

# Remote Method Invocation (RMI)

(A)

(Wyłącznie do użytku lokalnego w ramach PZR 420 i PZR 380)

*Wersja 1.2/2006*

## **Bibliografia:**

1. E.R. Harold, „JAVA. Programowanie Sieciowe”, RM
2. J. Bielecki, „JAVA 3 RMI, Podstawy Programowania Rozproszonego”, Helion
3. C.S. Horstman, G. Cornell, „Core JAVA 2. Techniki Zaawansowane”, Helion
4. R.Orfali & D. Harkey „Clien/Server Programming with JAVA & CORBA” J-W
5. <http://java.sun.com/>
6. <http://archives.java.sun.com/archives/rmi-users.html/>

# Obiekty i Komponenty

## Ery technologiczne w informatyce a „teoria fal”

(R.Orfali & D. Harkey „Clien/Server Programming with JAVA & CORBA” J-W.)

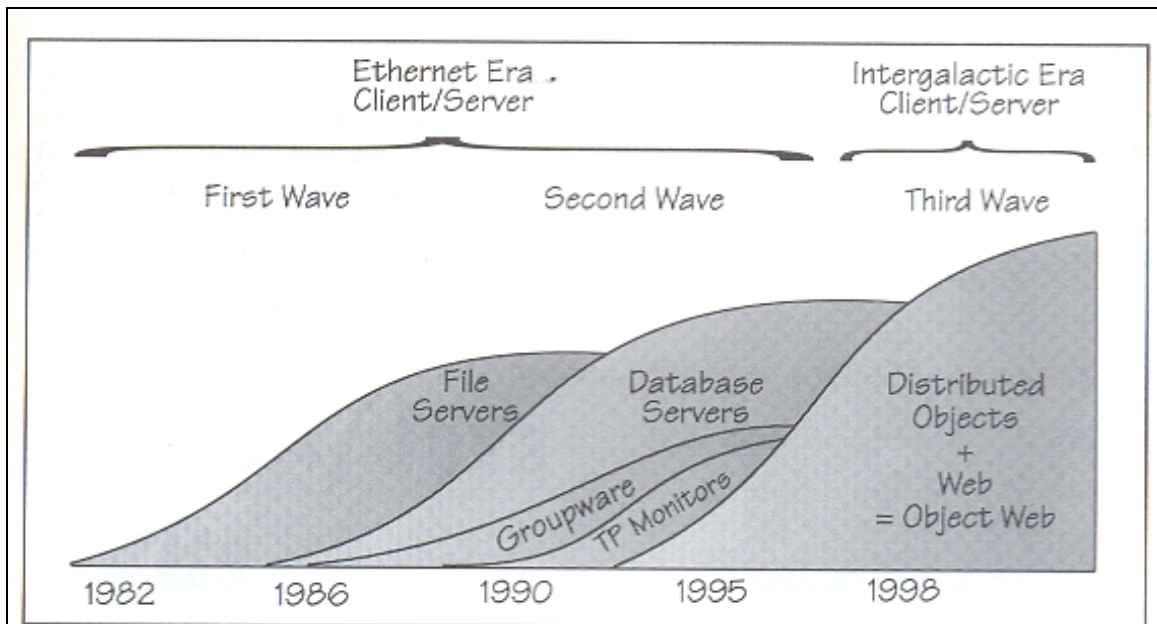
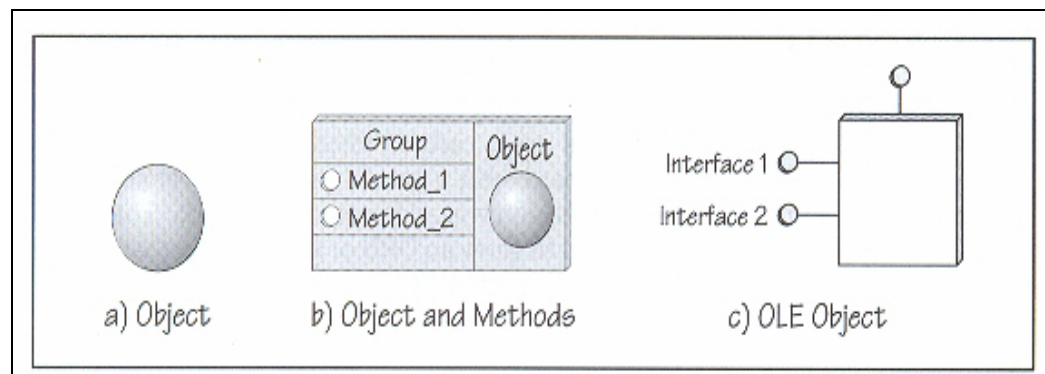


Figure 2-4. The Waves of Client/Server.

