

Konzepte objektorientierter Programmierung

Prof. Dr. Peter Müller

Werner Dietl

Software Component Technology

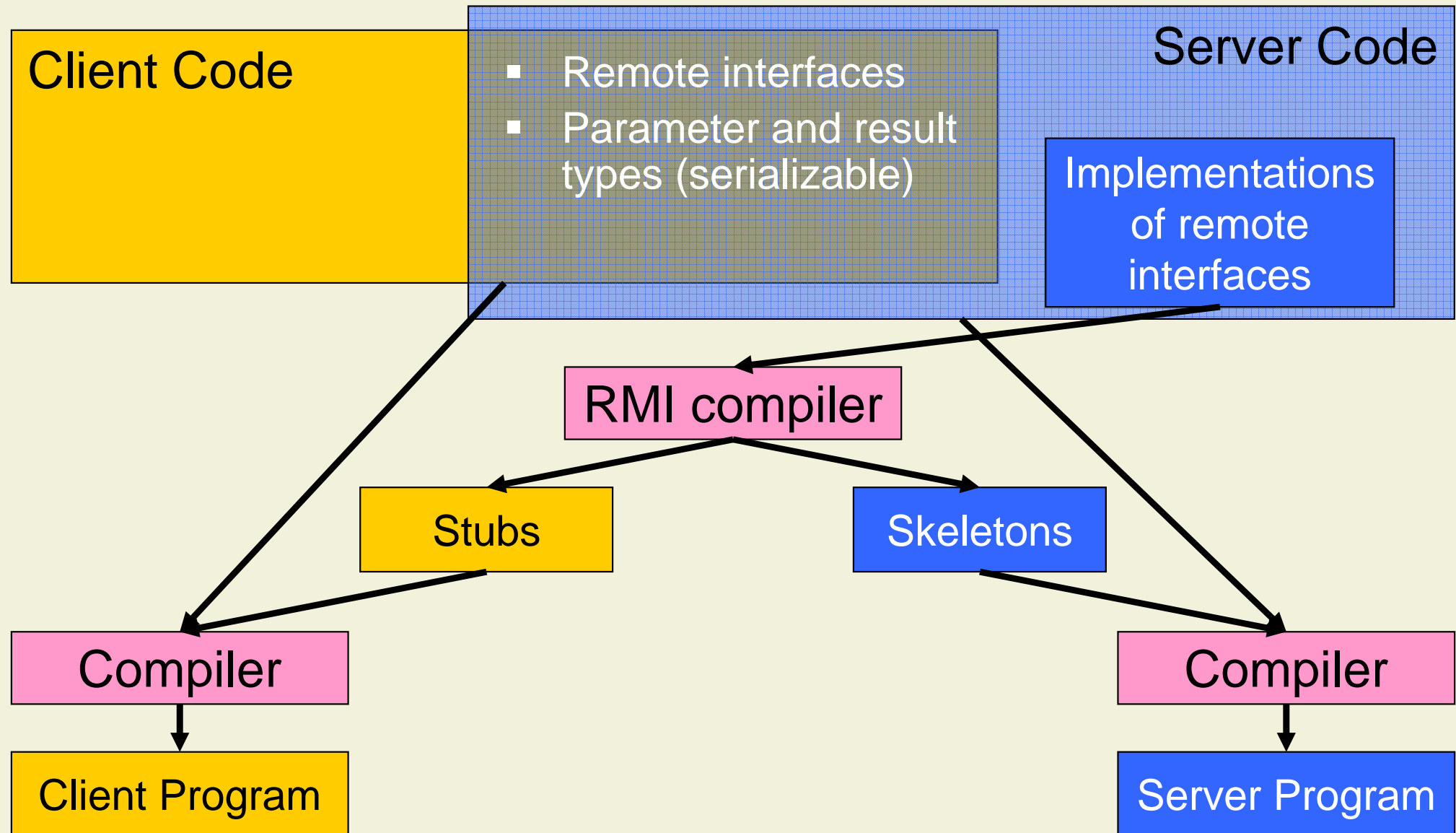
Exercises 12: Distribution and Reflection

Wintersemester 05/06

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zürich

Programming with Remote Objects



Chat Application

- One server that:
 - accepts registrations from clients
 - broadcasts messages received from one client to all registered clients

- Multiple clients that:
 - register with a server
 - send a message to the server to broadcast it
 - receive messages from the server and output them

