

Konzepte objektorientierter Programmierung

Prof. Dr. Peter Müller

Werner Dietl

Software Component Technology

Exercises 12: Mobile Code

Wintersemester 03/04

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Remote Code Execution

- A client program should be able to send code to a remote server
- The server takes the code, executes it locally and returns the result to the client
- Hand-coded example where you see how the mechanism works
- Better solution at the end

Package and Directory Layout

- Packages:
 - `common`: for client and server
 - `server`: server implementation
 - `client`: client implementation
- Directories:
 - The server only has access to the directories `common` and `server`
 - The client only has access to the directories `common` and `client`

Executable Interface

```
package common;
```

```
public interface Executable  
    extends java.io.Serializable {  
        Object run( Object param );  
    }
```

- Standardized interface for code that can be executed; takes one `Object` parameter and returns an `Object` as result
- This interface extends `Serializable` and is not a `Remote` interface!

ExeServer Interface

```
public interface ExeServer extends Remote {  
    Object execute( String name, byte[] code,  
                   Object param ) throws RemoteException;  
}
```

- The Remote interface for the server
- We transmit the class name and bytecode for the mobile code
- The method takes the parameter for the method
- The return value is the result of the code's execution

ExeServer Implementation

```
package server;

public class ExeServerImpl
    extends UnicastRemoteObject
    implements ExeServer {

    public static void main( String[] args ) {
        try {
            Naming.rebind( "ExeServer",
                           new ExeServerImpl( ) );
        } catch( Exception ex ) {
            System.err.println("Binding failed!");
            ex.printStackTrace();
        }
    }
}
```

ExeServer Implementation – ClassLoader

```
private ServerClassLoader cl;  
  
public ExeServerImpl()  
    throws java.rmi.RemoteException {  
    cl = new ServerClassLoader();  
}
```

- We use our own ClassLoader to instantiate a class from the given bytecode

ExeServer Implementation – Execute

```
public Object execute( String name,
                      byte[] code, Object param )
    throws RemoteException {
    cl.setBytecode( name, code );
    Executable exe = null;
    try {
        exe = (Executable)
            cl.loadClass(name).newInstance();
    } catch( Exception e ) {
        throw new RemoteException( "xxx!", e );
    }

    return exe.run( param );
}
```


ServerClassLoader Implementation

```
class ServerClassLoader extends ClassLoader {  
    private Hashtable cache;  
  
    public ServerClassLoader() {  
        cache = new Hashtable();  
    }  
  
    protected Class findClass(String name) {  
        byte[] b = (byte[]) cache.get( name );  
        return defineClass(name, b, 0, b.length);  
    }  
  
    public void setBytecode( String name,  
        byte[] code ) { cache.put( name, code ); }  
}
```

ExeClient Implementation

```
public static void main( String[] args ) {  
  
    String url = "rmi://localhost/ExeServer";  
    ExeServer es =  
        (ExeServer) Naming.lookup(url);  
  
    String cs_name = args[0];  
    byte[] bytecode = getBytecode( cs_name );  
  
    Object res = es.execute( cs_name, bytecode,  
                             args[1] );  
  
    System.out.println("Result: " + res );  
}
```

