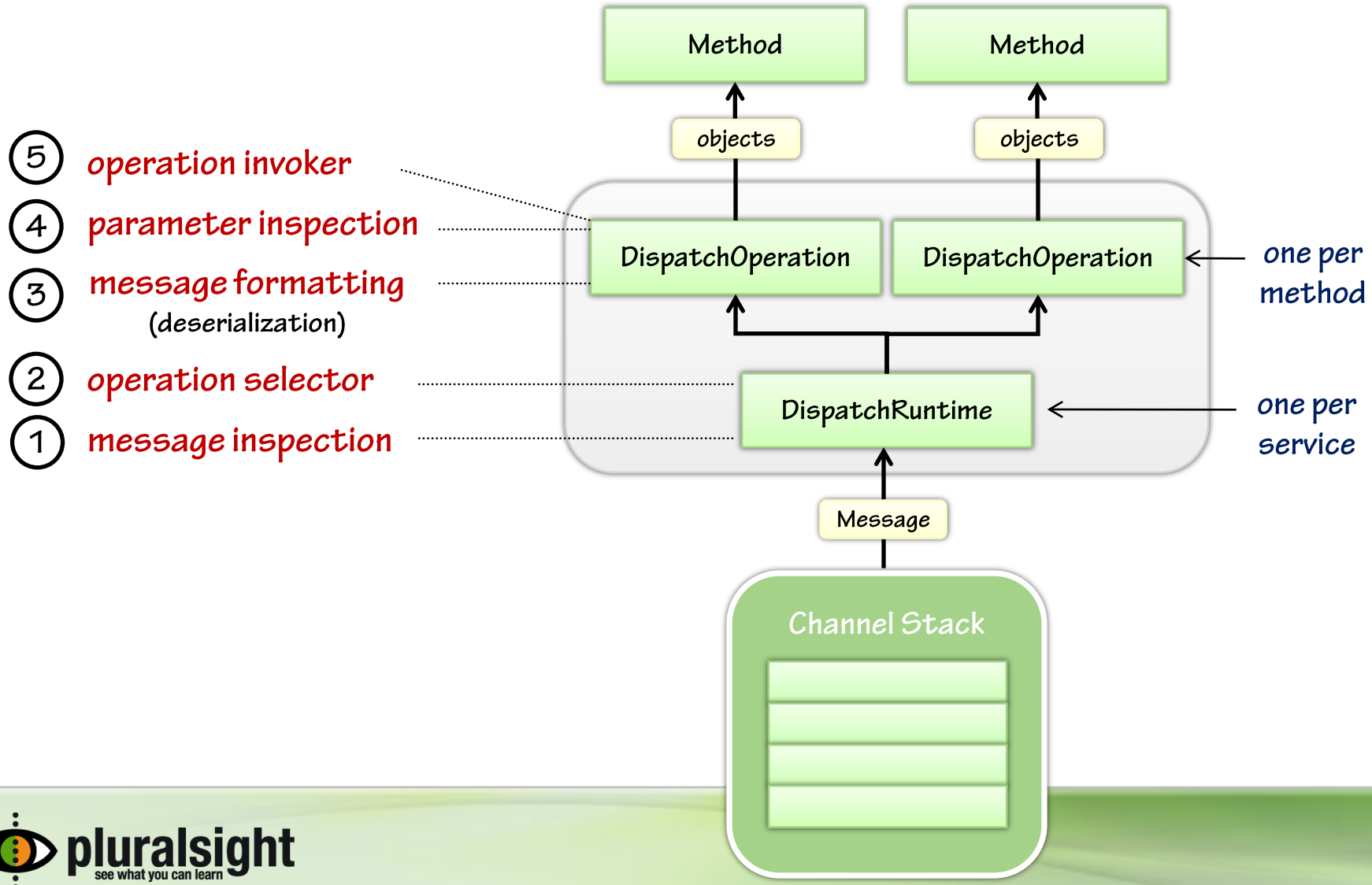
Dispatch/client extensibility overview

- **The dispatch/client runtimes shield you from messaging details**
  - They are responsible for translating between messages & method calls
  - Client runtime translates method calls into Message objects
  - Dispatch runtime translates Message objects into method calls
- **The runtimes are modeled by a few .NET classes**
  - The DispatchRuntime and ClientRuntime classes
- **These components expose numerous extensibility points**
  - Allows you to inject interceptors at key stages in the process
  - Interceptors can do pre & post processing during invocation

Client runtime stages



Dispatch runtime stages



Implementing an interceptor

- **Each interception stage is modeled by interface definitions**
  - You simply implement the interface for the stage you want to target