ChatServer Interface

```
package common;
```

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

```
public interface ChatServer extends Remote {
```

```
    void register( ChatClient c )  
        throws RemoteException;
```

```
    void broadcast( String s )  
        throws RemoteException;
```

```
}
```

ChatClient Interface

```
package common;
```

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

```
public interface ChatClient extends Remote {  
    void receive( String s )  
        throws RemoteException;  
}
```

ChatClient Implementation

```
public class ChatClientImpl
    extends UnicastRemoteObject
    implements ChatClient, Runnable {

    private ChatServer mycs;

    public ChatClientImpl( ChatServer cs )
        throws java.rmi.RemoteException {
        mycs = cs;
        mycs.register( this );
    }

    public synchronized void receive(String s)
        throws java.rmi.RemoteException {
        System.out.println( "Message: " + s );
    }
}
```

ChatClient Run Method

```
public void run() {  
    BufferedReader ir = new BufferedReader(  
        new InputStreamReader(System.in));  
    String msg;  
  
    while( true ) {  
        try {  
            msg = ir.readLine();  
            mycs.broadcast( msg );  
        } catch( Exception e ) {  
            System.err.println( "Problem..." );  
        }  
    }  
}
```

ChatClient Main Method

```
public static void main( String[] args ) {  
    String url = "rmi://gem/ChatServer";  
  
    try {  
        ChatServer cs =  
            (ChatServer) Naming.lookup(url);  
        new Thread(  
            new ChatClientImpl(cs)).start();  
    } catch( Exception e ) {  
        System.err.println( "Problem..." );  
        e.printStackTrace();  
    }  
  
}
```


ChatServer Implementation

```
public class ChatServerImpl extends
    UnicastRemoteObject implements ChatServer {

    private LinkedList myclients;

    public ChatServerImpl()
        throws java.rmi.RemoteException {
        myclients = new LinkedList();
    }

    public synchronized void register(ChatClient c)
        throws java.rmi.RemoteException {
        myclients.add( c );
    }
}
```

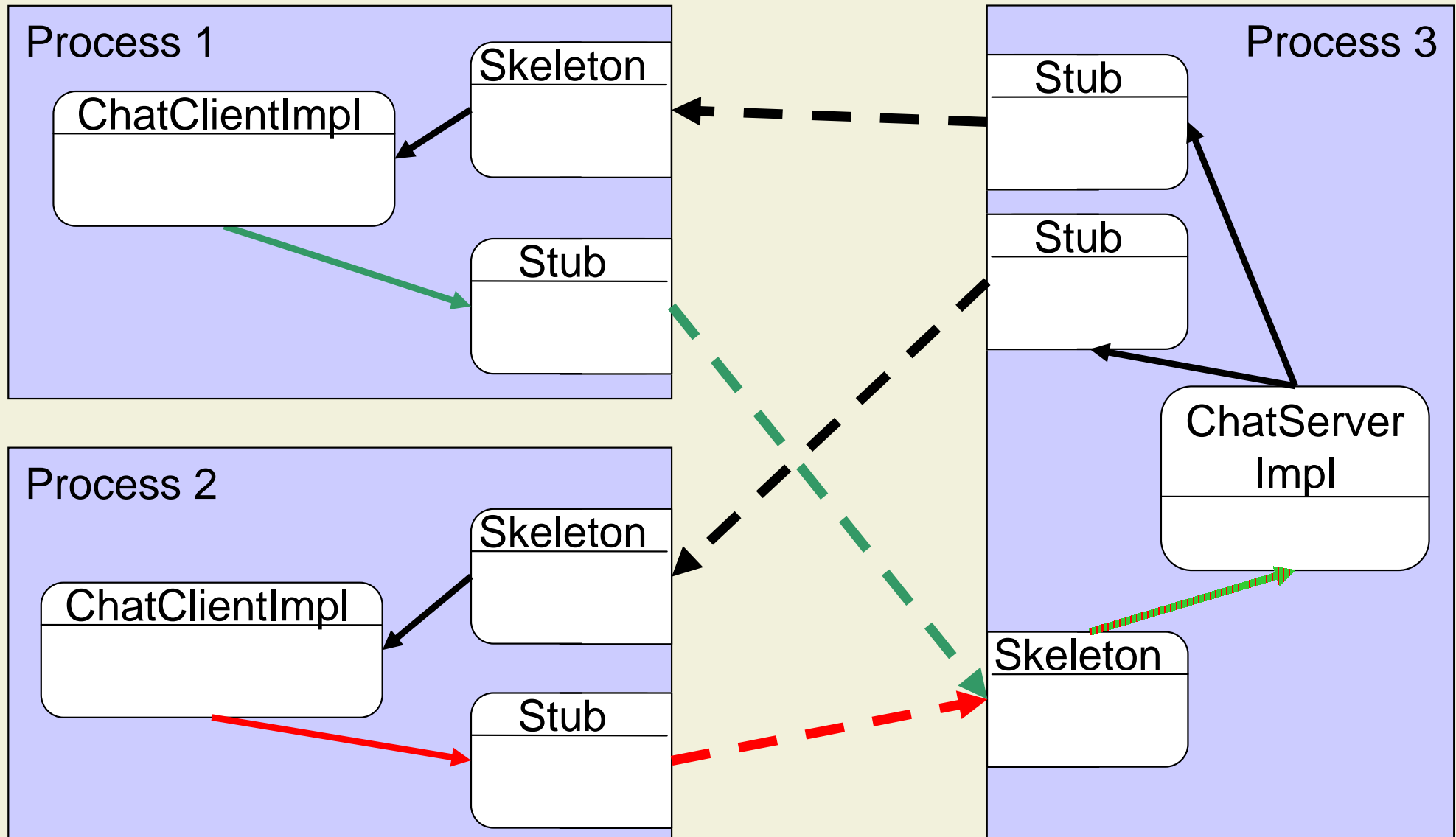
ChatServer Broadcast

```
public synchronized void broadcast( String s )  
    throws java.rmi.RemoteException {  
  
    ListIterator it = myclients.listIterator(0);  
    ChatClient c;  
  
    while( it.hasNext() ) {  
        c = (ChatClient) it.next();  
  
        c.receive( s );  
    }  
}
```