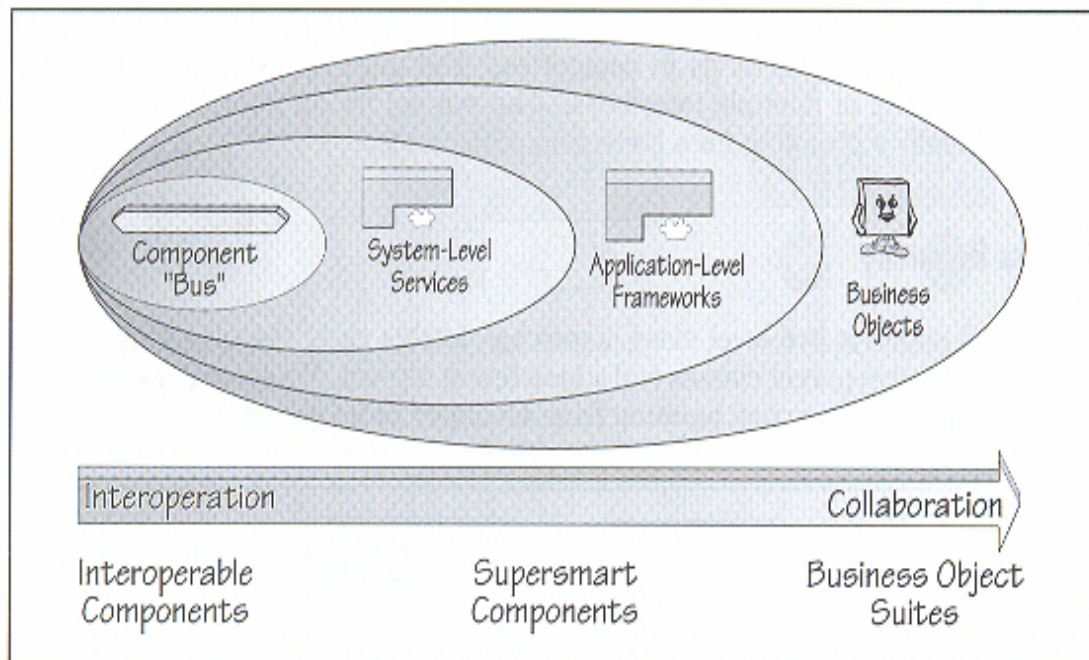
Table 2-2. Web Client/Server Versus Traditional Client/Server.

Application Characteristic	Intergalactic Era Client/Server	Ethernet Era Client/Server
Number of clients per application	Millions	Fewer than 100
Number of servers per application	100,000+	1 or 2
Geography	Global	Campus-based
Server-to-server interactions	Yes	No
Middleware	ORBs+OTMs on top of Internet	SQL and stored procedures
Client/server architecture	3-tier (or n-tier)	2-tier
Transactional updates	Pervasive	Very infrequent
Multimedia content	High	Low
Mobile agents	Yes	No
Client front-ends	OOUIs, WebTops, and shippable places	GUI
Timeframe	1998-2000	1985 till present