ExeClient Implementation – getBytecode

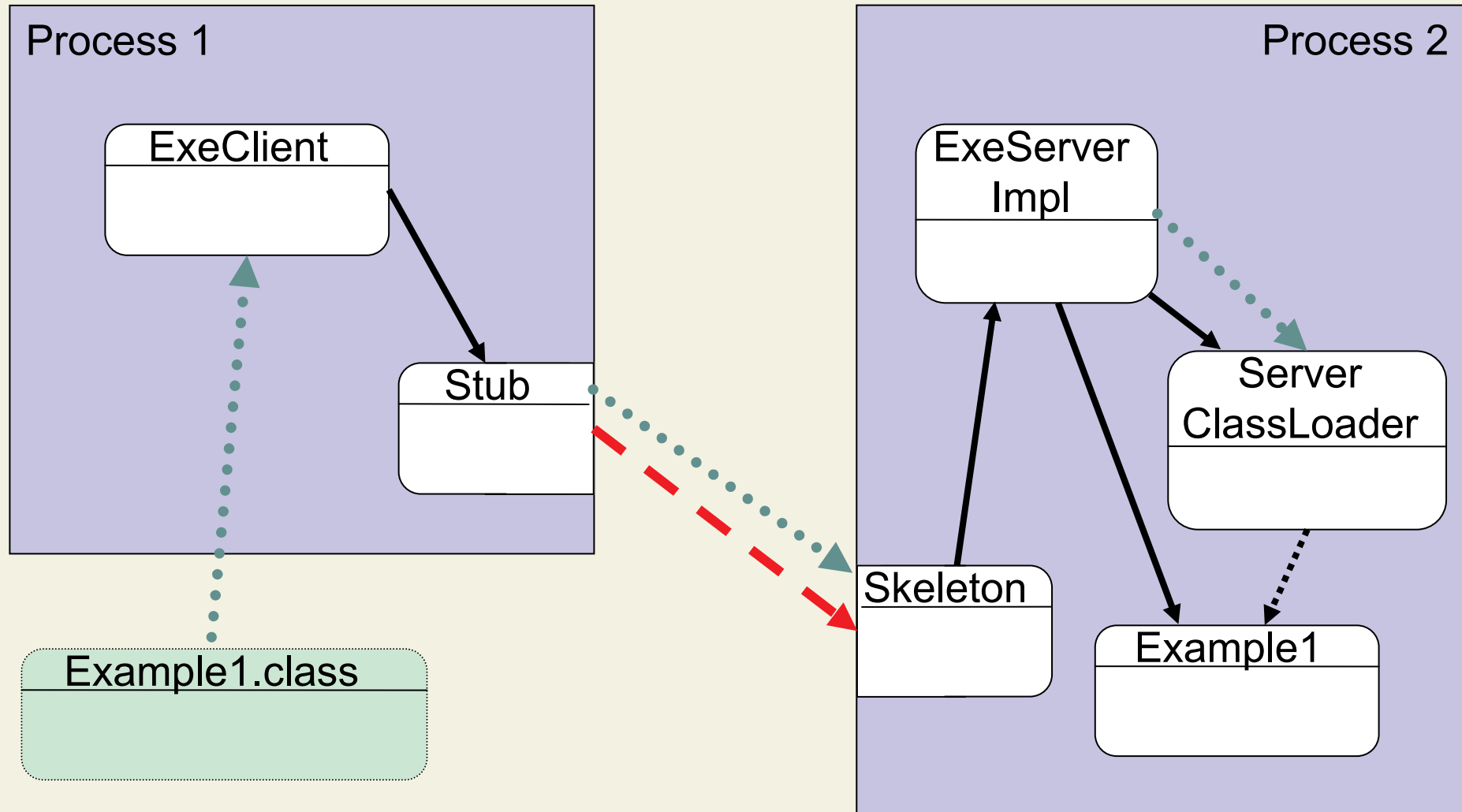
```
public static byte[] getBytecode(String name) {  
    String cs_path = cs_name.replace('.', '/') +  
        ".class";  
  
    FileInputStream fis = new FileInputStream(  
        cs_path );  
  
    byte[] buf = new byte[10000];  
    int len = fis.read( buf );  
  
    byte[] bytecode = new byte[ len ];  
    System.arraycopy(buf, 0, bytecode, 0, len );  
  
    return bytecode;  
}
```

Example for Mobile Code

```
public class Example1 implements Executable {  
  
    public Object run( Object param ) {  
        int num=Integer.parseInt( (String) param);  
  
        return Integer.toHexString( num );  
    }  
}
```

- Not really an exciting example...