Stage	Interceptor Interface	Description
Parameter Inspection	IParameterInspector	Called before and after invocation to inspect / modify parameter values
Message formatting	IDispatchMessageFormatter IClientMessageFormatter	Called to perform serialization and deserialization
Message inspection	IDispatchMessageInspector IClientMessageInspector	Called before send / after receive to inspect / replace message contents
Operation selection	IDispatchOperationSelector IClientOperationSelector	Called to select the operation to invoke for the given message
Operation invoker	IOperationInvoker	Called to invoke the operation

some stages use the same interface for both dispatch/proxy

Interceptor samples

sample parameter inspector

```
public class ZipCodeInspector : IParameterInspector {
     public object BeforeCall(string operationName, object[] inputs) {
        string zip = inputs[0] as string;
        if (!Regex.IsMatch(zip, @"\d{5}-\d{4}", RegexOptions.None))
            throw new FaultException("Invalid zip code format");
        return null;
    }
...
```

sample message inspector

```
public class ConsoleMessageTracer : IDispatchMessageInspector, IClientMessageInspector {
    public object AfterReceiveRequest(ref Message request, IClientChannel channel,
        InstanceContext instanceContext) {
        request = TraceMessage(request);
        return null;
    }
    public void BeforeSendReply(ref Message reply, object correlationState) {
        reply = TraceMessage(reply);
    }
...
```

IErrorHandler

- **In addition to these stages, you can inject a custom error handler**
  - Implement IErrorHandler and inject into the channel dispatcher
- **ProvideFault is called immediately after an exception is thrown**
  - Allows you to generate a custom fault message
- **HandleError is called on a separate thread after return to client**
  - Allows you to perform more time-consuming error logging techniques

```
public interface IErrorHandler {
    bool HandleError(Exception error);
    void ProvideFault(Exception error,
        MessageVersion version, ref Message fault);
}
```

Behaviors

- **So how do you inject an extension into the dispatch/client runtime?**
  - With a custom behavior
- **Your job is to add the behavior to the service description**
  - Then WCF calls each behavior during runtime initialization
- **It's the behavior's job to inject extensions into the runtime**
  - Dispatch/ClientRuntime objects supplied to the behavior
  - Your implementation uses the runtime objects to configure extensions

Adding behaviors to the description

- There are three ways to add a behavior to the service description

explicitly via code

```
ServiceHost host = new ServiceHost(
    typeof(ZipCodeService));
host.Description.Behaviors.Add(
    new ZipCodeInspector());
host.Open();
...
```