....od nazwy obiektu do interfejsu

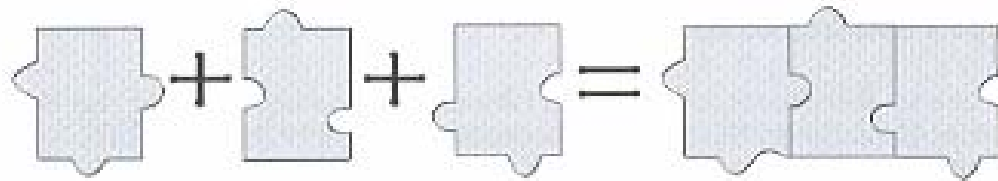


.... poziomy zarządzania komponentami



..... właściwości funkcjonalne rozproszonych komponentów

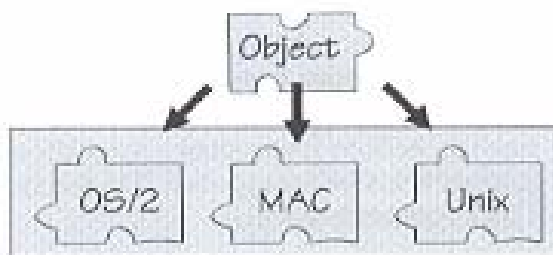
A) Plug-and-Play



B) Interoperability



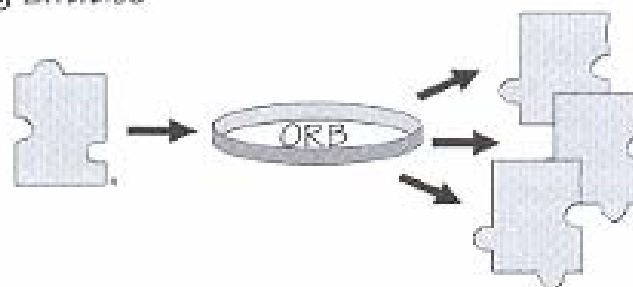
C) Portability



D) Coexistence



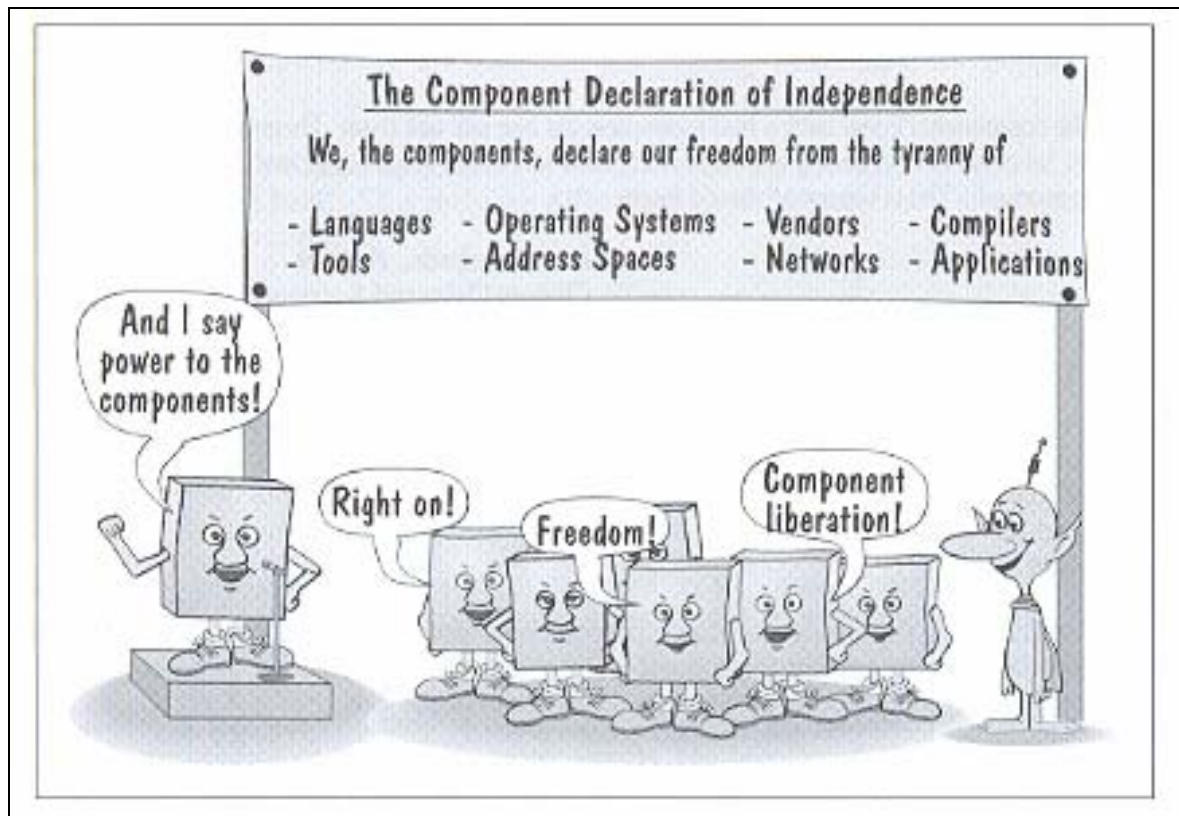
E) Self-managing Entities



..... podstawowe własności komponentów:

- . ***It is a marketable entity.*** A component is a self-contained, shrink-wrapped, binary piece of software that you can typically purchase in the open market.
- . ***It is not a complete application.*** A component can be combined with other components to form a complete application. It is designed to perform a limited set of tasks within an application domain. Components can be fine-grained objects-for example, a C++ size object; ITedium-grained objects-for example, a GUI control; or coarse-grained objects-for example, an applet.
- . ***It can be used in unpredictable combinations.*** Like real-world objects, a component can be used in ways that were totally unanticipated by the original developer. Typically, components can be combined with other components of the same family-called suites-using plug-and-play.
- . ***It has a well-specified interface.*** Like a classical object, a component can only be manipulated through its interface. This is how the component exposes its function to the outside world. A CORBNOpenDoc component also provides
- . ***It is an interoperable object.*** A component can be invoked as an object across address spaces, networks, languages, operating systems, and tools. It is a system-independent software entity.<sup>3</sup>
- . ***It is an extended object.*** Components are *bona fide* objects in the sense that they support encapsulation, inheritance, and polymorphism. However, components must also provide all the features associated with a shrink-wrapped standalone object. These features will be discussed in the next section.