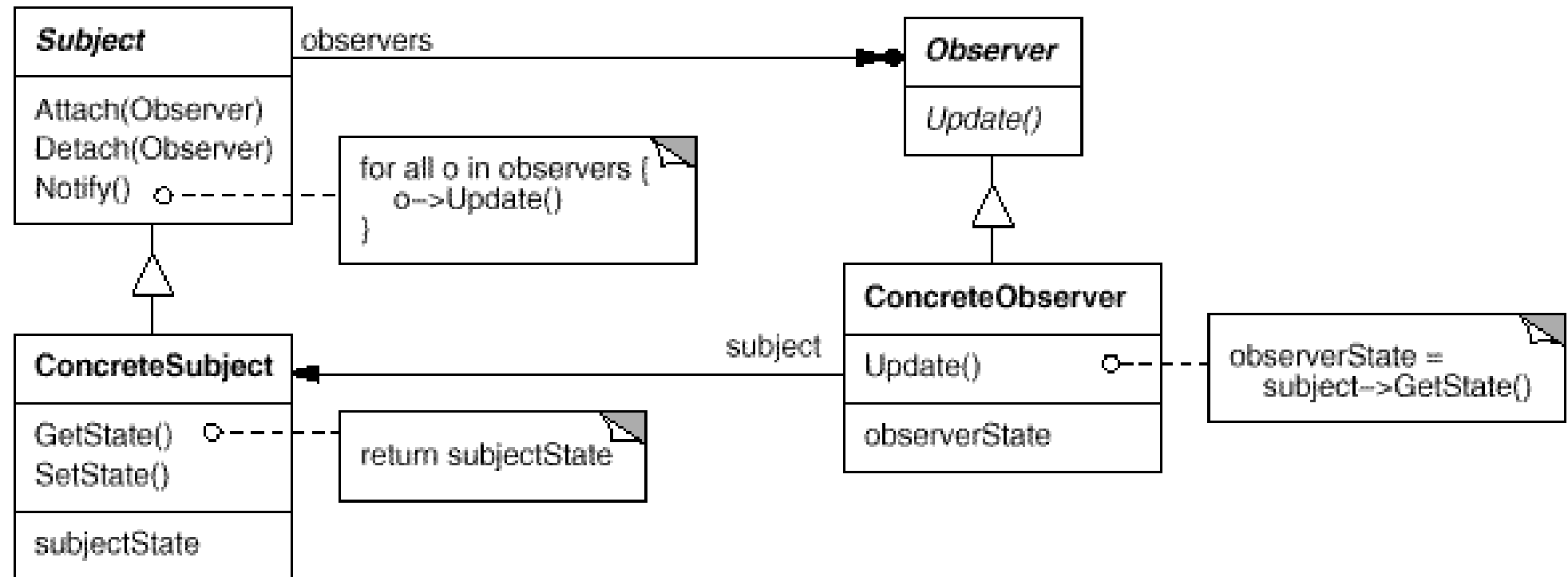
ChatServer Main Method

```
public static void main( String[] args ) {  
    try {  
  
        Naming.rebind( "ChatServer",  
                        new ChatServerImpl( ) );  
  
    } catch( Exception ex ) {  
        System.err.println( "Problem..." );  
        ex.printStackTrace();  
    }  
}
```