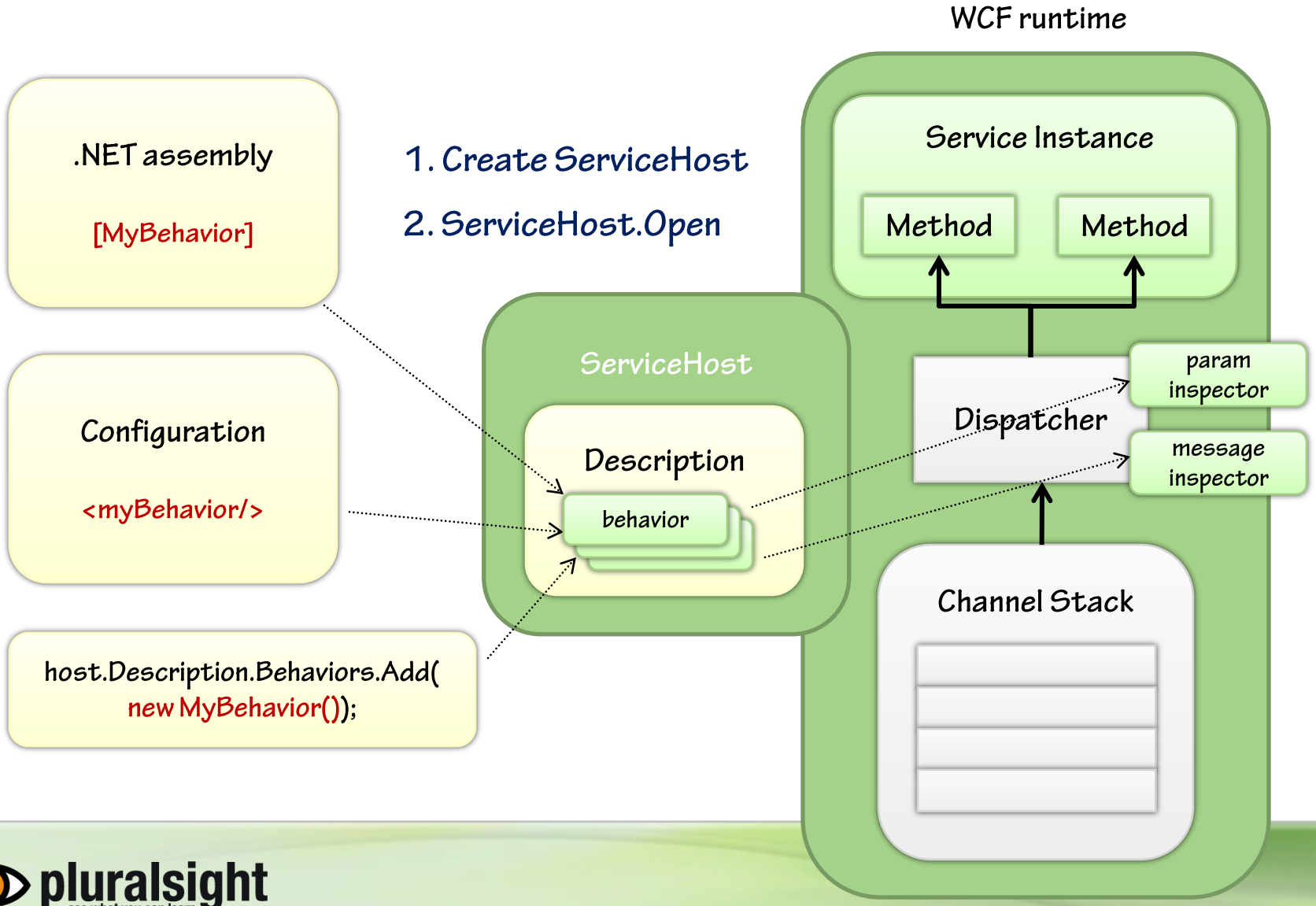
declaratively via attributes

```
[ZipCodeValidation]
[ConsoleMessageTracing]
public class ZipCodeService :
    IZipCodeService
{
   ...
```

declaratively via configuration

```
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="Default">
          <consoleMessageTracing/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <extensions>
      <behaviorExtensions>
        <add name="consoleMessageTracing"
        type="ConsoleMessageTracing,..."/>
      </behaviorExtensions>
    </extensions>
...
```

Applying behaviors at runtime



Types of behaviors

- **WCF defines four types of behaviors that map to different WCF scopes**
  - Each behavior type is modeled by a different interface definition
  - Each one is meant to be used at a particular scope

Type of Behavior	Interface	Scope of Impact			
		Service	Endpoint	Contract	Operation
Service	IServiceBehavior	X	X	X	X
Endpoint	IEndpointBehavior		X	X	X
Contract	IContractBehavior			X	X
Operation	IOperationBehavior				X

Behavior type usage details

- **There are constraints on how you use some behavior types**
  - All behavior types can be used on services
  - Service behaviors cannot be used on clients
- **There are different ways to apply each type of behavior**
  - See table below for details

Behavior Type	Usage		
	Attribute	Configuration	Explicit
Service	X	X	X
Endpoint		X	X
Contract	X		X
Operation	X		X

Implementing a behavior

- **Simply implement the appropriate interface for the behavior type**
  - Then implement the methods that you care about
  - Each behavior interface defines the following methods
  - However, signatures look different for each type

Method	Description
Validate	Called just before the runtime is built – allows you to perform custom validation on the service description
AddBindingParameters	Called in the first step of building the runtime, before the underlying channel is constructed – allows you to add parameters to influence channel stack creation
ApplyClientBehavior	Allows behavior to inject proxy (client) extensions Note: this method is not present on IServiceBehavior
ApplyDispatchBehavior	Allows behavior to inject dispatcher extensions

Implementing a behavior attribute

can be applied as an attribute

an operation behavior

```
public class ZipCodeValidation : Attribute, IOperationBehavior {

    public void ApplyClientBehavior(OperationDescription od,
        ClientOperation clientOperation) {
        ZipCodeInspector zipCodeInspector = new ZipCodeInspector();
        clientOperation.ParameterInspectors.Add(zipCodeInspector);
    }

    public void ApplyDispatchBehavior(OperationDescription od,
        DispatchOperation dispatchOperation) {
        ZipCodeInspector zipCodeInspector = new ZipCodeInspector();
        dispatchOperation.ParameterInspectors.Add(zipCodeInspector);
    }
...
```