**In** summary, a component is a reusable, self-contained piece of software that is independent of any application.



## So, What Is a Supercomponent?

*If the components come with a bad reputation, no one will use them. Therefore components must be of an extraordinary quality. They need to be well tested, efficient, and well documented... The component should invite reuse.*

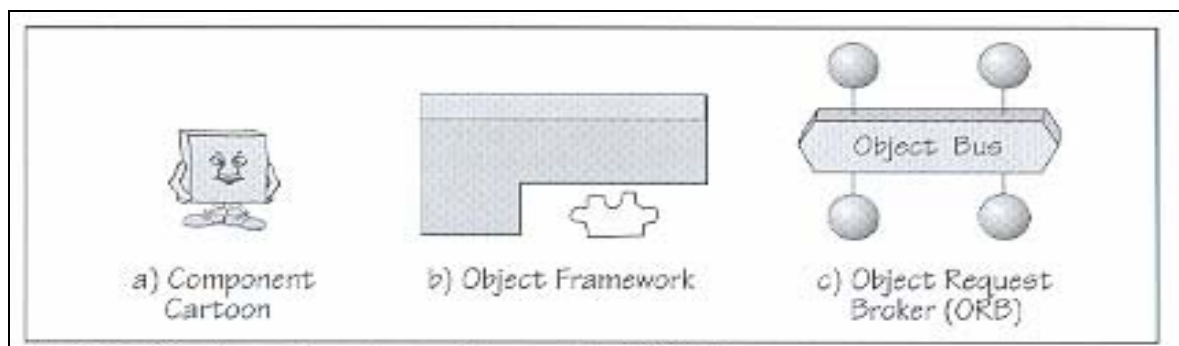
*- Ioan Jacobson, Author Object-Oriented Software Engineering (Addison-Wesley, 1993)*

Supercomponents are components with added smarts. The smarts are needed for creating autonomous, loosely-coupled, shrink-wrapped objects that can roam across machines and live on networks. Consequently, components need to provide the type of facilities that you associate with independent networked entities including:

- . **Security** – a component must protect itself and its resources from outside threats. It must authenticate itself to its clients, and vice versa. It must provide access controls. And it must keep audit trails of its use.
- . **Licensing** - a component must be able to enforce licensing policies including per-usage licensing and metering. It is important to reward component vendors for the use of their components.
- . **Versioning** - a component must provide some form of version control; it must make sure its clients are using the right version.

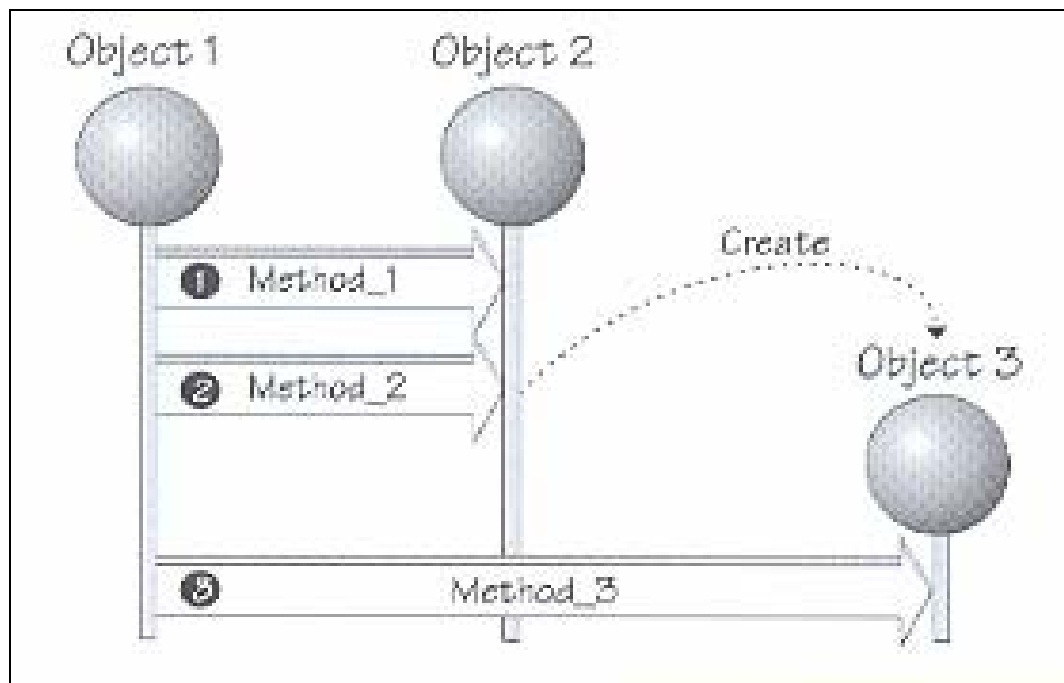
- . **Life cycle management** - a component must manage its creation, destruction, and archival. It must also be able to clone itself, externalize its contents, and move from one location to the next.
- . **Support for open tool palettes** - a component must allow itself to be imported within a standard tool palette. An example is a tool palette that supports OLE OCXs or OpenDoc parts. A component that abides by an open palette's rules can be assembled with other components using drag-and-drop and other visual assembly techniques.
- . **Event notification** - a component must be able to notify interested parties when something of interest happens to it.
- . **Configuration and property management** - a component must provide an interface to let you configure its properties and scripts.
- . **Scripting** - a component must permit its interface to be controlled via scripting languages. This means the interface must be self-describing and support late binding.
- . **Metadata and introspection** - a component must provide, on request, information about itself. This includes a description of its interfaces, attributes, and the suites it supports.
- . **Transaction control and locking** - a component must transactionally protect its resources and cooperate with other components to provide all or nothing integrity. In addition, it must provide locks to serialize access to shared resources.
- . **Persistence** - a component must be able to save its state in a persistent store and later restore it.
- . **Relationships** - a component must be able to form dynamic or permanent associations with other components. For example, a component can contain other components.
- . **Ease of use** - a component must provide a limited number of operations to encourage use and reuse. In other words, the level of abstraction must be as high as possible to make the component inviting to use.
- . **Self-testing** - a component must be self-testing. You should be able to run component-provided diagnostics to do problem determination.
- . **Semantic messaging** - a component must be able to understand the vocabulary of the particular suites and domain-specific extensions it supports.
- . **Self-installing** - a component must be able to install itself and automatically register its factory with the operating system or component registry. The component must also be able to remove itself from disk when asked to do so.