System Overview



Problems of this solution

- No sandbox for the mobile code, the code can do everything the server is allowed to do → security problem
- No object structures – just one class code is transferred. Usually the mobile code will be implemented by more than one object and we need to transfer the whole structure

Better Java Solution

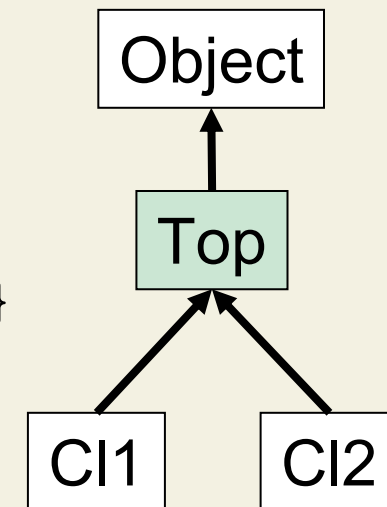
- Take a look at the Java Tutorial:
`http://java.sun.com/docs/books/...`
`...tutorial/rmi/example.html`
- Use an `RMI SecurityManager` for safety
- Make the client code available remotely, e.g. via HTTP
- Use a policy file to allow the client and server to exchange bytecode and to restrict the actions of the mobile code
- System loads bytecode on demand whenever it needs the implementation of a class

Bytecode Verification – Classes

```
abstract class Top {  
    abstract void m();  
}
```

```
class C11 extends Top {  
    public void m() {  
        System.out.println("C11.m"); }  
}
```

```
class C12 extends Top {  
    public void m() {  
        System.out.println("C12.m"); }  
}
```



Example Program

```
public class Test3 {  
    public static void main( String[] args ) {  
        xxx(true);  
        xxx(false);  
    }  
  
    public static void xxx( boolean param ) {  
        Top t = null;  
  
        if( param ) {    t = new Cl1();    }  
        else {           t = new Cl2();    }  
  
        t.m();  
    }  
}
```

The Generated Bytecode

```

@signature "(Z)V"
public static void
xxx(boolean) {
    @line 24
        @aconst_null
        @astore 1
    @line 26 // if
        @iload 0
        @ifeq _L9
    @line 27 // then
        @new C11
        @dup
        @invokespecial
            void C11.<init>()
        @astore 1
        @goto _L13

```

```

@line 29 // else
_L9: @new C12
    @dup
    @invokespecial
        void C12.<init>()
    @astore 1

@line 32 // call
_L13: @aload 1

    @invokevirtual
        void Top.m()

@line 33
    @return
}

```

The Generated Bytecode

```

@signature "(Z)V"
public static void
xxx(boolean) {
    @line 24
        @aconst_null
        @astore 1
    @line 26 // if
        @iload 0
        @ifeq _L9
    @line 27 //
        @new
        @dup
        @invokespecial
            void C11.<init>()
        @astore 1
        @goto _L13

```

SCS determines
Top as type

```

@line 29 // else
_L9: @new C12
    @dup
    @invokespecial
        void C12.<init>()
    @astore 1

@line 32 // call
_L13: @aload 1
    @invokevirtual
        void Top.m()

@line 33
    @return
}

```

allowed

The Modified Bytecode

```

@signature "(Z)V"
public static void
xxx(boolean) {
    @line 24
        @aconst_null
        @astore 1
    @line 26 // if
        @iload 0
        @ifeq _L9
    @line 27 // then
        @new C11
        @dup
        @invokespecial
            void C11.<init>()
        @astore 1
        @goto _L13

```

```

@line 29 // else
_L9: @new java.lang.String
        @dup
        @invokespecial
            void
            java.lang.String.<init>()
        @astore 1

@line 32 // call
_L13: @aload 1

        @invokevirtual
            void Top.m()

@line 33
        @return
}

```

Explanation

- The SCS finds that register 1 in line _L13 has type Object
- Therefore the call to method m() is not allowed and already rejected by the bytecode verifier:

Exception in thread "main"