Process Interaction



Homework 2 – Observer Pattern



From: Gamma, Helm, Johnson, Vlissides: "Design Patterns"

Java support for the Observer Pattern

- **Subject** → `java.util.Observable`
 - `void addObserver(Observer o)`
 - `void setChanged()`
 - `void notifyObservers(Object arg)`
- **Observer** → `java.util.Observer`
 - `void update(Observable o, Object arg)`
- **Events** → `java.util.EventObject`
 - `EventObject(Object source)`
 - `Object getSource()`

ActiveValue – Setup

```
public class ActiveValue
    extends java.util.Observable {

    private int myval;

    public void run() {
        int change;
        Random crnd = new Random();

        myval = 0;

        while( true ) { ... }
    }
}
```

SignChangeObserver

```
public class SignChangeObserver
    implements java.util.Observer {

    public void update( java.util.Observable s,
                        Object e ) {

        if( !(e instanceof ChangeEvent) ) return;
        ChangeEvent ce = (ChangeEvent) e;

        if( ce instanceof IncreaseEvent ) {
            System.out.println( "+" );
        } else {
            System.out.println( "-" );
        }
    }
}
```


New Solution

- No fixed interface
- Use reflection for target observer and method
- One target can be registered with multiple methods
- One method can be registered with multiple targets

New Solution – Observable

```
import java.util.*;
import java.lang.reflect.Method;

public class MethodObservable {
    // Mapping from target object to
    // a set of methods to call
    private Map<Object, Set<Method>> calllist;

    public MethodObservable() {
        calllist = new HashMap<Object,
                                Set<Method>> ( );
    }
}
```

New Solution – Observable

```
public void addObserver(Object t, Method m) {  
    Set<Method> s;  
    if( calllist.containsKey(t) ) {  
        s = calllist.get(t);  
        s.add( m );  
    } else {  
        s = new HashSet<Method>( );  
        s.add( m );  
    }  
    calllist.put( t, s );  
}
```

New Solution – Observable

```
public void deleteObserver(Object t, Method m) {  
    if( calllist.containsKey(t) ) {  
        Set<Method> s = calllist.get(t);  
        s.remove( m );  
        if( s.isEmpty() ) {  
            calllist.remove(t);  
        } else {  
            calllist.put( t, s );  
        }  
    }  
}
```

New Solution – Observable

```
public void notifyObservers(Object arg) {  
    Object target = null;  
    Method method = null;  
    try {  
        for( Object t: calllist.keySet() ) {  
            for( Method m: calllist.get(t) ) {  
                target = t; method = m;  
                method.invoke(target, this, arg); } }  
    } catch( Exception e ) {  
        handleException( target, method, e );  
    }  
}
```

New Solution – ActiveValue

```
import java.lang.reflect.*;  
public class ActiveValue  
    extends MethodObservable {  
  
    // implementation of ActiveValue  
    // stays unchanged
```

New Solution – Observer

```
public class TextChangeObserver {  
    public void update( MethodObservable source,  
                        Object event ) {  
        if( ! (event instanceof ChangeEvent) )  
            return;  
        ChangeEvent ce = (ChangeEvent) event;  
  
        System.out.println(  
            "New Value: " + ce.getValue() +  
            " Change: " + ce.getChange() );  
    }  
}
```

New Solution – Main Program

```
public class TextMain {  
    public static void main( String[] args ) {  
        ActiveValue v = new ActiveValue();  
  
        v.addObserver( new TextChangeObserver( ),  
            TextChangeObserver.class.getMethod(  
                "update", MethodObservable.class,  
                Object.class) );  
  
        v.addObserver( new SignChangeObserver( ),  
            SignChangeObserver.class.getMethod(  
                "update", MethodObservable.class,  
                Object.class) );  
    }  
}
```