.....dodatkowe pojęcia:

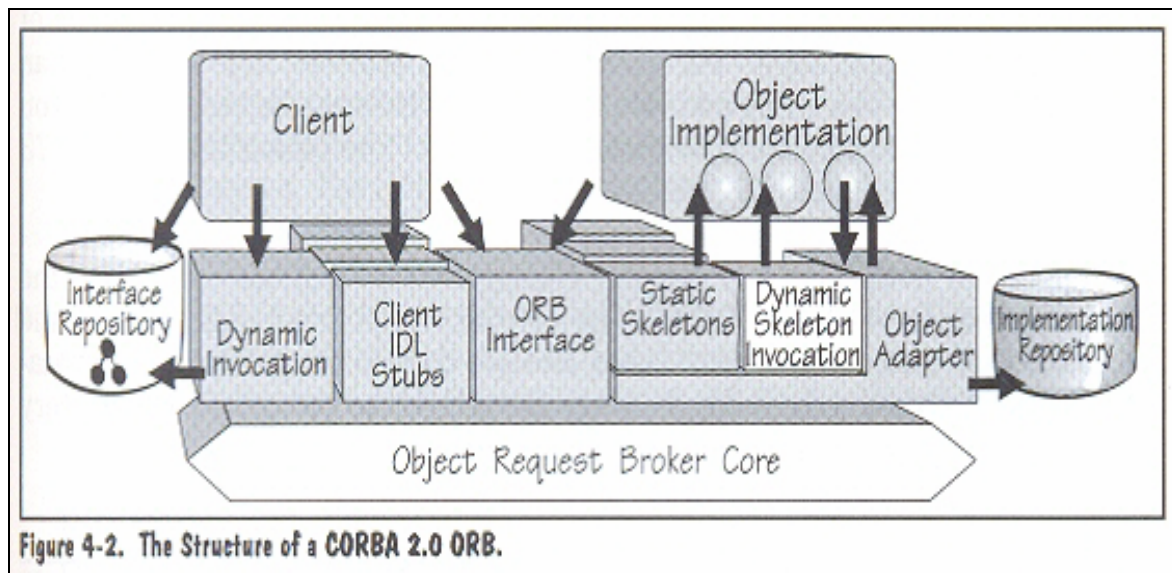




..... współpraca pomiędzy obiektami i zastosowanie diagramów:



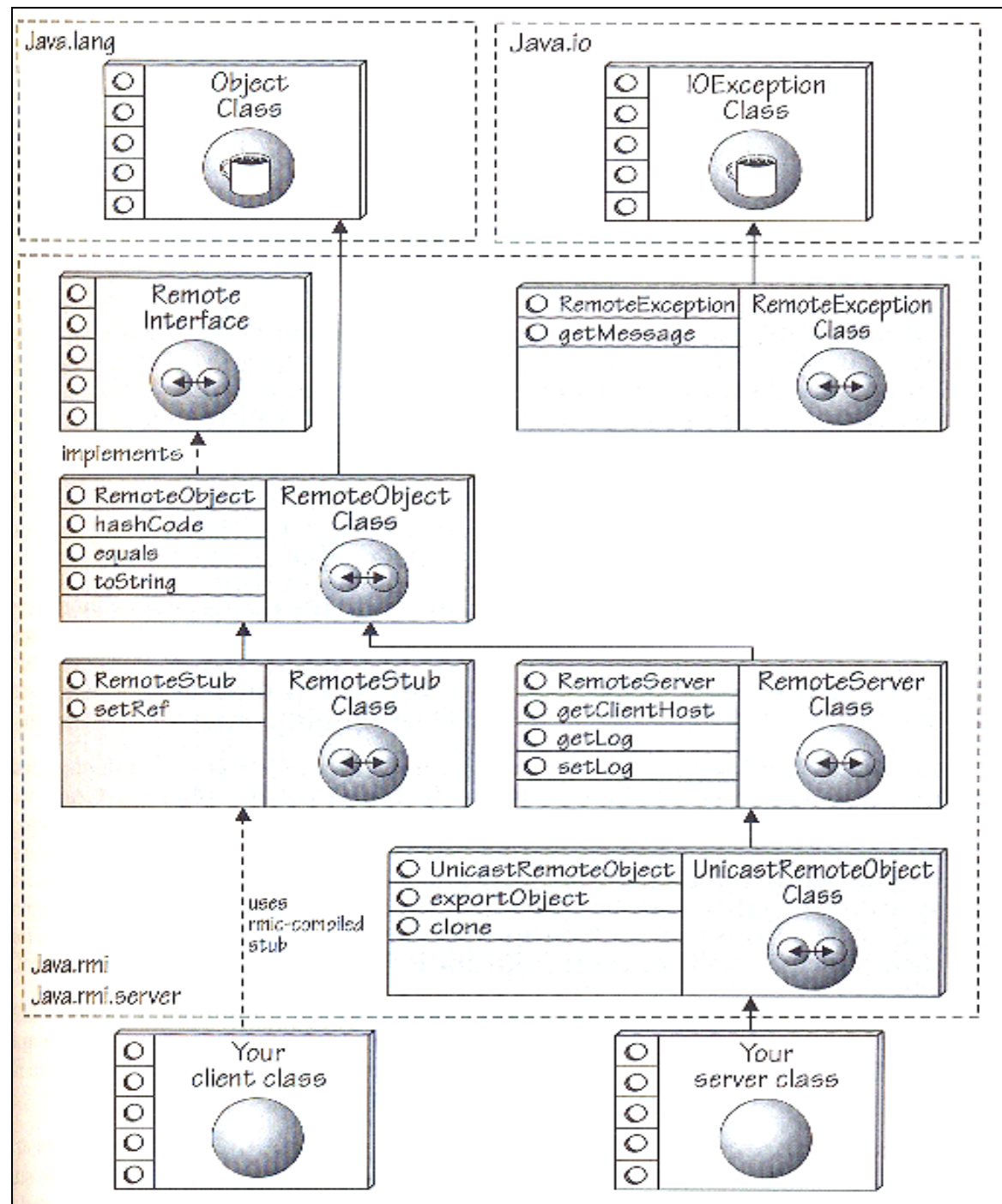
..... przykładowa architektura współdziałania rozproszonych komponentów – CORBA