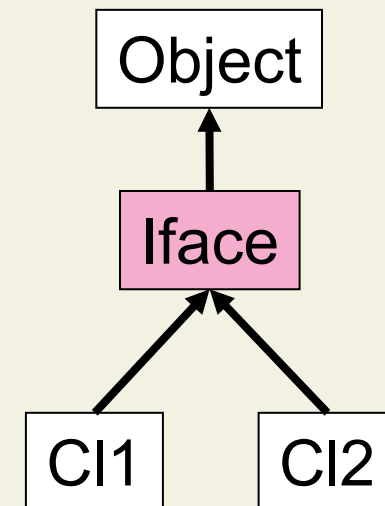
```
java.lang.VerifyError: (class: Test3,  
method: xxx signature: (Z)V  
Incompatible object argument for  
function call
```

Bytecode Verification – Interfaces

```
interface Iface {  
    void m();  
}
```

```
class C11 implements Iface {  
    public void m() {  
        System.out.println("C11.m"); }  
}
```

```
class C12 implements Iface {  
    public void m() {  
        System.out.println("C12.m"); }  
}
```



Example Program

```
public class Test1 {  
    public static void main( String[] args ) {  
        xxx(true);  
        xxx(false);  
    }  
  
    public static void xxx( boolean param ) {  
        Iface iface = null;  
  
        if( param ) {      iface = new Cl1(); }  
        else {             iface = new Cl2(); }  
  
        iface.m();  
    }  
}
```

The Generated Bytecode

```

@signature "(Z)V"
public static void
xxx(boolean) {
    @line 24
        @aconst_null
        @astore 1
    @line 26 // if
        @iload 0
        @ifeq _L9
    @line 27 // then
        @new C11
        @dup
        @invokespecial
            void C11.<init>()
        @astore 1
        @goto _L13

```

```

@line 29 // else
_L9: @new C12
    @dup
    @invokespecial
        void C12.<init>()
    @astore 1

@line 32 // call
_L13: @aload 1

    @invokeinterface
        void Iface.m() 1

@line 33
    @return
}

```


Explanation

- The SCS for register 1 after the end of the `if`-statement (`_L13`) is `Object`
- At execution the system checks whether the actual object has the correct method
- The compiler makes sure that that is the case
- What happens if we manually modify the bytecode?

The Modified Bytecode

```

@signature "(Z)V"
public static void
xxx(boolean) {
    @line 24
        @aconst_null
        @astore 1
    @line 26 // if
        @iload 0
        @ifeq _L9
    @line 27 // then
        @new C11
        @dup
        @invokespecial
            void C11.<init>()
        @astore 1
        @goto _L13

```

```

@line 29 // else
_L9: @new java.lang.String
    @dup
    @invokespecial
        void
        java.lang.String.<init>()
    @astore 1

    @line 32 // call
_L13: @aload 1

        @invokeinterface
            void Iface.m() 1

    @line 33
        @return
}

```

Program Output and Explanation

C11.m

Exception in thread "main"

java.lang.IncompatibleClassChangeError

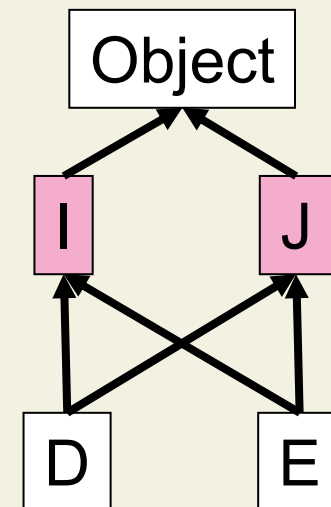
at Test2.xxx(Test2.java:32)

at Test2.main(Test2.java:20)

- The Bytecode-Verifier did not find the problem
- Only when the actual method call was executed did the virtual machine notice the problem
- Type safety limited

Handling Multiple Subtyping

- With multiple subtyping, **several smallest common supertypes** may exist
- JDK solution
 - Ignore interfaces
 - Treat all interface types as Object
 - Works because of single inheritance of classes
- Problem
 - **invokeinterface** I.m cannot check whether target object implements I
 - Runtime check is necessary



How did you do this?

- The `javap` class file disassembler produces human-readable output, but it can not be assembled again
- For the examples I used the tools `dis` and `kasm` from the `kopi` compiler suite, available from:

`http://www.dms.at/kopi/`

- Great learning experience to look at the bytecode and play around with it