Suppose that we have the following architecture.



We have two components – Adder and AdderClient.

The connector is a procedure call connector. AdderClient initiates the connection to Adder and synchronously receives the result back.

There are no other constraints.

This architecture may be realized in several ways.

**Implementation 1**

public class Impl1

{

private static int add(int a, int b)

{

return a + b;

}

public static void main(String[] args)

{

System.out.println(add(1, 2));

}

}

**Mapping between the architecture and the implementation:**

Adder: Impl1.add

AdderClient: Impl1.main

Connector: Impl1.main() -> Impl1.add()

This implementation corresponds to how the architecture may be implemented in an imperative or functional language.

**Implementation 2**

public class Adder

{

public int add(int a, int b)

{

return a + b;

}

}

public class AdderClient

{

public static void main(String[] args)

{

Adder a = new Adder();

System.out.println(a.add(1, 2));

}

}

**Mapping between the architecture and the implementation:**

Adder: Adder

AdderClient: AdderClient

Connector: AdderClient.main() -> Adder.add()

**Implementation 3**

public class Adder

{

public int add(int a, int b)

{

return a + b;

}

}

public class AdderClient

{

private static int add(int a, int b)

{

Adder adder = new Adder();

return adder.add(a, b);

}

public static void main(String[] args)

{

System.out.println(add(1, 2));

}

}

**Mapping between architecture and implementation:**

Adder: Adder

AdderClient: AdderClient

Connector: AdderClient.main() -> AdderClient.add() -> Adder.add()

**Implementation 4**

import java.rmi.Remote;

import java.rmi.RemoteException;

public interface IAdder extends Remote

{

int add(int a, int b) throws RemoteException;

}

import java.rmi.registry.Registry;

import java.rmi.registry.LocateRegistry;

import java.rmi.server.UnicastRemoteObject;

public class Adder implements IAdder

{

public int add(int a, int b)

{

return a + b;

}

public static void main(String[] args)

{

try

{

Adder a = new Adder();

IAdder stub = (IAdder) UnicastRemoteObject.exportObject(a, 0);

// Bind the remote object's stub in the registry

Registry registry = LocateRegistry.getRegistry();

registry.bind("Adder", stub);

System.out.println("Adder ready");

}

catch (Exception e)

{

e.printStackTrace();

}

}

}

import java.rmi.registry.Registry;

import java.rmi.registry.LocateRegistry;

public class AdderClient

{

public static void main(String[] args)

{

String host = (args.length < 1) ? null : args[0];

try

{

Registry registry = LocateRegistry.getRegistry(host);

IAdder stub = (IAdder) registry.lookup("Adder");

System.out.println(stub.add(1, 2));

}

catch (Exception e)

{

e.printStackTrace();

}

}

}

**Mapping between architecture and implementation**

Adder: Adder

AdderClient: AdderClient

Connector:

*Adder.main*

* An object of the Adder class named “a” is created.
* A stub for this object is generated.
* The stub for the object “a” is registered with the RMI registry running on the server machine using the name “Adder”.

*AdderClient.main*

* The stub registered with the RMI registry running on the server machine using the name “Adder” is obtained.
* The method add() is called on the obtained stub.

**One-to-many mappings**

A component may be implemented in more than one way. The user could choose among the available implementations at run time.

public class LimitedPrecisionAdder

{

public int add(int a, int b)

{

return a + b;

}

}

import java.math.BigInteger;

public class ArbitraryPrecisionAdder

{

BigInteger add(BigInteger a, BigInteger b)

{

return a.add(b);

}

}

import java.math.BigInteger;

public class AdderClient

{

public static void main(String[] args)

{

boolean useAP = Boolean.parseBoolean(System.getProperty("use.ap"));

if (useAP)

{

BigInteger a = new BigInteger("1");

BigInteger b = new BigInteger("2");

ArbitraryPrecisionAdder adder = new ArbitraryPrecisionAdder();

System.out.println(adder.add(a, b));

}

else

{

LimitedPrecisionAdder adder = new LimitedPrecisionAdder();

System.out.println(adder.add(3, 4));

}

}

}

**Mapping between architecture and implementation**

Adder: LimitedPrecisionAdder, ArbitraryPrecisionAdder

AdderClient: AdderClient

Connector:

* AdderClient.main() -> ArbitraryPrecisonAdder.add()
* AdderClient.main() -> LimitedPrecisionAdder.add()

Implementation of a component may require several code modules. For example, instead of Adder and AdderClient, suppose that we have SymbolicMathServer and SymbolicMathClient. In this case, the SymbolicMathServer component would be implemented by multiple code units such as ArithmeticProvider, LinearEquationSolver, Differentiator, etc.

**Many-to-one mappings**

Although multiple components can, in principle, be implemented by a single code unit (function or class), we need not consider this as this seems to be a bad practice. Doing this makes the code units less cohesive.

**Issues**

1. In the above examples, the code structure is implicit in the implementation. Should we have a language for making the relationships (e.g., dependencies) among the code units explicit?

One argument for making the relationships explicit is that a user might want to visualize the code structure by generating its graphical representation. Having an explicit representation of the code relationships would simplify this.

2. It seems useful to not restrict the implementation language and allow users to pick the language of their choice.