CUSTOMIZING AN RPC IMPLEMENTATION

Instructor: Prasun Dewan (FB 150, dewan@unc.edu)
Replacement?

- Can explain/understand incrementally
- Can implement incrementally
- Can bootstrap off lower layers
- Can use lighter-weight lower level layers directly
- Can replace layers (separation of concerns)
How to Replace Layers and Trappers?

Layer $^i$ Sender (Port)

Layer $^i$ Trapper

Layer $^i$ Trapper

Layer $^{i-1}$ Sender (Port)

Layer $^i$ Receive Listener

Layer $^i$ Receive Notifier

Layer $^{i-1}$ Trapper

Layer $^{i-1}$ Trapper

Layer $^{i-1}$ Receive Listener

Replacement Unit in OO Language?
Replacing a Class with Another Interface

Class 1 implements Interface
Class 2

Without changing any variable declaration?
Replacing a Class with Another

DuplexClient
  InputPort

ADuplexClient
  InputPort
  implements

ACustomDuplex
  Client
  InputPort
  implements

AReceiveRegistrarAndNotifier

ReceiveNotifier
  implements

ACustomReceiveNotifier
public class ACustomDuplexObjectClientInputPort extends ADuplexObjectClientInputPort{
    public ACustomDuplexObjectClientInputPort(
        DuplexClientInputPort<ByteBuffer> aBBClientInputPort){
        super(aBBClientInputPort);
    }
    @Override
    public void send(String aDestination, Object aMessage) {
        System.out.println (aDestination + "<" + aMessage);
        super.send(aDestination, aMessage);
    }
}
public class ACustomReceiveNotifier extends AReceiveRegistrarAndNotifier {

    @Override
    public void notifyPortReceive(String aSource, Object aMessage) {
        System.out.println(aSource + "->" + aMessage);
        super.notifyPortReceive(aSource, aMessage);
    }
}
Attaching our Implementation

How do we make sure rest of the code uses our implementation?

Type variables using interfaces

How to change class instantiation?
Static Factory-Method in Instantiated Class

- Instantiated class has private constructor
- Instantiated class has static create method

DuplexClient

- Create method takes instantiation parameters
- Typically instantiation parameters become constructor parameters

Implements

ADuplexClient

- static create (...)
- Good solution for creating singletons.
- Instantiation method should be outside the replaceable class

ACustomDuplex

- static create (...)
- Awful idea for replacing a class
- Getting rid of the replaced class gets rid of the static method
Separate Factory- Class with Static Methods

Factory

create^{1}(...)  
create^{2}(...)

Separate Factory instantiates a set of related types such as ports

Provides a method for creating each type of object

Each method takes instantiation parameters

Typically instantiation parameters become constructor parameters

Changing instantiated classes means changing all the create methods

Information on how to return to the previous classes is lost and need access to source code

A factory class needs to know details about all the replaceable classes

In general it will be a package different from the replaced class so needs public constructor

In previous solution information on how to instantiate a class was tied to the class, now it is spread

Static method are not overridable
Separate Factory-Class With Instance Methods

Factory

create¹(...)
create²(...)

AFactory
AnotherFactory

Factory instantiates a set of related types such as ports
Provides a method for creating each type of object
Each method takes instantiation parameters
Typically instantiation parameters become constructor parameters
Different implementations can instantiate different classes

Factory class tied to replaced class and can be in same package, thus not needing public constructor
Will refer to this as the factory class solution
A factory can instantiate multiple related classes.

**Factory-Based Instantiation**

- Duplex InputPort Factory
  - createDuplexClientInputPort(DuplexClientInputPort<ByteBuffer>)
  - createDuplexServerInputPort(DuplexServerInputPort<ByteBuffer>)
- ADuplex InputPort Factory
- ACustom Duplex InputPort Factory

implements

A factory can instantiate multiple related classes.
public class ACustomDuplexObjectInputPortFactory extends ADuplexObjectInputPortFactory{
  public DuplexClientInputPort<Object> createDuplexClientInputPort(
      DuplexClientInputPort<ByteBuffer> bbClientInputPort) {
    return new ACustomDuplexObjectClientInputPort(bbClientInputPort);
  }
  public DuplexServerInputPort<Object> createDuplexServerInputPort(
      DuplexServerInputPort<ByteBuffer> bbServerInputPort) {
    return new ACustomDuplexObjectServerInputPort(bbServerInputPort);
  }
}
Choosing Factory?