**RMI**

## Klasy i metody systemu RMI dla JDK 1.1

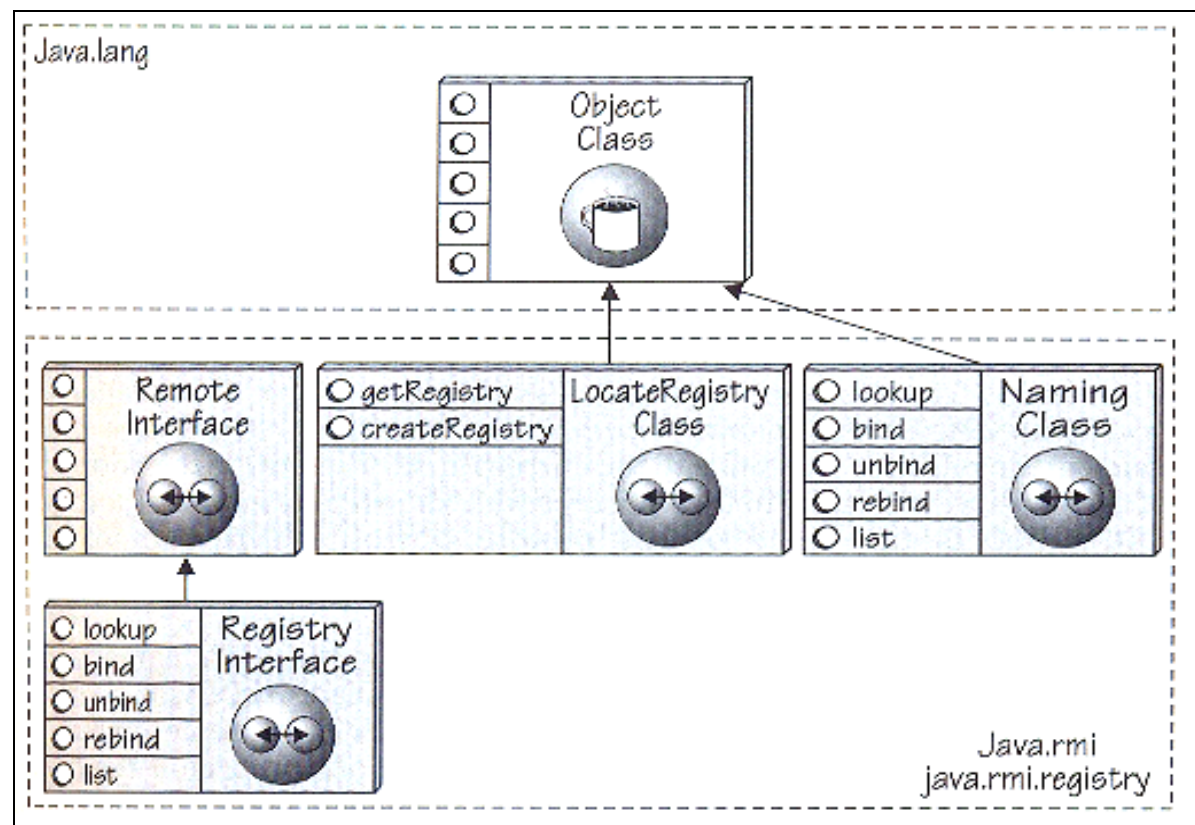


- . The **RemoteException** class is the superclass of all exceptions that can be thrown by the RMI runtime. Each method you declare in a remote interface must specify **RemoteException** in its throws clause. A remote method throws a **RemoteException** when its invocation fails—for example, when a network fails or if a server cannot be found. You construct a **RemoteException** object by invoking the class constructor. You invoke *getMessage* to obtain the cause of a remote exception.
- . The **Remote** interface is used to identify all remote objects. Every remote RMI object must implement this interface. Of course, the interface only serves to flag remote objects; it does not define any methods.
- . The **RemoteObject** class provides a remote version of the Java root **Object** class. It implements remote versions of the methods *hashCode*, *equals*, and *toString*. You should note that the default implementation for *Object.get* *Class* is appropriate for all Java objects, local or remote; the method needs no special implementation for remote objects. When used on a remote object, the *getClass* method reports the exact type of the generated stub object. The *Object.wait* and *Object.notify* methods are used for thread waiting and notification. In remote situations, these methods operate on the client's local reference to the remote object, not on the actual object at the remote site.
- . The **RemoteServer** class defines methods to create server objects and export them (Le., make them available remotely). This class is also the common superclass to all server implementations. The *getClientHost* method returns the host address of the invoking client. The *getLog* method returns a stream for the RMI call log. The *setLog* method logs RMI calls to an output stream.
- . The **UnicastRemoteObject** class implements a remote server object with the following characteristics: 1) all references to the remote object are only valid during the life of the process that creates the remote object; 2) the remote protocol requires a TCP connection-based transport; and 3) the client and server communicate parameters, invocations, and results using a stream protocol. The *exportObject* method returns a stub that serves as a local proxy for the remote object. Clients use this method to dynamically download stubs. The *clone* method returns a clone of the remote object that is distinct from the original.

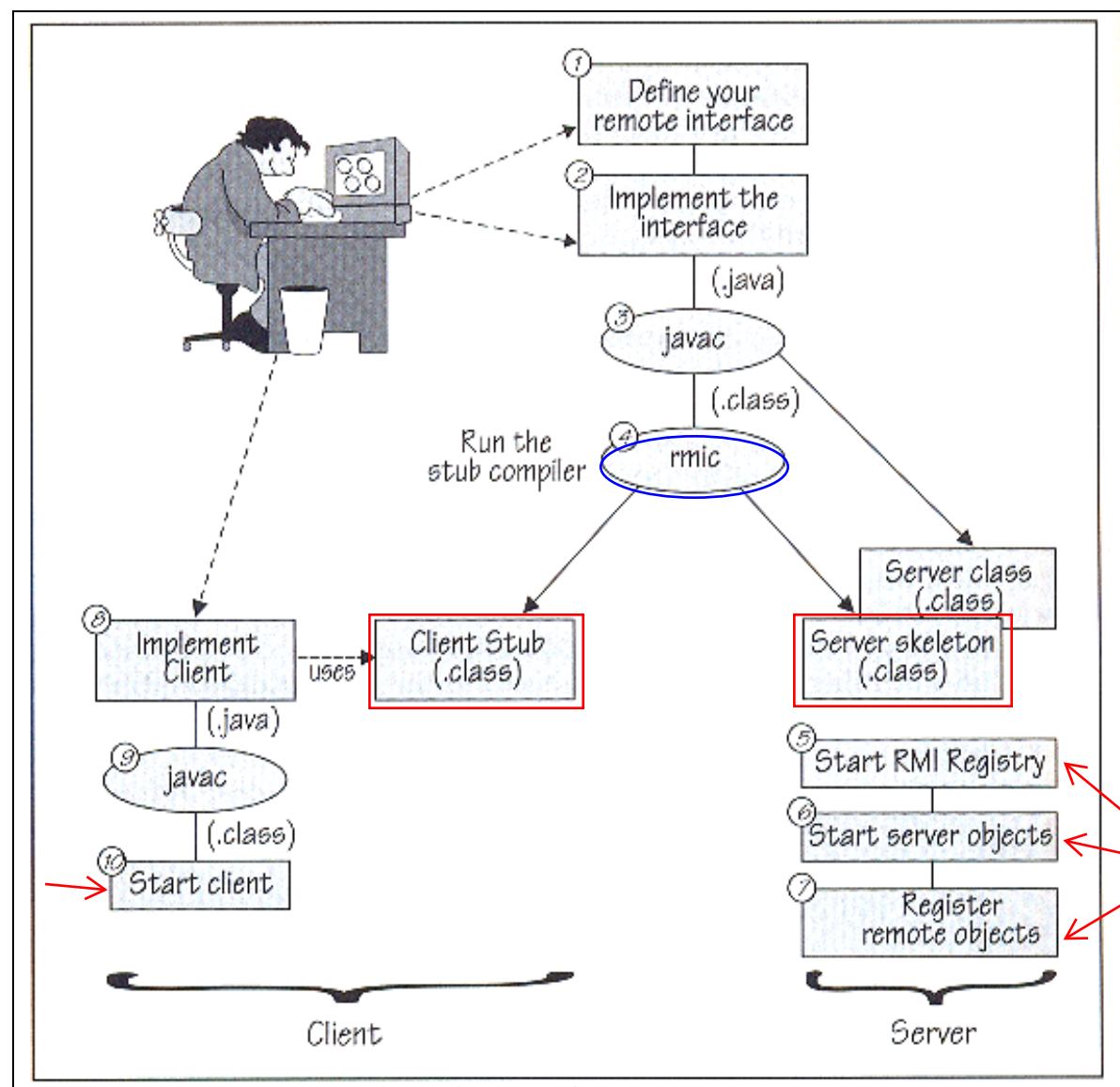
Your server classes must either directly or indirectly extend the **UnicastRemoteObject** class and inherit its remote behavior. You can implement any number of remote interfaces in your server class. Methods that do not appear in a remote interface are only available locally.