adds a param inspector to the ClientOperation

adds a param inspector to the DispatchOperation

Implementing a behavior element

allows you to configure this as a <behaviorExtension>

returns the behavior type

creates an instance of the behavior type

```
public class ConsoleMessageTracingElement : BehaviorExtensionElement {
    public override Type BehaviorType {
        get { return typeof(ConsoleMessageTracing); }
    }
    protected override object CreateBehavior() {
        return new ConsoleMessageTracing();
    }
}
```

configure this behavior extension

```
<configuration>
  <system.serviceModel>
    <extensions>
      <behaviorExtensions>
        <add name="consoleMessageTracing"
        type="ConsoleMessageTracingElement,..."/>
      </behaviorExtensions>
    </extensions>
...
```

the name you use in a behavior configuration

Validating the description

- **Behaviors also get a chance to validate the service description**
  - This is your chance to check the configuration before initialization
  - If you don't like something you find, you can throw an exception
  - An exception prevents the client/service from being used

verifies that no BasicHttpEndpoints exist

```
public class NoBasicEndpointValidator : Attribute, IServiceBehavior {

    public void Validate(ServiceDescription sd, ServiceHostBase shb) {
        foreach (ServiceEndpoint se in sd.Endpoints)
            if (se.Binding.Name.Equals("BasicHttpBinding"))
                throw new FaultException("BasicHttpBinding is not allowed");
    }
}
```

Sharing state between extensions

- **You may need to share state across different runtime components**
  - The mechanism for managing state in WCF is IExtensionCollection<T>
- **WCF provides three standard contexts for storing state**
  - ServiceHost, InstanceContext, and OperationContext
  - Each provides an Extensions property of type IExtensionCollection<T>

State Scope	Type	Description
ServiceHost.Extensions	IExtensionCollection <ServiceHostBase>	Same lifetime as the ServiceHost
InstanceContext.Extensions	IExtensionCollection <InstanceContext>	Same lifetime as current service instance
OperationContext.Extensions	IExtensionCollection <OperationContext>	Only available for duration of invocation

Implement an extension object

- **The state collections manage objects of type IExtension<T>**
  - Hence, the objects you use must derive from IExtension<T>
  - IExtension<T> defines Attach and Detach methods
  - Called when object is added or removed from the collection

extension object for managing state

```
public class MyStateContainer : IExtension<ServiceHostBase> {
    ... // properties and methods for managing state
    public void Attach(ServiceHostBase owner) {
    }
    public void Detach(ServiceHostBase owner) {
    }
}
```

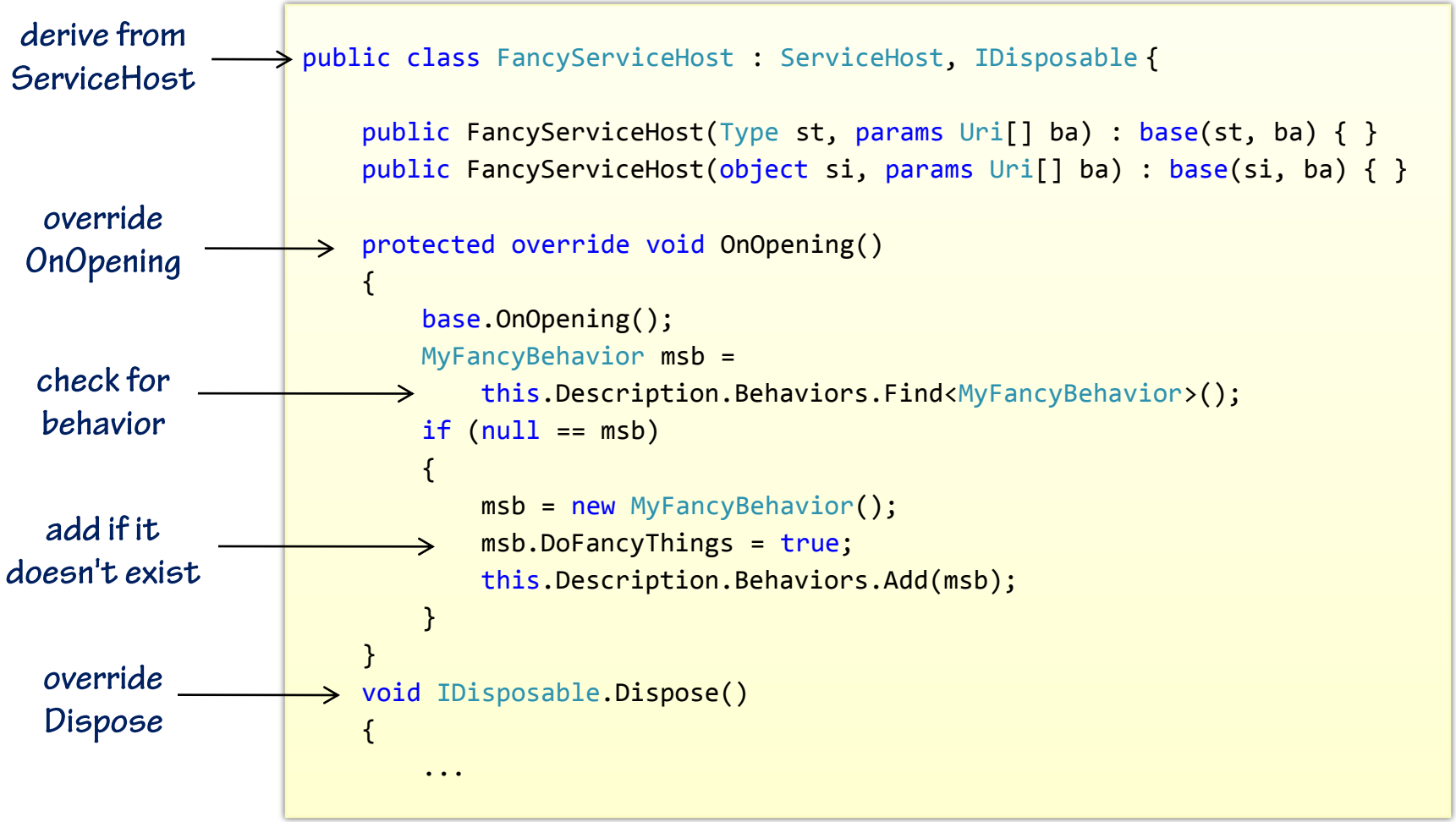
adding & retrieving from ServiceHost

```
ServiceHost host = new ServiceHost(typeof(MyService));
// add the state object
host.Extensions.Add(new MyStateContainer());
...
// grab the state object
MyStateContainer state = host.Extensions.Find<MyStateContainer>();
```

A custom ServiceHost

- **Implementing a custom ServiceHost can simplify the experience**
  - Automatically add/configure endpoints & behaviors
  - Hides the behavior details from other developers
  - Helps ensure a correct configuration and proper usage
- **Simply derive from ServiceHost**
  - Hook lifecycle events: OnOpening, OnOpened, OnClosing, etc.
- **For advanced customization, derive from ServiceHostBase**

Implementing a custom ServiceHost



A custom ChannelFactory

- **You can do the same for clients with a custom ChannelFactory**
  - Same benefits as implementing a custom ServiceHost
- **Derive from ChannelFactory<T>**
  - Hook lifecycle events: OnOpening, OnOpened, OnClosing, etc.
- **For advanced customization, derive from ChannelFactoryBase**

derive from ChannelFactory<T> →

override OnOpening and add behavior →

```
public class MyFancyFactory<T> : ChannelFactory<T>{
    public MyFancyFactory(string endpoint) : base(endpoint) { }
    protected override void OnOpening()
    {
        this.Endpoint.Behaviors.Add(new MyFancyBehavior());
        base.OnOpening();
    }
}
```

Summary

- **WCF offers numerous extensibility options**
  - The dispatch/proxy runtimes provide various interception points
  - You implement interceptors to take advantage of them
  - You plug interceptors into the runtime by applying behaviors
  - There are four behavior types: service, endpoint, contract & operation
- **You can share state between runtime components**
  - Take advantage of the IExtensionCollection<T> properties
- **You can make your extensions easy to use through**
  - By writing custom ServiceHost and ChannelFactory classes

References

- **Extending WCF with Custom Behaviors, Skonnard**
  - http://msdn.microsoft.com/msdnmag/issues/07/12/ServiceStation/
- **Extending WCF Webcast, Skonnard**
  - http://msevents.microsoft.com/cui/WebCastEventDetails.aspx?EventID=1032299313&EventCategory=5&culture=en-US&CountryCode=US
- **WCF Channels Mini Book**
  - http://wcf.netfx3.com/files/folders/product_team/entry3550.aspx
- **Extending the Channel Layer, MSDN**
  - http://msdn2.microsoft.com/en-us/library/ms731088.aspx
- **Pluralsight's WCF Wiki**
  - http://pluralsight.com/wiki/default.aspx/Aaron/TheIndigoWiki.html