![Diagram showing Factory, AFactory, AnotherFactory, and their methods create\(^1\)(...) and create\(^2\)(...) with a note on how to change factory choice and a key indicating that a factory is a singleton.]

How to change factory choice?

Key: A factory is a singleton
**GLOBAL ABSTRACT FACTORY**

Static Global Registry

- simplexBufferInputPortFactory
- duplexBufferInputPortFactory
- simplexObjectInputPortFactory
- duplexObjectInputPortFactory

**Static Properties**

- Has link to all factory instances
- Static methods to change and get references

Must change registry each time a new port is added
**Multiple GIPC Abstract Factories**

- **Factory Selector**
  - Has-A
  - factory
  - setFactory(Factory f)
  - create1(...)
  - create2(...)

- **GIPC**
  - Has a link to a Factory instance
  - Static methods to change and get reference
  - Can get reference to factory and invoke factory methods
  - Abstract factory provides static versions of factory creation methods for convenience

- **General Idea**
  - Helps choose between factories
**Example Selector**

```
setDuplexInputPortFactory
(DuplexInputPortFactory<Object> theInputPortFactory);

DuplexInputPortFactory<Object>
getDuplexInputPortFactory();

DuplexClientInputPort<MessageType>
createDuplexClientInputPort(String aHost,
String aServerId, String aServerName,
String aClientName);

DuplexServerInputPort<Object>
createDuplexServerInputPort
(String theServerId, String theServerName);
```
public class DuplexObjectInputPortSelector {
    static DuplexInputPortFactory<Object> inputPortFactory = 
        new ADuplexObjectInputPortFactory();
    public static void setDuplexInputPortFactory 
        (DuplexInputPortFactory<Object> theInputPortFactory) {
        inputPortFactory = theInputPortFactory;
    }
    public static DuplexInputPortFactory<Object> 
        getDuplexInputPortFactory() {
        return inputPortFactory;
    }
    public static DuplexServerInputPort<Object> 
        createDuplexServerInputPort(String theServerId, String 
        theServerName) {
        return inputPortFactory.createDuplexServerInputPort 
            (theServerId, theServerName);
    }
    public static DuplexClientInputPort<Object> 
        createDuplexClientInputPort(String theHost, String 
        theServerId, String aServerName, String theClientName) {
        return inputPortFactory.createDuplexClientInputPort 
            (theHost, theServerId, aServerName, theClientName);
    }
}
EXAMPLE USE OF SELECTORS

```java
public class ACustomCounterClient extends AMultiLayerCounterClient {
    public static void setFactories() {
        //DuplexReceivedCallInvokerSelector.setReceivedCallInvokerFactory(
        //    new ACustomDuplexReceivedCallInvokerFactory());
        DuplexReceivedCallInvokerSelector.setReceivedCallInvokerFactory(
            new ACustomDuplexReceivedCallInvokerFactory());
        DuplexSentCallCompleterSelector.setDuplexSentCallCompleterFactory(
            new ACustomSentCallCompleterFactory());
        DuplexObjectInputPortSelector.setDuplexInputPortFactory(
            new ACustomDuplexObjectInputPortFactory());
        SerializerSelector.setSerializerFactory(
            new ACustomSerializerFactory());
    }
    ...
}
```
**Class Instantiation?**

- DuplexClient InputPort
- ReceiveNotifier
  - AReceive RegistrarAnd Notifier
  - ACustom ReceiveNotifier
- ADuplexClient InputPort
- Acustom Duplex Client InputPort

How to change class instantiation?

A lighter-weight choice?

Create factories and abstract factories

(Overridable) factory methods
```java
public class ACustomDuplexObjectServerInputPort extends ADuplexObjectServerInputPort {
    public ACustomDuplexObjectServerInputPort (
        DuplexServerInputPort<ByteBuffer> aBBDuplexServerInputPort) {
        super(aBBDuplexServerInputPort);
    }
    @Override
    protected ReceiveRegistrarAndNotifier<Object>
    createReceiveRegistrarAndNotifier() {
        return new ACustomReceiveNotifier();
    }
}
```

**Example Factory Method**

Factory method vs factory class?

Cannot be called from arbitrary classes that need new instance: may not have access to instance of class that has method and the method may not be public

Overridable factory methods frowned upon but better than no factories
public class ACustomCounterClient extends AMultiLayerCounterClient{
    public static void setFactories() {
        //DuplexReceivedCallInvokerSelector.setReceivedCallInvokerFactory(
        //    new AnAsynchronousCustomDuplexReceivedCallInvokerFactory
        //);
        DuplexReceivedCallInvokerSelector.setReceivedCallInvokerFactory(
            new ACustomDuplexReceivedCallInvokerFactory());
        DuplexSentCallCompleterSelector.setDuplexSentCallCompleterFactory(
            new ACustomSentCallCompleterFactory());
        DuplexObjectInputPortSelector.setDuplexInputPortFactory(
            new ACustomDuplexObjectInputPortFactory());
        SerializerSelector.setSerializerFactory(
            new ACustomSerializerFactory());
    }
    ...
}
CALLER PIPE

Sender <Object> (RPC Port)

Call Sender <Object, Object>

SendForwarder <Object, Object>

Sender <Object> (Object Port)

Receive Listener <Object> (RPC Port)

Receive Notifier <Object> (RPC Port)

ReceiveForwarder <Object, Object>

Call Trapper: → Call Completer <Object, Object>

Receive Listener <Object> (Object Port)
CALLEE PIPE

Sender <Object> (RPC Port)

Call Sender <Object, Object>

SendForwarder <Object, Object>

Sender <Object> (Object Port)

Receive Listener <Object> (RPC Port)

Receive Listener <Object> (Object Port)

Receive Notifier <Object> (RPC Port)

ReceiveForwarder <Object, Object>

Call Trapper \(\rightarrow\) Call Invoker <Object, Object>

Receive Listener <Object> (Object Port)
public class ACustomReceivedCallInvoker extends ADuplexReceivedCallInvoker {

    public ACustomReceivedCallInvoker(LocalRemoteReferenceTranslator aRemoteHandler, DuplexInputPort<Object> aReplier, RPCRegistry theRPCRegistry) {
        super(aRemoteHandler, aReplier, theRPCRegistry);
    }

    protected void handleProcedureReturn(String aSender, Exception e) {
        System.out.println("Procedure call returning from:"
                        + " with exception:" + e);
        super.handleProcedureReturn(aSender, e);
    }

    @Override
    protected void handleFunctionReturn(String aSender, Object retVal, Class aRetType, Exception e) {
        System.out.println("Function call returning from:" + aSender
                        + " with " + " result " + retVal + "or exception:" + e);
        super.handleFunctionReturn(aSender, retVal, aRetType, e);
    }
}
public class ACustomSentCallCompleter extends ADuplexSentCallCompleter {

    public ACustomSentCallCompleter(DuplexRPCInputPort aPort, LocalRemoteReferenceTranslator aRemoteHandler) {
        super(aPort, aRemoteHandler);
    }

    @Override
    protected void returnValueReceived(String aRemoteEndPoint, Object message) {
        System.out.println("Processing return value of call:" + aRemoteEndPoint + "." + message);
        super.returnValueReceived(aRemoteEndPoint, message);
    }

    @Override
    public Object waitForReturnValue(String aRemoteEndPoint) {
        Object retVal = super.waitForReturnValue(aRemoteEndPoint);
        System.out.println(aRemoteEndPoint + "-->") + retVal);
        return retVal;
    }
}
protected Object getReturnValueOfRemoteFunctionCall(String aRemoteEndPoint, Object aMessage) {
    System.out.println("getReturnValueOfRemoteFunctionCall called");
    Object retVal = super.getReturnValueOfRemoteFunctionCall(aRemoteEndPoint, aMessage);
    System.out.println("Returning:" + retVal);
    return retVal;
}

protected Object getReturnValueOfRemoteProcedureCall(String aRemoteEndPoint, Object aMessage) {
    System.out.println("getReturnValueOfRemoteProcedureCall called");
    Object retVal = super.getReturnValueOfRemoteProcedureCall(aRemoteEndPoint, aMessage);
    System.out.println("Returning:" + retVal);
    return retVal;
}
**Duplex Sent Call Completer (New Names)**

- `ADuplexSent Call Completer`
- `Object getReturnValueOfRemoteFunctionCall(String aRemoteEndPoint, Object aCall)`
- `Object getReturnValueOfRemoteProcedureCall(String aRemoteEndPoint, Object aCall)`
- `returnValueReceived(String aSource, Object aMessage);`
**Duplex Sent Call Completer (Old Names)**

- `ADuplexSentCallCompleter`
  - `Object returnValueOfRemoteFunctionCall(String aRemoteEndPoint, Object aCall)`
  - `Object returnValueOfRemoteProcedureCall(String aRemoteEndPoint, Object aCall)`
  - `processReturnValue(String aSource, Object aMessage)`
OLD VS NEW NAMES

```java
protected Object returnValueOfRemoteProcedureCall(String aRemoteEndPoint, Object aMessage) {
    return getReturnValueOfRemoteProcedureCall(aRemoteEndPoint, aMessage);
}

protected Object returnValueOfRemoteFunctionCall (String aRemoteEndPoint, Object aMessage) {
    return getReturnValueOfRemoteFunctionCall(aRemoteEndPoint, aMessage);
}

protected void processReturnValue(String source, Object message) {
    returnValueReceived(source, message);
}
```
public class ACustomCounterClient extends AMultiLayerCounterClient{
public static void setFactories() {
    //DuplexReceivedCallInvokerSelector.setReceivedCallInvokerFactory(
    //new AnAsynchronousCustomDuplexReceivedCallInvokerFactory
    DuplexReceivedCallInvokerSelector.setReceivedCallInvokerFactory(
        new ACustomDuplexReceivedCallInvokerFactory());
    DuplexSentCallCompleterSelector.setDuplexSentCallCompleterFactory(
        new ACustomSentCallCompleterFactory());
    DuplexObjectInputPortSelector.setDuplexInputPortFactory(
        new ACustomDuplexObjectInputPortFactory());
    SerializerSelector.setSerializerFactory(
        new ACustomSerializerFactory());
}
...
}
**OBJECT TO BUFFER PIPE**

- **Sender**: <Object (Object Port)
- **Send Forwarder**: <Object, Object>
- **Serializer**: <Object, ByteBuffer>
- **Sender**: <ByteBuffer> (ByteBuffer Port)
- **Receive Listener**: <Object>
- **Receive Notifier**: <Object>
- **Receive Forwarder**: <Object, Object>
- **Receive Listener**: <ByteBuffer>
- **Proxy**
- **Translator**
- **Deserializer**: <ByteBuffer, Object>
- **Receive Listener**: <ByteBuffer>
public class ACustomSerializer extends ASimpleSerializer {
    @Override
    public synchronized Object objectFromInputBuffer(ByteBuffer inputBuffer) {
        Object retVal = super.objectFromInputBuffer(inputBuffer);
        System.out.println(inputBuffer + "-->" + retVal);
        return retVal;
    }

    @Override
    public ByteBuffer outputBufferFromObject(Object object) {
        ByteBuffer retVal = super.outputBufferFromObject(object);
        System.out.println(object + "-->" + retVal);
        return retVal;
    }
}
public class AMultiLayerCounterClient
    extends ASimpleGIPCCounterClient {
public static void main (String[] args) {
    ....
    sendByteBuffers();
    sendObjects();
    doOperations();
    while (true) {
        ReceiveReturnMessage aReceivedMessage =
        gipcRegistry.getRPCClientPort().receive();
        if (aReceivedMessage == null) {
            break;
        }
        System.out.println("Received message:" + aReceivedMessage);
    }
}}
ACustomCounterClient [Java Application] D:\Program Files\Java\jdk1.8.0_60\bin\javaw.exe (Mar 18, 2017, 1:52:03 PM)

Creating proxy out of interface: interface examples.mvc.rmi.duplex.DistributedRMICounter

Default Server<->java.nio.HeapByteBuffer[pos=0 lim=1 cap=1]
Default Server<->2
2-->java.nio.HeapByteBuffer[pos=0 lim=81 cap=128]
Default Server<->ASerializableCall(counter.increment[1])
ASerializableCall(counter.increment[1])-->java.nio.HeapByteBuffer[pos=0 lim=574 cap=848]
getReturnValueOfRemoteProcedureCall called
Returning:null
Default Server<->ASerializableCall(counter.getValuenull)
ASerializableCall(counter.getValuenull)-->java.nio.HeapByteBuffer[pos=0 lim=451 cap=848]
getReturnValueOfRemoteFunctionCall called
java.nio.HeapByteBuffer[pos=4 lim=189 cap=4194304]-->AnRPCReturnValue(6)
Processing return value of call:Default Server.AnRPCReturnValue(6)
Default Server-->6
Returning:6
6
Receive not implemented

ACustomCounterServer [Java Application] D:\Program Files\Java\jdk1.8.0_60\bin\javaw.exe (Mar 18, 2017, 1:51:55 PM)

Receive not implemented
Server0<->Client 1 (Opened)
java.nio.HeapByteBuffer[pos=4 lim=5 cap=4194304]-->null
Forwarding to receive listener unserializable byte buffer:java.nio.HeapByteBuffer[pos=4 lim=5 cap=4194304]
Client 1-->java.nio.HeapByteBuffer[pos=4 lim=5 cap=4194304]
java.nio.HeapByteBuffer[pos=4 lim=85 cap=4194304]-->2
Client 1-->2
java.nio.HeapByteBuffer[pos=4 lim=578 cap=4194304]-->ASerializableCall(counter.increment[1])
Client 1-->ASerializableCall(counter.increment[1])
java.nio.HeapByteBuffer[pos=4 lim=455 cap=4194304]-->ASerializableCall(counter.getValuenull)
Client 1-->ASerializableCall(counter.getValuenull)
Procedure call returning from:Client 1 with exception:null
Function call returning from:Client 1 with result 6 or exception:null
AnRPCReturnValue(6)-->java.nio.HeapByteBuffer[pos=0 lim=185 cap=210]
public class AMultiLayeServerReceiveListener extends ASimpleGIPCRegistryAndCounterServer implements ReceiveListener {
    protected DistributedRMICounter counter;
    public AMultiLayeServerReceiveListener(DistributedRMICounter aCounter) {
        counter = aCounter;
    }
}

public void messageReceived(String aSourceName, Object aMessage) {
    try {
        if (aMessage instanceof ByteBuffer) {
            Integer anInt = Integer
            .parseInt(AGenericSimplexBufferServerInputPort
            .extractString((ByteBuffer) aMessage));
            counter.increment(anInt);
        }
        else {
            counter.increment((Integer) aMessage);
        }
    } catch (RemoteException e) {
        e.printStackTrace();
    }
}
public class AMultiLayerCounterServer
    extends ASimpleGIPCRegistryAndCounterServer {
protected static DuplexRPCServerInputPort duplexRPCServerInputPort;
protected static void setPort() {
    duplexRPCServerInputPort = gipcRegistry.getRPCServerPort();
}
public static void addListeners() {
    duplexRPCServerInputPort.addReceiveListener(new AMultiLayerServerReceiveListener(counter));
}
public static void main (String[] args) {
    init();
    setPort();
    addListeners();
}
LISTENER CHAINING (CALLER AND CALLEE)

Sender <Object> (RPC Port)

Call Sender <Object, Object>

SendForwarder <Object, Object>

Sender <Object> (Object Port)

Receive Listener <Object> (RPC Port)

Receive Notifier <Object> (RPC Port)

ReceiveForwarder <Object, Object>

Call Trapper \( \rightarrow \) Call Invoker <Object, Object>

Receive Listener <Object> (Object Port)
public class ACustomCounterClient extends AMultiLayerCounterClient{
public static void setFactories() {
    //DplexReceivedCallInvokerSelector.setReceivedCallInvokerFactory(new AnAsynchronousCustomDuplexReceivedCallInvokerFactory)
    DuplexReceivedCallInvokerSelector.setReceivedCallInvokerFactory(new ACustomDuplexReceivedCallInvokerFactory());
    DuplexSentCallCompleterSelector.setDuplexSentCallCompleterFactory(new ACustomSentCallCompleterFactory());
    DuplexObjectInputPortSelector.setDuplexInputPortFactory(new ACustomDuplexObjectInputPortFactory());
    SerializerSelector.setSerializerFactory(new ACustomSerializerFactory());
}
...
public class ACustomCounterClient extends AMultiLayerCounterClient{
    public static void setFactories() {
        // DuplexReceivedCallInvokerSelector.setReceivedCallInvokerFactory(
        // new AnAsynchronousCustomDuplexReceivedCallInvokerFactory
        DuplexReceivedCallInvokerSelector.setReceivedCallInvokerFactory(
            new ACustomDuplexReceivedCallInvokerFactory());
        DuplexSentCallCompleterSelector.setDuplexSentCallCompleterFactory(
            new ACustomSentCallCompleterFactory());
        DuplexObjectInputPortSelector.setDuplexInputPortFactory(
            new ACustomDuplexObjectInputPortFactory());
        SerializerSelector.setSerializerFactory(
            new ACustomSerializerFactory());
    }
    ...
}
public class AnAsynchronousCustomDuplexReceivedCallInvokerFactory
    extends ACustomDuplexReceivedCallInvokerFactory{

    @Override
    public DuplexReceivedCallInvoker createDuplexReceivedCallInvoker(
        LocalRemoteReferenceTranslator aRemoteHandler,
        DuplexInputPort<Object> aReplier, RPCRegistry anRPCRegistry) {
        return new AnAsynchronousSingleThreadDuplexReceivedCallInvoker(
            super.createDuplexReceivedCallInvoker(
                aRemoteHandler, aReplier, anRPCRegistry));
    }
}
MULTIPLE CALL THREADS?

- Could create multiple threads for calls
- Conflicts with current call scheme
  - Call id not sent
  - Cannot match calls with return values based on sequence number
  - Need to build own rpc return value = queue that wakes up all waiters who can remove value if it matches their id
  - Receive?
    - Need guarded receive
public class ACustomCounterClient extends AMultiLayerCounterClient{
public static void setFactories() {
//DuplexReceivedCallInvokerSelector.setReceivedCallInvokerFactory(  
//new AnAsynchronousCustomDuplexReceivedCallInvokerFactory
DuplexReceivedCallInvokerSelector.setReceivedCallInvokerFactory(  
    new ACustomDuplexReceivedCallInvokerFactory());
DuplexSentCallCompleterSelector.setDuplexSentCallCompleterFactory(  
    new ACustomSentCallCompleterFactory());
DuplexObjectInputPortSelector.setDuplexInputPortFactory(  
    new ACustomDuplexObjectInputPortFactory());
SerializerSelector.setSerializerFactory(  
    new ACustomSerializerFactory());
}
...
}
**CONFIGURATION FILE**

DuplexClientPort =
examples.gipc.counter.customization.ACustomDuplexObjectClientInputPort

Each abstract factory creation call can check file to see if an entry exists

Or no abstract factories and factories are needed, the instantiators can check the file
CUSTOMIZATION SUMMARY

- Key idea, multiple classes implementing same interface
- Static “provider” factory methods
  - Create instances of some provider class whose services are needed.
  - Declared in the provider class.
  - Good for a singleton (class with only one instance) as constructors of provider class can be private.
  - Now good for substitution.
- Factory classes
  - Have create methods that instantiate classes or return singletons.
  - Special classes – separate from provider classes and
- Abstract factories/selectors
  - Store factories
  - Can have create methods that use stored factories to instantiate classes
- (Instance) Factory methods
  - Overridden to change class of object instantiated
  - Can exist in factory classes or user classes (classes needing the instances)
  - Not recommended in user classes but better than direct instantiation
- Call Thread creation
  - Configurable but interferes with no call id if separate thread for each call as in RMI.
  - In RMI, caller blocks, so no danger of calls from same thread being executed out of order.
  - In GIPC should not have calls from same source(thread) executed out of order.
- Configuration file
  - Indicate which class is to be instantiated
  - Can be used with or without factory classes and abstract factories.