- . The **RemoteStub** class is the common superclass to all client stubs. It represents a remote stub for a specified implementation class. Either the stub object or the object itself can be passed as arguments in calls or returned to clients. During marshaling, if a reference to a remote object is passed, a lookup is performed to find the matching remote stub.

## Klasy i metody prostego serwisu katalogowego dla RMI JDK1.1



## "Kompilacja" dla Java Development Kit 1.1



## Przykład – Aplikacja “RMI Count”

// CountRmi Interface

```
public interface CountRMI extends java.rmi.Remote
{
    int sum() throws java.rmi.RemoteException;
    void sum(int val) throws java.rmi.RemoteException;
    public int increment() throws java.rmi.RemoteException;
}
```

// CountRMIClient.java RMI Count client

```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class CountRMIClient
{
    public static void main(String args[])
    {
        // Create and install the security manager
        System.setSecurityManager(new RMISecurityManager());

        try
        {
            CountRMI myCount = (CountRMI)Naming.lookup("rmi://"
                + args[0] + "/" + "my CountRMI");

            // Set Sum to initial value of 0
            System.out.println("Setting Sum to 0");
            myCount.sum(0);

            // Calculate Start time
            long startTime = System.currentTimeMillis();

            // Increment 1000 times
            System.out.println("Incrementing");
            for (int i = 0 ; i < 1000 ; i++ )
            {
                myCount.increment();
            }

            // Calculate stop time; print out statistics
            long stopTime = System.currentTimeMillis();
            System.out.println("Avg Ping = "
                + ((stopTime - startTime)/1000f)
                + " msecs");
            System.out.println("Sum = " + myCount.sum());
        } catch (Exception e)
        {
            System.err.println("System Exception" + e);
        }
        System.exit(0);
    }
}
```

// CountRMIIImpl.java, CountRMI implementation

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class CountRMIIImpl extends UnicastRemoteObject
    implements CountRMI
{
    private int sum;

    public CountRMIIImpl(String name) throws RemoteException
    {
        super();
        try
        {
            Naming.rebind(name, this);
            sum = 0;
        } catch (Exception e)
        { System.out.println("Exception: " + e.getMessage());
          e.printStackTrace();
        }
    }

    public int sum() throws RemoteException
    { return sum;
    }

    public void sum(int val) throws RemoteException
    { sum = val;
    }

    public int increment() throws RemoteException
    { sum++;
      return sum;
    }
}
```

// CountRMIServer.java

```
import java.rmi.*;
import java.rmi.server.*;

public class CountRMIServer
{
    public static void main(String args[])
    {
        // Create and install the security manager
        System.setSecurityManager(new RMISecurityManager());

        try
        {
            // Create CountRMIIImpl
            CountRMIIImpl myCount = new CountRMIIImpl("my CountRMI");
            System.out.println("CountRMI Server ready.");
        } catch (Exception e)
        { System.out.println("Exception: " + e.getMessage());
          e.printStackTrace();
        }
    }
}
```

rem CountRMI make

```
javac -d \CorbaJavaBook.2e\classes CountRMI.java
javac -d \CorbaJavaBook.2e\classes CountRMIIImpl.java
javac -d \CorbaJavaBook.2e\classes CountRMIClient.java
javac -d \CorbaJavaBook.2e\classes CountRMIServer.java
rmic -d \CorbaJavaBook.2e\classes CountRMIIImpl
```



## Scenariusz:

