

prepare for.....

Proxy Deep Dive

JavaOne 2015 20151027

@SvenRuppert

@SvenRuppert

has been coding java since 1996

Fellow / Senior Manager

reply Group

Germany - Munich

@SvenRuppert

has been coding java since 1996

Projects in the field of:

- Automobile-industry
- Energy
- Finance / Leasing
- Space- Satellit-
- Government / UN / World-bank

Where?

- Europe
- Asia - from India up to Malaysia

prepare for.....

Proxy Deep Dive - the first steps

some words about Adapter, Delegator
and the first Proxy

@SvenRuppert

Why a Proxy ?

@SvenRuppert

Proxies you can use them for :

- generic parts inside your application

- decouple parts of your applications

- hiding technical stuff

Why a Proxy ?

@SvenRuppert

but it must be easy to use like...

```
final InnerDemoClass original = new InnerDemoClass();  
final InnerDemoInterface demoLogic =  
    ProxyBuilder.createBuilder(InnerDemoInterface.class, original)  
        .addLogging()  
        .addMetrics()  
        .addSecurityRule() -> true)  
        .addSecurityRule() -> true)  
        .build();
```


Why a Proxy ?

@SvenRuppert

but it must be easy to use like...

@Inject

@Proxy(virtual = **true**, metrics = **true**)

Service **service**;

Difference between Proxy and Adapter

Adapter - or called Wrapper

maybe : Use an Adapter if you have to use an incompatible interface between two systems.

In detail I am using/talking about an **Adapter with Delegator**.

Adapter v Proxy - Pattern Hello World

@SvenRuppert

```
public class LegacyWorker {  
    public void writeTheStrings(String[] strArray){  
        Arrays.stream(strArray).forEach(System.out::println);  
    }  
}
```



```
public interface WorkerAdapter {  
    public void workOnSomething(List<String> stringListe);  
}
```



```
public class Adapter implements WorkerAdapter {  
    private LegacyWorker worker = new LegacyWorker();  
    @Override  
    public void workOnSomething(List<String> stringListe) {  
        final String[] strings = stringListe.toArray(new String[0]);  
        worker.writeTheStrings(strings);  
    }  
}
```



Adapter v Proxy - Pattern Hello World

@SvenRuppert

```
public class Main {  
    public static void main(String[] args) {  
        final ArrayList<String> strings = new ArrayList<>();  
        Collections.addAll(strings, „A“, "B", "C", "D");  
  
        new Adapter().workOnSomething(strings);  
    }  
}
```



- No need to know the LegacyWorker
- No inheritance between LegacyWorker and Adapter
- only a transformation of an interface
- same creation time for Delegator and Adapter

Proxy - the easiest version

@SvenRuppert

We need a few steps to come from an Adapter with Delegator to a Proxy

- We need to know the LegacyWorker
- Inheritance between LegacyWorker and Adapter
- do not use it like an Adapter !

please show the code

Proxy - the easiest version - (Delegator)

@SvenRuppert

```
public interface Service {  
    String work(String txt);  
}
```

```
    public class ServiceImpl implements Service {  
        public String work(String txt) {  
            return "ServiceImpl - " + txt;  
        }  
    }
```

```
public class ServiceProxy implements Service {  
    private Service service = new ServiceImpl();  
    public String work(String txt) { return service.work(txt); }  
}
```


Security Proxy

@SvenRuppert

```
public class ServiceProxy implements Service {  
    private Service service = new ServiceImpl();  
    public String work(String txt) { return service.work(txt); }  
}
```

What could we
change now ?

SecurityProxy:
execute only if it is allowed

```
public class ServiceSecurityProxy implements Service {  
    private Service service = new ServiceImpl();  
    private String code; //who will set this ?  
    public String work(String txt) {  
        if(code.equals(„hoppel")) { return service.work(txt); }  
        else { return "noooooop"; }  
    }  
}
```



Remote Proxy - JDK only

@SvenRuppert

```
public class ServiceProxy implements Service {  
    private Service service = new ServiceImpl();  
    public String work(String txt) { return service.work(txt); }  
}
```

What could we
change now ?

RemoteProxy:
execute outside of my process

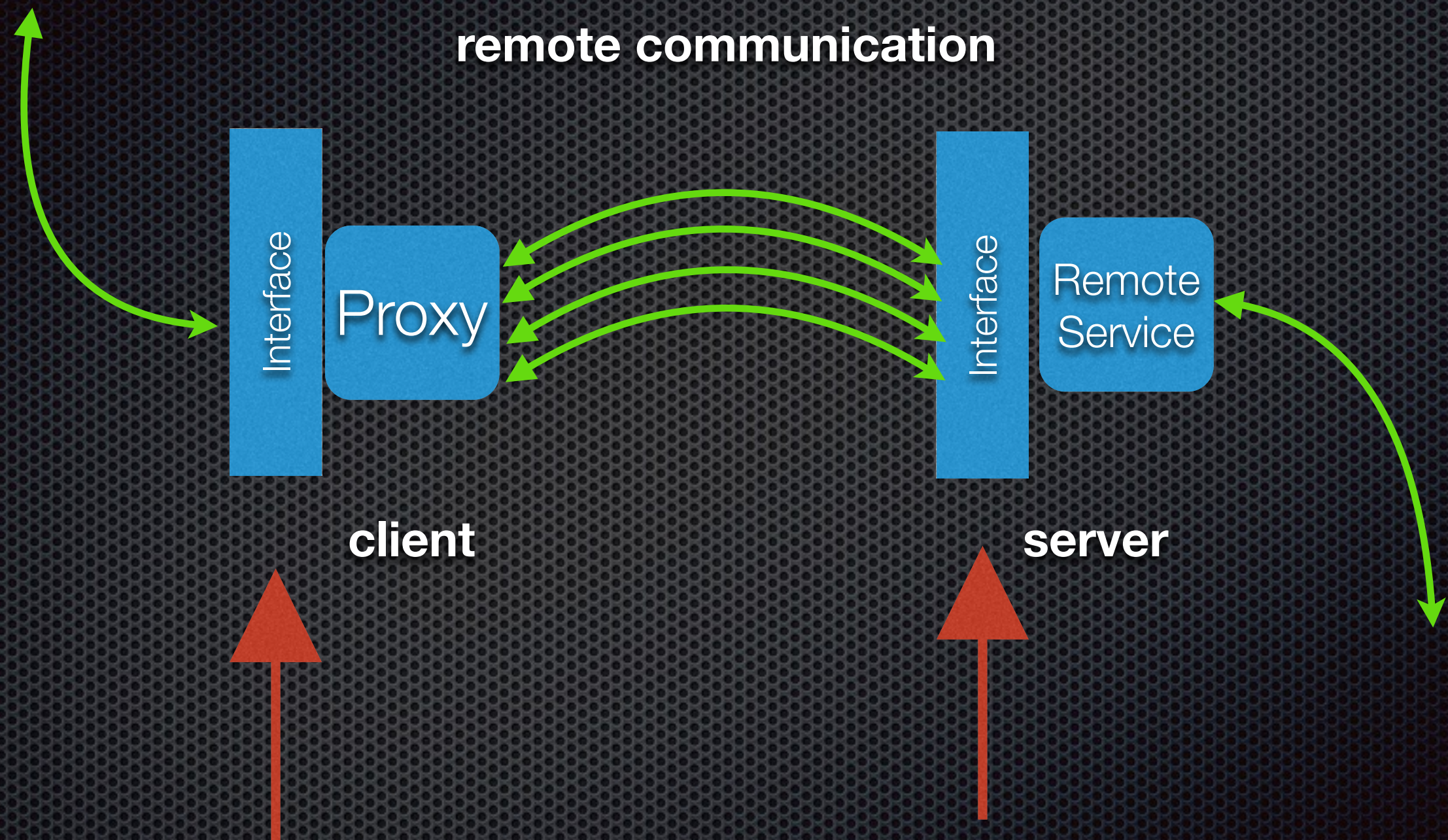
```
Service proxy = new ServiceRemoteProxy();  
String hello = proxy.work(„Hello“);
```

some work needed



Remote Proxy - JDK only

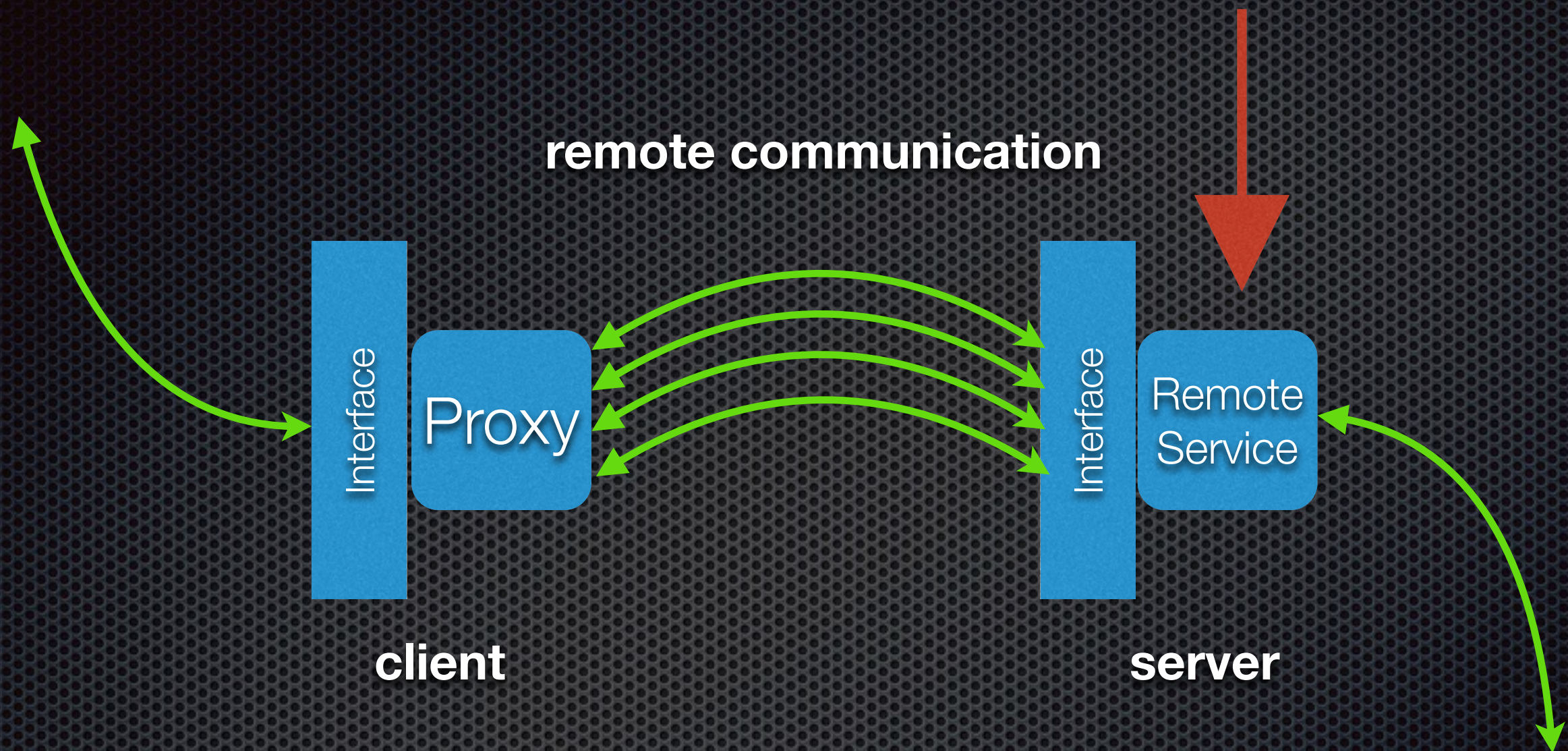
@SvenRuppert




```
@WebService  
@SOAPBinding(style = SOAPBinding.Style.RPC)  
public interface Service {  
    @WebMethod  
    public String work(String txt);  
}
```


Remote Proxy - JDK only

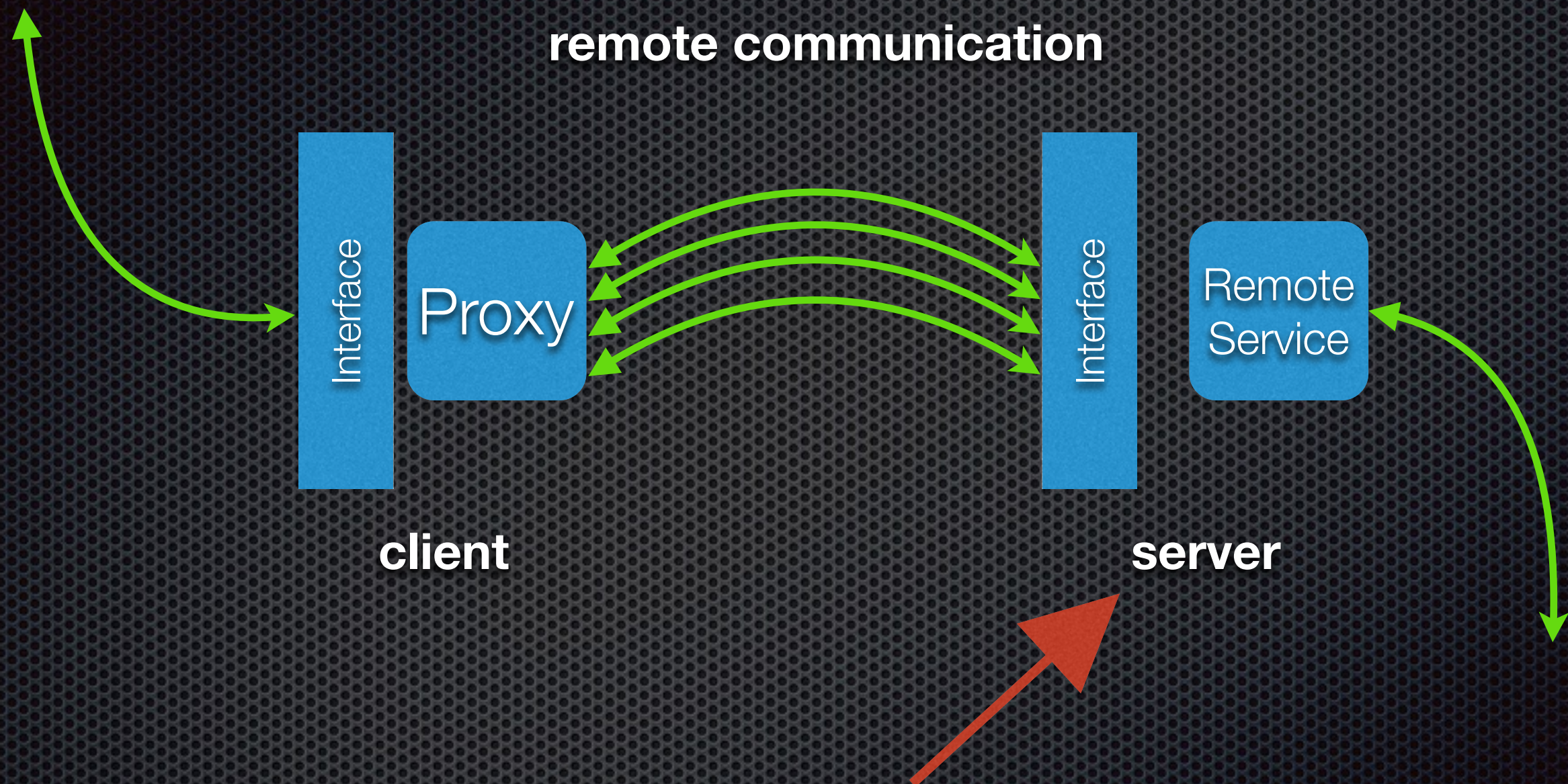
@SvenRuppert




```
@SOAPBinding(style = SOAPBinding.Style.RPC)
@WebService(endpointInterface = "org.rapidpm.Service")
public class ServiceImpl implements Service {
    @Override
    public String work(String txt) {
        return "ServiceImpl - " + txt;
    }
}
```


Remote Proxy - JDK only

@SvenRuppert




```
final String address = "http://localhost:9999/ws/service";  
final ExecutorService threadPool = Executors.newFixedThreadPool(10);
```

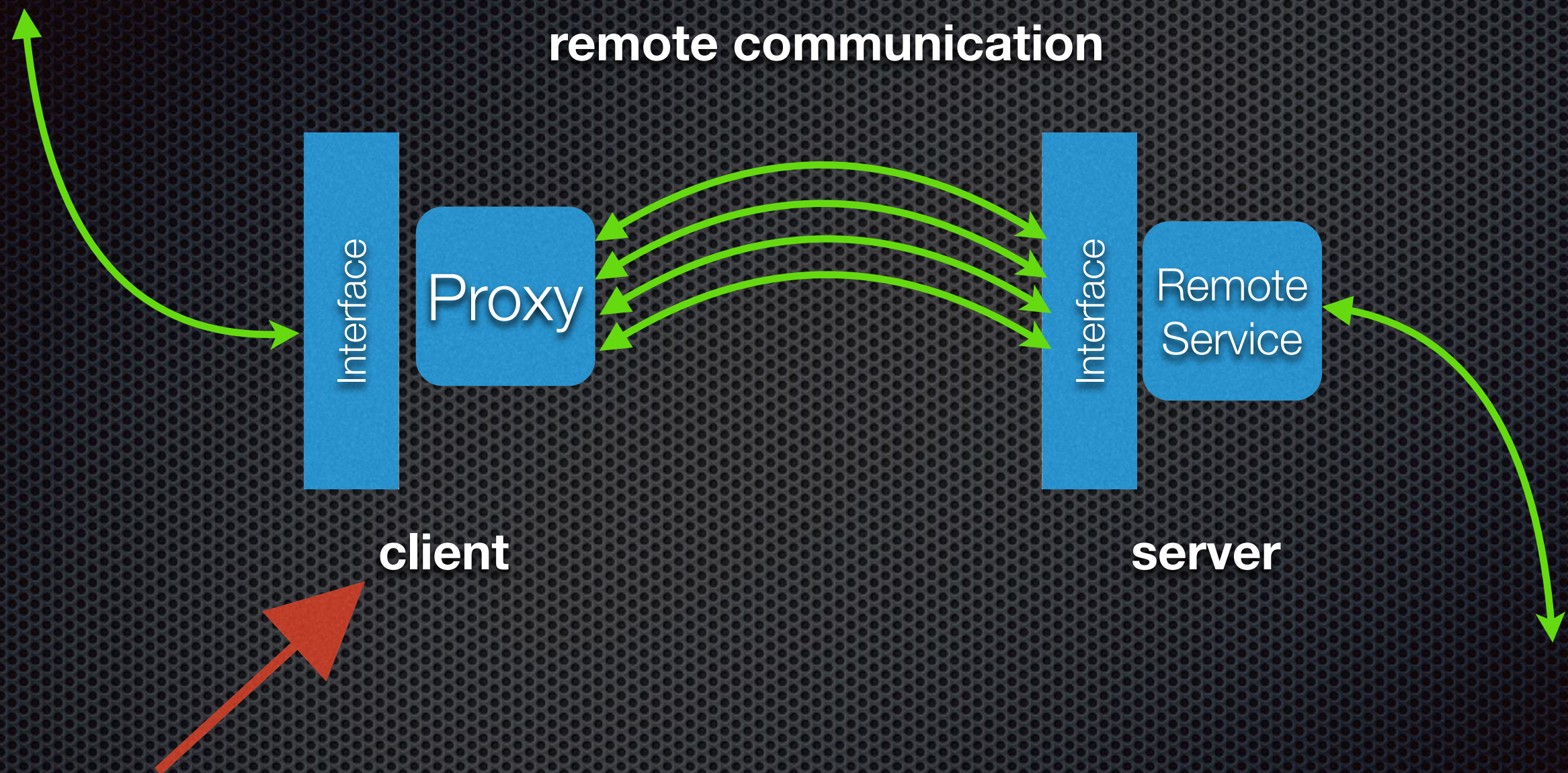
```
final Endpoint endpoint = Endpoint.create(new ServiceImpl());  
endpoint.setExecutor(threadPool);  
endpoint.publish(address);
```



```
System.out.println("endpoint = " + endpoint.isPublished());
```


Remote Proxy - JDK only

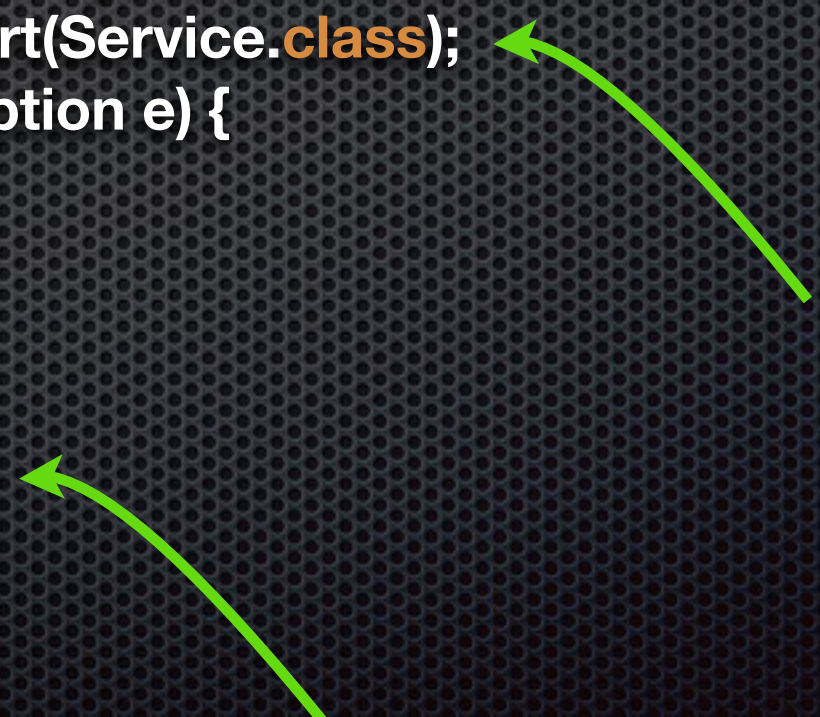
@SvenRuppert

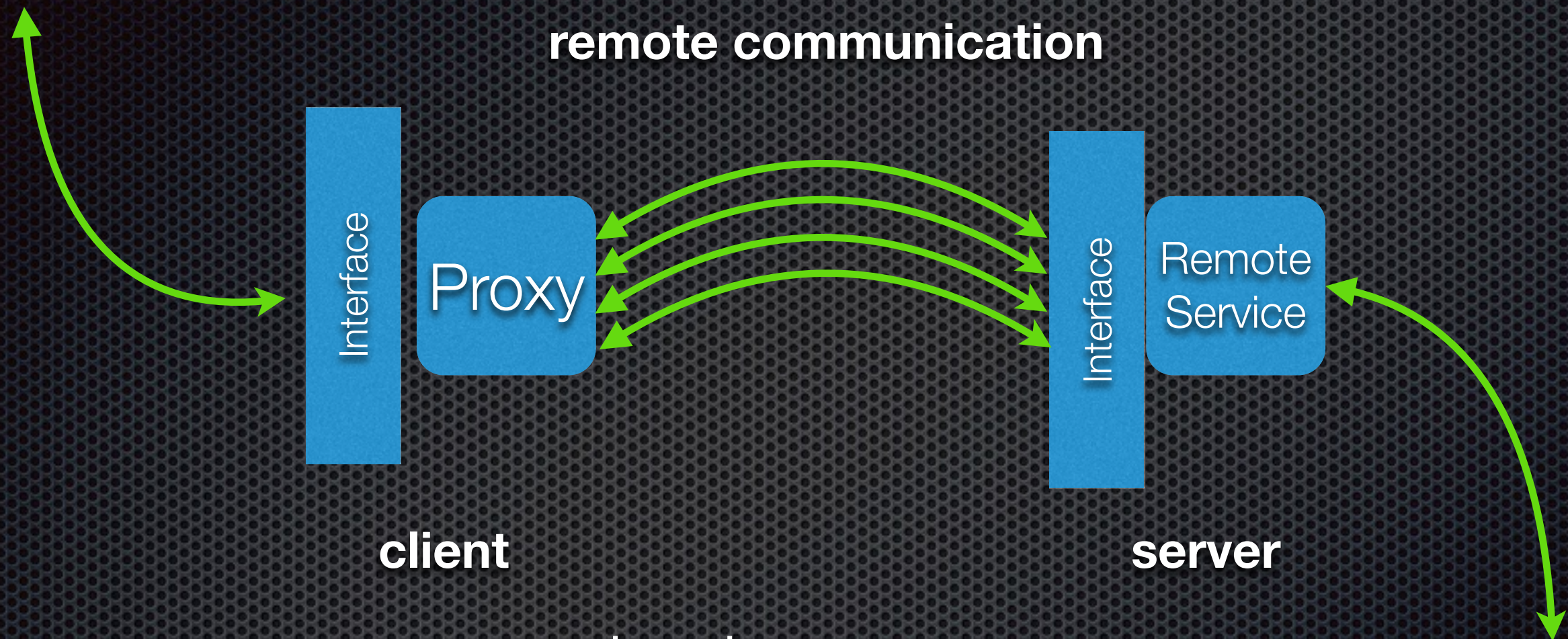


Remote Proxy - JDK only

@SvenRuppert

```
public class ServiceRemoteProxy implements Service {  
  
    private URL url;  
    private Service realSubject;  
    final String namespaceURI = "http://rapidpm.org/";  
    final String localPart = "ServiceImplService";  
  
    public ServiceRemoteProxy() {  
        try {  
            url = new URL("http://localhost:9999/ws/service?wsdl");  
            QName qname = new QName(namespaceURI, localPart);  
            javax.xml.ws.Service service = javax.xml.ws.Service.create(url, qname);  
            realSubject = service.getPort(Service.class);  
        } catch (MalformedURLException e) {  
            e.printStackTrace();  
        }  
    }  
  
    public String work(String txt){  
        return realSubject.work(txt);  
    }  
}
```





- remote communication
- most of this is technology based work
- **goal: developer will see no remote communication**

Virtual Proxy

@SvenRuppert

```
public class ServiceProxy implements Service {  
    private Service service = new ServiceImpl();  
    public String work(String txt) { return service.work(txt); }  
}
```

What could we
change now ?

Virtual Proxy:
create the Delegator later

```
public class VirtualService implements Service {  
    private Service service = null;  
    public String work(String txt) {  
        if(service == null) { service = new ServiceImpl(); }  
        return service.work(txt);  
    }  
}
```



Virtual Proxy

@SvenRuppert

```
public class VirtualService implements Service {  
    private Service service = null;  
    public String work(String txt) {  
        if(service == null) { service = new ServiceImpl(); }  
        return service.work(txt);  
    }  
}
```

This is NOT ThreadSafe

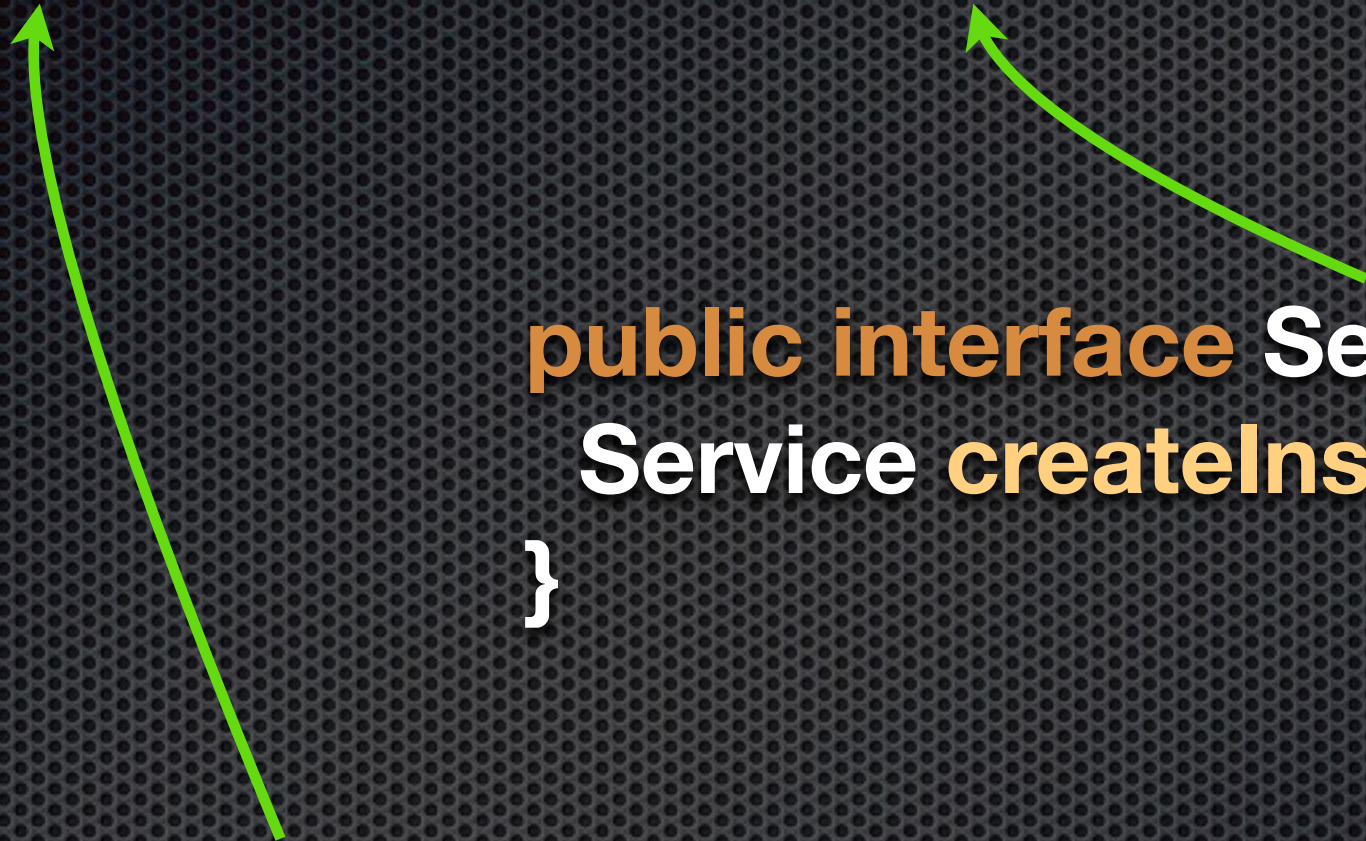
**fixed decision for
an implementation**

**how to combine it with
a FactoryPattern ?**


```
if(service == null) { service = new ServiceImpl(); }
```

```
public interface ServiceFactory {  
    Service createInstance();  
}
```

```
public interface ServiceStrategyFactory {  
    Service realSubject(ServiceFactory factory);  
}
```



Virtual Proxy - Not - ThreadSafe

@SvenRuppert

```
if(service == null) { service = new ServiceImpl(); }
```

```
private ServiceFactory serviceFactory = ServiceImpl::new;
```

```
public class ServiceStrategyFactoryImpl implements ServiceStrategyFactory {  
    Service realSubject;  
    public Service realSubject(final ServiceFactory factory) {  
        if (realSubject == null) {  
            realSubject = factory.createInstance();  
        }  
        return realSubject;  
    }  
}
```


Virtual Proxy - Not - ThreadSafe

@SvenRuppert

```
if(service == null) { service = new ServiceImpl(); }
```

```
public class ServiceProxy implements Service {  
    private ServiceFactory serviceFactory = ServiceImpl::new;  
    private ServiceStrategyFactory strategyFactory = new ServiceStrategyFactoryImpl();  
  
    public String work(String txt) {  
        return strategyFactory.realSubject(serviceFactory).work(txt);  
    }  
}
```


Virtual Proxy - Synchronized

@SvenRuppert

```
if(service == null) { service = new ServiceImpl(); }
```

```
private ServiceFactory serviceFactory = ServiceImpl::new;
```



```
public class ServiceStrategyFactorySynchronized implements ServiceStrategyFactory {  
    Service realSubject;  
    public synchronized Service realSubject(final ServiceFactory factory) {  
        if (realSubject == null) {  
            realSubject = factory.createInstance();  
        }  
        return realSubject;  
    }  
}
```

This is ThreadSafe

Virtual Proxy - Method Scoped

@SvenRuppert

```
if(service == null) { service = new ServiceImpl(); }
```

```
private ServiceFactory serviceFactory = ServiceImpl::new;
```

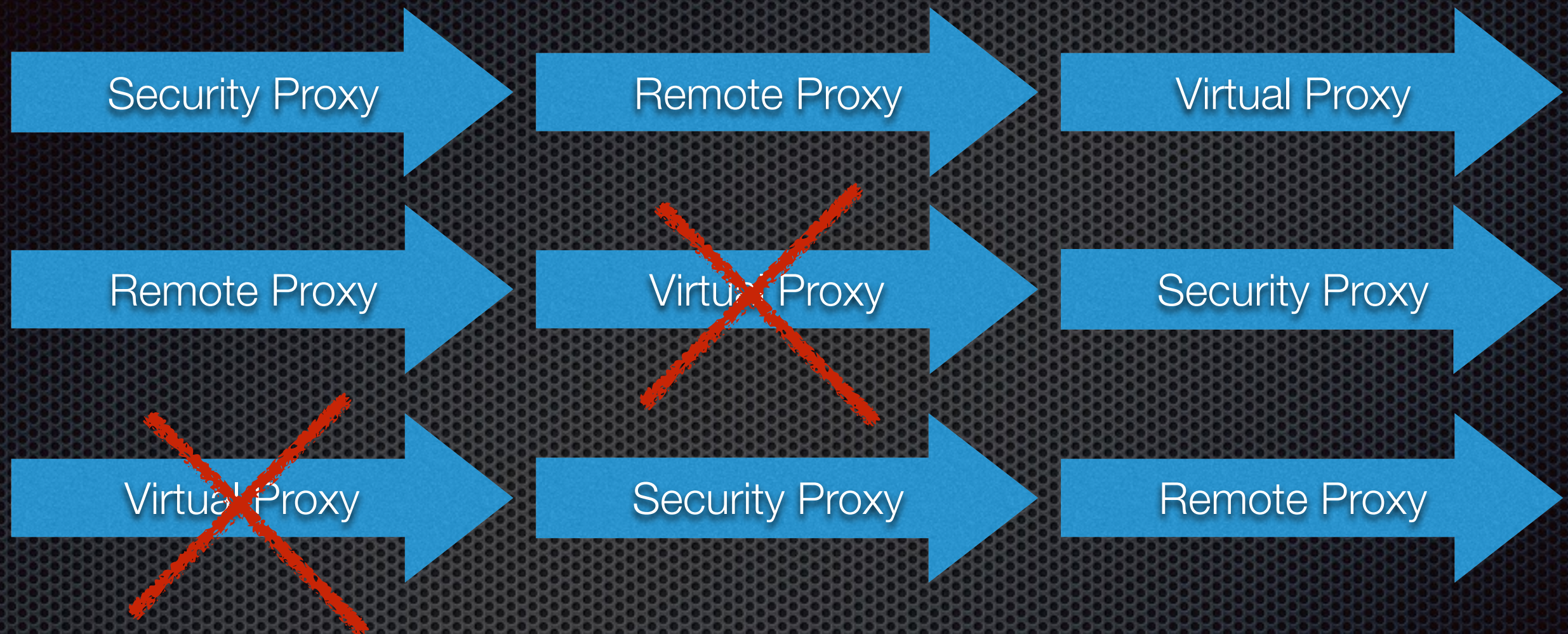


```
public class ServiceStrategyFactoryMethodScoped implements  
ServiceStrategyFactory {  
    public Service realSubject(final ServiceFactory factory) {  
        return factory.createInstance();  
    }  
}
```



Combining Proxies

@SvenRuppert



combining Proxies is easy

VirtualProxies are mostly the last one

SecurityProxies are mostly the first one ?

Combining Proxies - SecureVirtualProxy

@SvenRuppert

Security Proxy

Virtual Proxy

```
public class SecureVirtualProxy implements Service {  
    public SecureProxy() { out.println("SecureProxy " + now()); }  
    private Service service = new VirtualProxy();  
    //SNIPP....  
    public String work(String txt) {  
        if(code.equals("hoppe1")) { return service.work(txt); }  
        return "noooooop";  
    }  
}
```

hard wired proxies

What could we
change now ?

```
public class VirtualProxy implements Service {  
    public VirtualProxy() { out.println("VirtualProxy " + now()); }  
    private Service service = null;  
    public String work(String txt) {  
        if(service == null) { service = new ServiceImpl(); }  
        return service.work(txt);  
    }  
}
```




we need a Generic Proxy
since JDK1.3 we have it
the DynamicProxy

hard wired proxies
What could we
change now ?

prepare for.....

Proxy Deep Dive - DynamicProxies

some words about :

how to use and how to extend....but.. also what is not nice

@SvenRuppert

Dynamic Proxy - Hello World

@SvenRuppert

How to use it ?

java.lang.reflect.Proxy

we need an interface : here Service

we need an implementation : here ServiceImpl

```
Service proxyInstance = Service.class.cast( Proxy.newProxyInstance(
    Service.class.getClassLoader(),
    new Class<?>[]{Service.class},
    new InvocationHandler() {
        private final ServiceImpl service = new ServiceImpl();
        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            return method.invoke(service, args);
        }
    }
    );
```

OK, step by step , please

Dynamic Proxy - Hello World

@SvenRuppert

```
Service proxyInstance = Service.class.cast (  
    Proxy.newProxyInstance (  
        Service.class.getClassLoader(),  
        new Class<?>[]{Service.class},  
        new InvocationHandler() {  
            private final ServiceImpl service = new ServiceImpl();  
            @Override  
            public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {  
                return m.invoke(service, args);  
            }  
        }  
    )  
);
```

how to make a VirtualProxy?

Dynamic Virtual Proxy

@SvenRuppert

```
Service proxyInstance = Service.class.cast (  
    Proxy.newProxyInstance (  
        Service.class.getClassLoader(),  
        new Class<?>[]{Service.class},  
        new InvocationHandler() {  
            private final ServiceImpl service;  
            @Override  
            public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {  
  
                if(service == null) { service = new ServiceImpl(); }  
  
                return m.invoke(service, args);  
            }  
        }  
    )  
);
```



Dynamic Proxy - make it generic

@SvenRuppert

```
Service proxyInstance = Service.class.cast (  
    Proxy.newProxyInstance (  
        Service.class.getClassLoader(),  
        new Class<?>[]{Service.class},  
        new InvocationHandler() {  
            private final ServiceImpl service = new ServiceImpl();  
            @Override  
            public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {  
                return m.invoke(service, args);  
            }  
        }  
    )  
);
```

get it from outside

we will remove it

we will get it from outside

and finally we will call it ProxyGenerator

Dynamic Proxy - make it generic

@SvenRuppert

```
public class ProxyGenerator {  
    public static <P> P makeProxy(Class<P> subject, P realSubject) {  
        Object proxyInstance = Proxy.newProxyInstance(  
            subject.getClassLoader(),  
            new Class<?>[]{subject},  
            new InvocationHandler() {  
                @Override  
                public Object invoke(final Object proxy, final Method m, final Object[] args) throws Throwable {  
                    return m.invoke(realSubject, args);  
                }  
            }  
        );  
        return subject.cast(proxyInstance);  
    }  
}
```


Dynamic Proxy - make it generic

@SvenRuppert

```
public class ProxyGenerator {  
    public static <P> P makeProxy(Class<P> subject, P realSubject) {  
        Object proxyInstance = Proxy.newProxyInstance(  
            subject.getClassLoader(),  
            new Class<?>[]{subject},  
            (proxy, m, args) -> m.invoke(realSubject, args)  
        );  
        return subject.cast(proxyInstance);  
    }  
}
```


Dynamic Proxy - make it generic

@SvenRuppert

with DynamicProxies

- we are writing less (more generic) code
- we can create Virtual-/Remote-/Security Proxies

but how to combine Proxies more generic ?

Dynamic Proxy - make it generic

@SvenRuppert

but how to combine Proxies more generic ?

for this we have to switch from

Factory-Method-Pattern



Builder-Pattern

but how to combine Proxies more generic ?

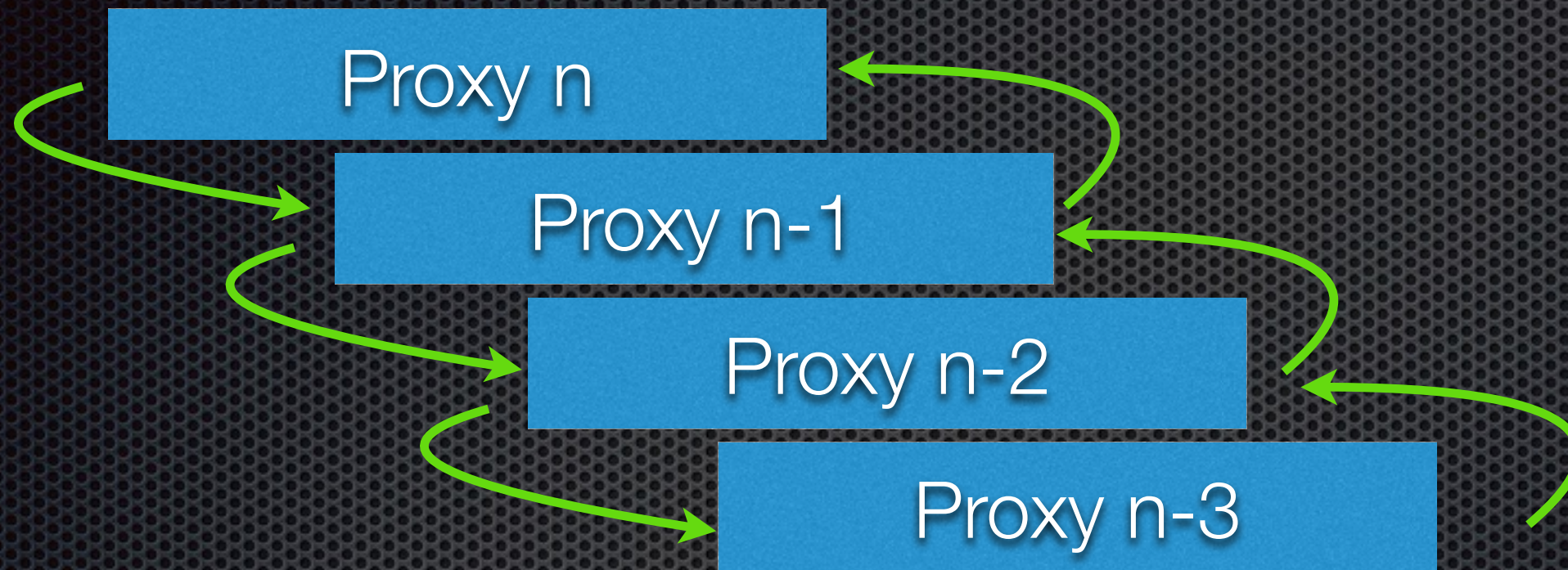


let's think about Two possible ways:

- 1 Security Proxy with 2 Rules and 1 VirtualProxy
- 2 Security Proxies with 1 Rule and 1 VirtualProxy

OK, we need a generic CascadedProxy

OK, we need a generic CascadedProxy



a child must know his parent

first we only add different DynamicProxies

always the same target interface

for example: add n SecurityRules

```
public interface SecurityRule {  
    boolean checkRule();  
}
```

functional interface



```
private List<SecurityRule> securityRules = new ArrayList<>();
```

```
Collections.reverse(securityRules);  
securityRules.forEach(this::build_addSecurityRule);
```


DynamicProxyBuilder

@SvenRuppert

```
Collections.reverse(securityRules);  
securityRules.forEach(this::build_addSecurityRule);
```

```
private ProxyBuilder<I, T> build_addSecurityRule(SecurityRule rule) {  
    final ClassLoader classLoader = original.getClass().getClassLoader();  
    final Class<?>[] interfaces = {clazz};  
    final Object nextProxy = Proxy.newProxyInstance(  
        classLoader, interfaces,  
        new InvocationHandler() {  
            private T original = ProxyBuilder.this.original;  
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
                final boolean checkRule = rule.checkRule();  
                if (checkRule) {  
                    return method.invoke(original, args);  
                } else {  
                    return null;  
                }  
            }  
        });  
    original = (T) clazz.cast(nextProxy);  
    return this;  
}
```

last child

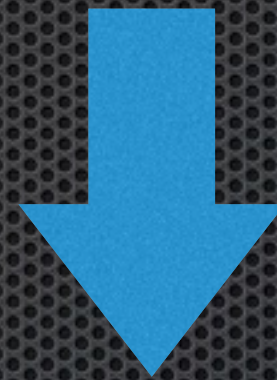
work on child

set child for next level

how this will be used ?


```
final InnerDemoClass original = new InnerDemoClass();  
final InnerDemoInterface demoLogic =  
    ProxyBuilder.createBuilder(InnerDemoInterface.class, original)  
        .addLogging()  
        .addMetrics()  
        .addSecurityRule() -> true)  
        .addSecurityRule() -> true)  
        .build();
```

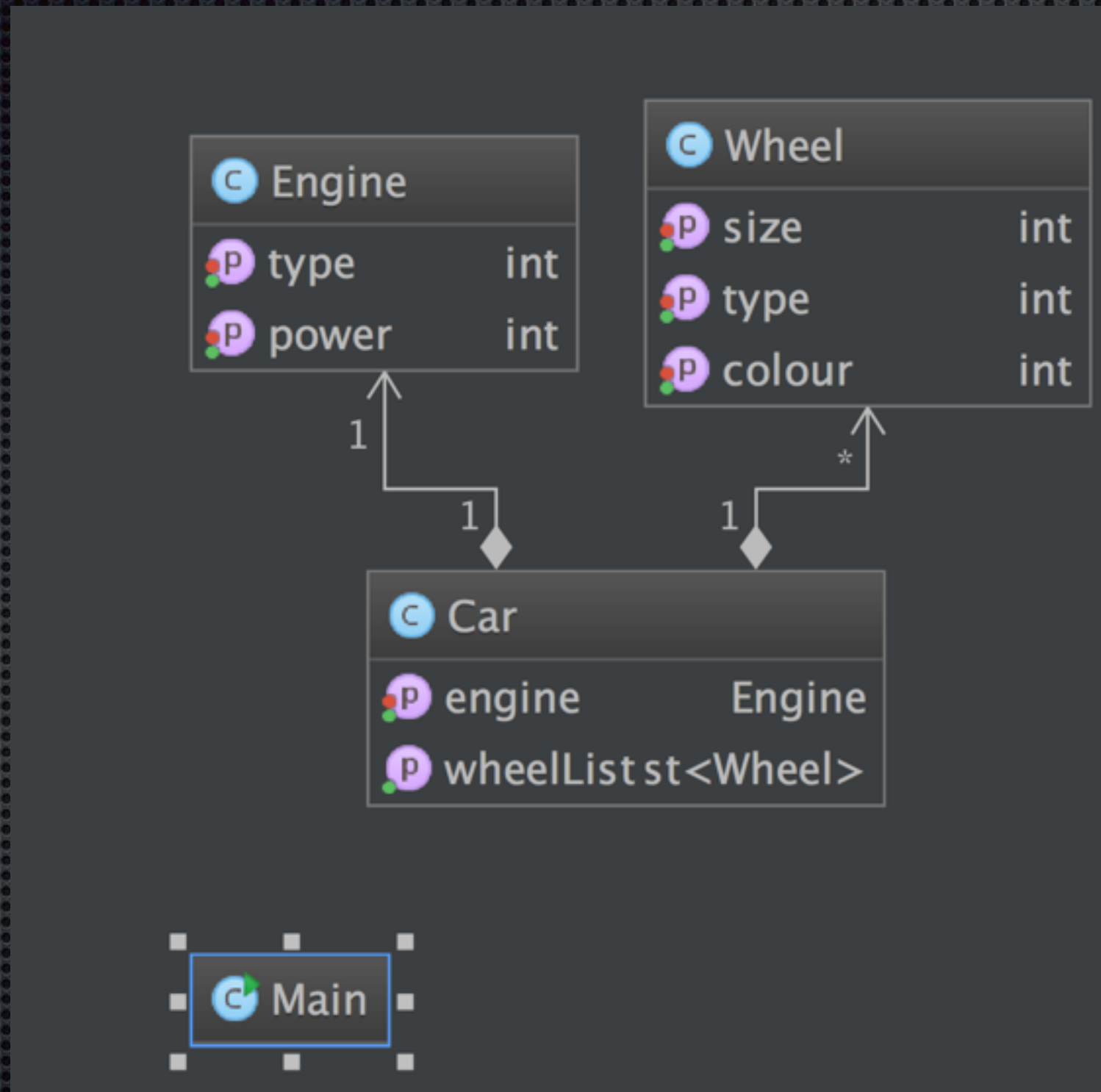

Until now, we had only a cascade of one interface
how to deal with different interfaces ?



we need a generic NestedBuilder

NestedBuilder - The Beginning

@SvenRuppert



NestedBuilder - The Beginning

@SvenRuppert

```
Engine engine = Engine.newBuilder().withPower(100).withType(5).build();
```

```
Car car = Car.newBuilder()  
    .withEngine(engine)  
    .withWheelList(wheels)  
    .build();
```

```
List<Wheel> wheels = new ArrayList<>();  
{  
    wheels.add(wheel1);  
    wheels.add(wheel2);  
    wheels.add(wheel3);  
}
```

```
Wheel wheel1 = Wheel.newBuilder().withType(2).withColour(3).withSize(4).build();  
Wheel wheel2 = Wheel.newBuilder().withType(2).withColour(3).withSize(4).build();  
Wheel wheel3 = Wheel.newBuilder().withType(2).withColour(3).withSize(4).build();
```


NestedBuilder - The Beginning

@SvenRuppert

```
Engine engine = Engine.newBuilder().withPower(100).withType(5).build();
```

```
Wheel wheel1 = Wheel.newBuilder().withType(2).withColour(3).withSize(4).build();
```

```
Wheel wheel2 = Wheel.newBuilder().withType(2).withColour(3).withSize(4).build();
```

```
Wheel wheel3 = Wheel.newBuilder().withType(2).withColour(3).withSize(4).build();
```

```
List<Wheel> wheels = new ArrayList<>();
```

```
    wheels.add(wheel1);
```

```
    wheels.add(wheel2);
```

```
    wheels.add(wheel3);
```

```
Car car = Car.newBuilder()
```

```
    .withEngine(engine)
```

```
    .withWheelList(wheels)
```

```
    .build();
```


NestedBuilder - The Beginning

@SvenRuppert

```
Engine engine = Engine.newBuilder().withPower(100).withType(5).build();
```

```
Wheel wheel1 = Wheel.newBuilder().withType(2).withColour(3).withSize(4).build();  
Wheel wheel2 = Wheel.newBuilder().withType(2).withColour(3).withSize(4).build();  
Wheel wheel3 = Wheel.newBuilder().withType(2).withColour(3).withSize(4).build();
```

```
List<Wheel> wheels = new ArrayList<>();  
// wheels.add(wheel1);  
// wheels.add(wheel2);  
// wheels.add(wheel3);
```

```
List<Wheel> wheelList = WheelListBuilder.newBuilder()  
    .withNewList()  
    .addWheel(wheel1)  
    .addWheel(wheel2)  
    .addWheel(wheel3)  
    .build(); //more robust if you add tests at build()
```

how to combine ?

```
Car car = Car.newBuilder()  
    .withEngine(engine)  
    .withWheelList(wheels)  
    .build();
```


NestedBuilder - The Beginning

@SvenRuppert

```
Engine engine = Engine.newBuilder().withPower(100).withType(5).build();
```

```
Wheel wheel1 = Wheel.newBuilder().withType(2).withColour(3).withSize(4).build();  
Wheel wheel2 = Wheel.newBuilder().withType(2).withColour(3).withSize(4).build();  
Wheel wheel3 = Wheel.newBuilder().withType(2).withColour(3).withSize(4).build();
```

```
List<Wheel> wheelList = WheelListBuilder.newBuilder()  
    .withNewList()  
    .addWheel(wheel1)  
    .addWheel(wheel2)  
    .addWheel(wheel3)  
    .build();
```

how to combine ?

```
Car car = Car.newBuilder()  
    .withEngine(engine)  
    .withWheelList(wheels)  
    .build();
```


NestedBuilder - The Beginning

@SvenRuppert

```
Engine engine = Engine.newBuilder().withPower(100).withType(5).build();
```

```
// Wheel wheel1 = Wheel.newBuilder().withType(2).withColour(3).withSize(4).build();  
// Wheel wheel2 = Wheel.newBuilder().withType(2).withColour(3).withSize(4).build();  
// Wheel wheel3 = Wheel.newBuilder().withType(2).withColour(3).withSize(4).build();
```

```
// List<Wheel> wheelList = WheelListBuilder.newBuilder()  
    .withNewList()  
    .addWheel(wheel1)  
    .addWheel(wheel2)  
    .addWheel(wheel3)  
    .build();
```

```
List<Wheel> wheels = wheelListBuilder  
    .addWheel().withType(1).withSize(2).withColour(2).addWheelToList()  
    .addWheel().withType(1).withSize(2).withColour(2).addWheelToList()  
    .addWheel().withType(1).withSize(2).withColour(2).addWheelToList()  
    .addWheel().withType(1).withSize(2).withColour(2).addWheelToList()  
    .build();
```

```
Car car = Car.newBuilder()  
    .withEngine(engine)  
    .withWheelList(wheels)  
    .build();
```


NestedBuilder - The Beginning

@SvenRuppert

```
Engine engine = Engine.newBuilder().withPower(100).withType(5).build();
```

```
List<Wheel> wheels = wheelListBuilder
```

```
    .addWheel().withType(1).withSize(2).withColour(2).addWheelToList()  
    .addWheel().withType(1).withSize(2).withColour(2).addWheelToList()  
    .addWheel().withType(1).withSize(2).withColour(2).addWheelToList()  
    .addWheel().withType(1).withSize(2).withColour(2).addWheelToList()  
    .build();
```

WheelListBuilder

WheelBuilder

```
Car car = Car.newBuilder()  
    .withEngine(engine)  
    .withWheelList(wheels)  
    .build();
```


NestedBuilder - The Beginning

@SvenRuppert

```
Engine engine = Engine.newBuilder().withPower(100).withType(5).build();
```

```
List<Wheel> wheels = wheelListBuilder
```

```
    .addWheel().withType(1).withSize(2).withColour(2).addWheelToList()  
    .addWheel().withType(1).withSize(2).withColour(2).addWheelToList()  
    .addWheel().withType(1).withSize(2).withColour(2).addWheelToList()  
    .addWheel().withType(1).withSize(2).withColour(2).addWheelToList()  
    .build();
```

```
Car car = Car.newBuilder()  
    .withEngine(engine)  
    .withWheelList(wheels)  
    .build();
```

now we have to combine all

```
Car car = Car.newBuilder()  
    .addEngine().withPower(100).withType(5).done()  
    .addWheels()  
        .addWheel().withType(1).withSize(2).withColour(2).addWheelToList()  
        .addWheel().withType(1).withSize(2).withColour(2).addWheelToList()  
        .addWheel().withType(1).withSize(2).withColour(2).addWheelToList()  
        .addWheel().withType(1).withSize(2).withColour(2).addWheelToList()  
    .done()  
    .build();
```


NestedBuilder - The Pattern

@SvenRuppert

```
public abstract class NestedBuilder<T, V> {  
    public abstract V build()  
    protected T parent;  
  
    public <P extends NestedBuilder<T, V>> P  
    withParentBuilder(T parent) {  
        this.parent = parent;  
        return (P) this;  
    }  
}
```

parent will connect itself...



NestedBuilder - The Pattern

@SvenRuppert

```
public abstract class NestedBuilder<T, V> {
```

```
    public T done() {
```

```
        Class<?> parentClass = parent.getClass();
```

connect itself with parent



```
    try {
```

```
        V build = this.build();
```

```
        String methodname = "with" + build.getClass().getSimpleName();
```

```
        Method method = parentClass.getDeclaredMethod(  
                                                    methodname, build.getClass());
```

```
        method.invoke(parent, build);
```

```
    } catch (NoSuchMethodException
```

```
            | IllegalAccessException | InvocationTargetException e) {  
        e.printStackTrace();
```

```
    }
```

```
}
```


the basic steps in short words

the Parent-Builder will hold the Child-Builder

the Parent-Builder will have a addChild - Method

the Child-Builder will extend the NestedBuilder

**the rest could be generated with a
default Builder-Generator**

NestedBuilder - The Pattern

@SvenRuppert

```
public class Parent {  
    private KidA kidA;  
    private KidB kidB;  
    //snipp.....  
    public static final class Builder {  
        private KidA kidA;  
        private KidB kidB;  
        // to add manually  
        private KidA.Builder builderKidA = KidA.newBuilder().withParentBuilder(this);  
        private KidB.Builder builderKidB = KidB.newBuilder().withParentBuilder(this);  
  
        public KidA.Builder addKidA() {  
            return this.builderKidA;  
        }  
        public KidB.Builder addKidB() {  
            return this.builderKidB;  
        }  
    }  
    //-----  
}
```

connect itself with child

switch to Child-Builder

NestedBuilder - The Pattern

@SvenRuppert

```
public static final class Builder  
    extends NestedBuilder<Parent.Builder, KidA> {
```



only extends on Child-Builder

NestedBuilder - The Pattern

@SvenRuppert

```
Parent build = Parent.newBuilder()
```

```
    .addKidA().withNote("A").done()
```

```
    .addKidB().withNote("B").done()
```

```
    .build();
```

```
System.out.println("build = " + build);
```

and a child could be a parent in the same time

```
Parent build = Parent.newBuilder()
```

```
    .addKidA().withNote("A")
```

```
        .addKidB().withNote("B").done()
```

```
    .done()
```

```
    .build();
```

```
System.out.println("build = " + build);
```

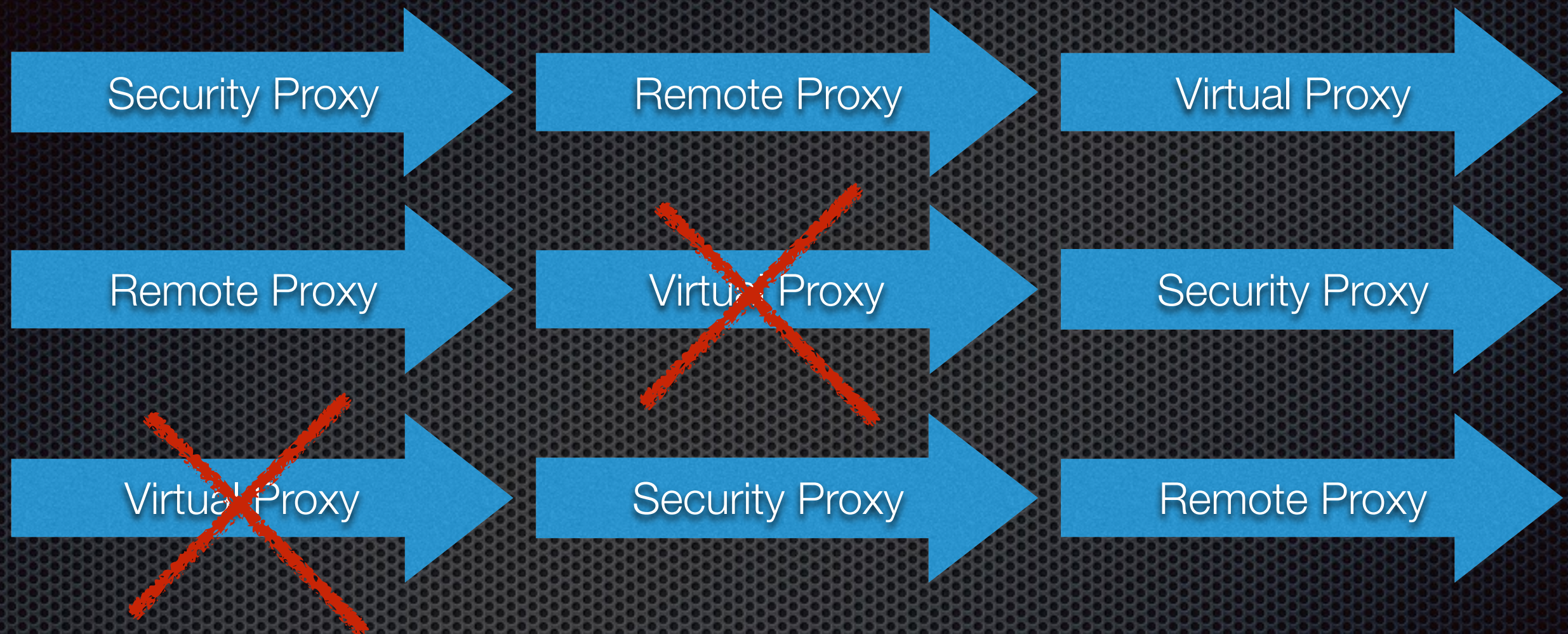

NestedProxyBuilder - The Goal

@SvenRuppert

handwritten , params for diff Proxies

Combining Proxies

@SvenRuppert



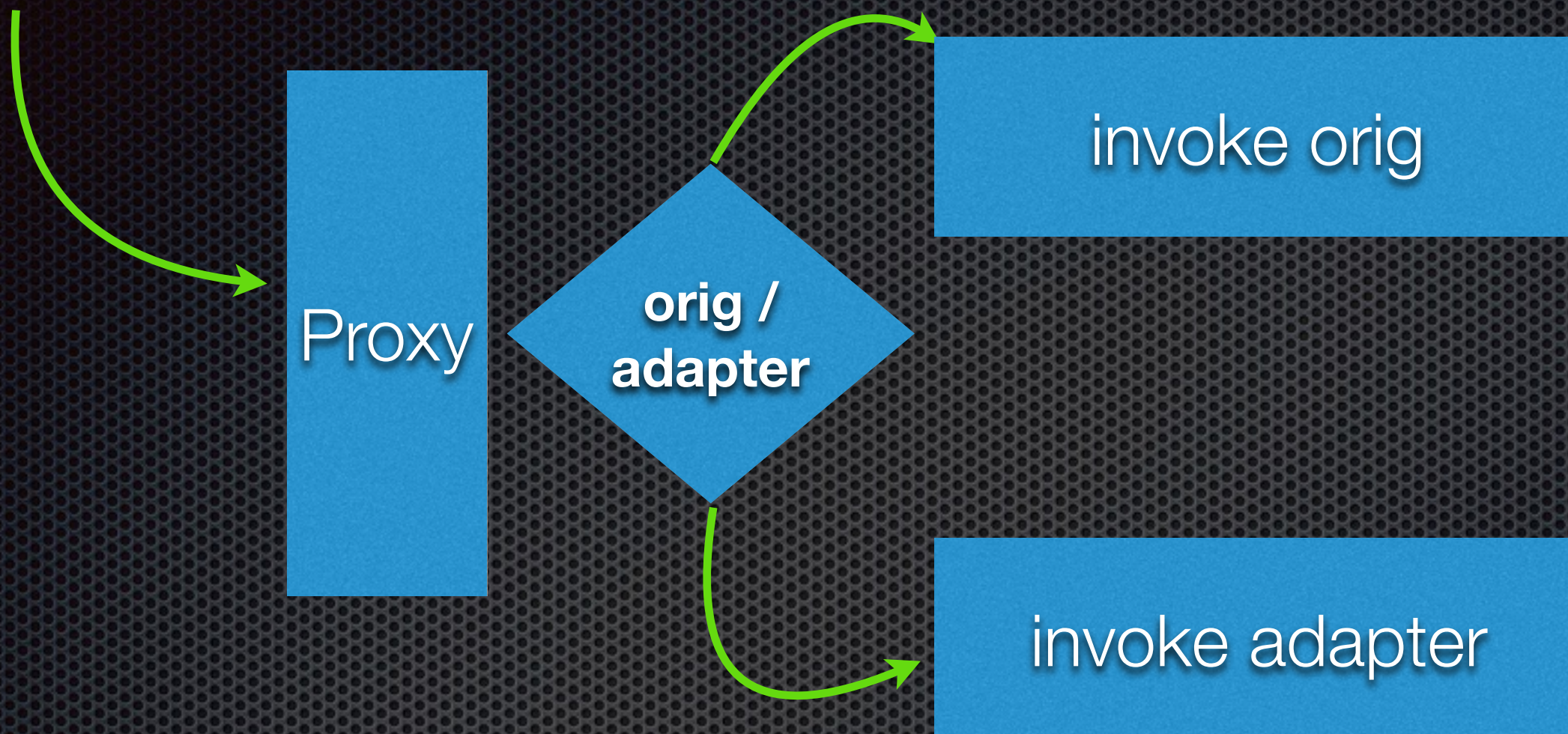
combining Proxies is easy

VirtualProxies are mostly the last one

SecurityProxies are mostly the first one ?

One functionality leads to one Proxy Instance
runtime overhead for every Proxy - but
make the first design simple.. optimize later

the original Version of the DynamicObjectAdapter
was written by Dr. Heinz Kabutz



**Goal: do not need to write an
Adapter with Delegator**

core concept:
switching between method implementations
how to identify a method?


```
public interface Service {  
    String doWork_A();  
}  
  
public static class ServiceImpl implements Service {  
    public String doWork_A() {  
        return ServiceImpl.class.getSimpleName();  
    }  
}
```


how to identify a method?

```
public class MethodIdentifier {  
    private final String name;  
    private final Class[] parameters;  
  
    public MethodIdentifier(Method m) {  
        name = m.getName();  
        parameters = m.getParameterTypes();  
    }  
  
    // we can save time by assuming that we only compare against  
    // other MethodIdentifier objects  
    public boolean equals(Object o) {  
        MethodIdentifier mid = (MethodIdentifier) o;  
        return name.equals(mid.name) &&  
            Arrays.equals(parameters, mid.parameters);  
    }  
    public int hashCode() { return name.hashCode(); }  
}
```



DynamicObjectAdapter - orig. Version

@SvenRuppert

```
private static class DOAInvocationHandler<T extends I, I> implements InvocationHandler {  
    private Map<MethodIdentifier, Method> adaptedMethods = new HashMap<>();  
  
    private T original;    private Object adapter;  
  
    private void addAdapter(Object adapter) {  
        this.adapter = adapter;  
        final Class<?> adapterClass = adapter.getClass();  
        Method[] methods = adapterClass.getDeclaredMethods();  
        for (Method m : methods) {  
            final MethodIdentifier key = new MethodIdentifier(m);  
            adaptedMethods.put(key, m);  
        }  
    }  
  
    public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {  
        //SNIPP  
    }  
}
```


DynamicObjectAdapter - orig. Version

@SvenRuppert

```
private static class DOAInvocationHandler<T extends I, I> implements InvocationHandler {  
    private Map<MethodIdentifier, Method> adaptedMethods = new HashMap<>();  
  
    private T original;    private Object adapter;  
  
    private void addAdapter(Object adapter) {  
        //SNIPP  
    }  
}
```

what could be a problem ?

what could be improved ?

```
public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {  
    try {  
        final MethodIdentifier key = new MethodIdentifier(m);  
        Method other = adaptedMethods.get(key);  
  
        if (other != null) { return other.invoke(adapter, args); }  
        else                { return m.invoke(original, args); }  
    } catch (InvocationTargetException e) { throw e.getTargetException(); }  
}
```


DynamicObjectAdapter - orig. Version

@SvenRuppert

```
private Map<MethodIdentifier, Method> adaptedMethods = new HashMap<>();
```

```
private T original; private Object adapter;
```

```
private void addAdapter(Object adapter) {
```

what could be a problem ?

what could be improved ?

```
public interface Service {  
    String doWork_A();  
}
```

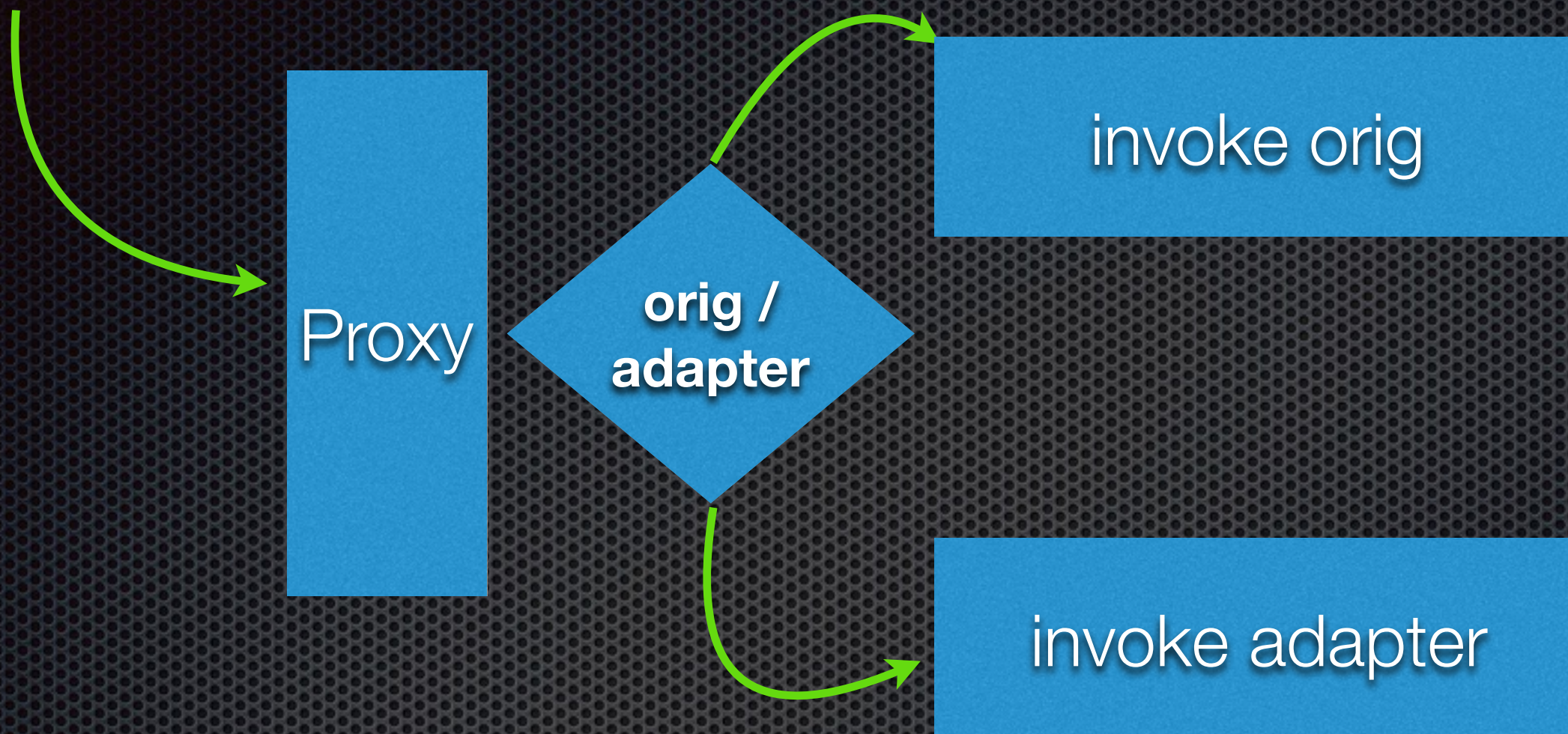
```
public class Adapter_A {  
    public String doWork_A() { .. }  
}
```

```
public class Adapter_B {  
    public String doWork_B() { .. }  
}
```



DynamicObjectAdapter - orig Version

@SvenRuppert



first extension !


```
public interface Service {  
    String doWork_A(String txt);  
    String doWork_B(String txt);  
}  
  
public static class ServiceImpl implements Service {  
  
    public String doWork_A(String txt) {  
        return "doWorkd_A_Original";  
    }  
  
    public String doWork_B(String txt) {  
        return "doWorkd_B_Original";  
    }  
}
```



```
public interface Service {  
    String doWork_A(String txt);  
    String doWork_B(String txt);  
}
```

functional interface



```
public interface ServiceAdapter_A {  
    String doWork_A(String txt);  
}
```

functional interface



```
public interface ServiceAdapter_B {  
    String doWork_B(String txt);  
}
```


DynamicObjectAdapter - ext. Version

@SvenRuppert

```
private Map<MethodIdentifier, Method> adaptedMethods = new HashMap<>();  
private Map<MethodIdentifier, Object> adapters = new HashMap<>();
```

```
private T original;  
private Class<T> target;
```

```
private void addAdapter(Object adapter) {  
    final Class<?> adapterClass = adapter.getClass();  
    Method[] methods = adapterClass.getDeclaredMethods();  
    for (Method m : methods) {  
        final MethodIdentifier key = new MethodIdentifier(m);  
        adaptedMethods.put(key, m);  
        adapters.put(key, adapter);  
    }  
}
```


DynamicObjectAdapter - ext. Version

@SvenRuppert

```
private Map<MethodIdentifier, Method> adaptedMethods = new HashMap<>();  
private Map<MethodIdentifier, Object> adapters = new HashMap<>();
```

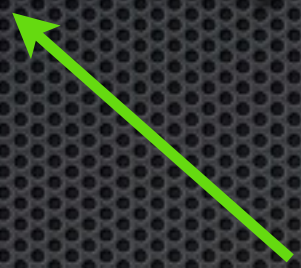
```
private T original;  
private Class<T> target;
```

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
    try {  
        final MethodIdentifier key = new MethodIdentifier(method);  
        Method other = adaptedMethods.get(key);  
  
        if (other != null) {  
            return other.invoke(adapters.get(key), args);  
        } else {  
            return method.invoke(original, args);  
        }  
    } catch (InvocationTargetException e) {  
        throw e.getTargetException();  
    }  
}
```


DynamicObjectAdapter - ext. Version

@SvenRuppert

```
private Map<MethodIdentifier, Method> adaptedMethods = new HashMap<>();  
private Map<MethodIdentifier, Object> adapters = new HashMap<>();  
  
private T original;  
private Class<T> target;
```



if you will use this without a Builder
you can add useless Adapter !!

Add some typed with-Methods

DynamicObjectAdapter - orig Version

@SvenRuppert

Add some typed with-Methods

```
public AdapterBuilder<T> withDoWork_A(ServiceAdapter_A adapter) {  
    addAdapter(adapter);  
    return this;  
}
```

remember

```
public interface Service {  
    String doWork_A(String txt);  
    String doWork_B(String txt);  
}
```

```
public interface ServiceAdapter_A {  
    String doWork_A(String txt);  
}
```


How to use this?

```
Service service = AdapterBuilder.<Service>newBuilder()  
    .withTarget(Service.class)  
    .withOriginal(new ServiceImpl())  
    .withDoWork_A((txt) -> txt + "_part")  
    .build();
```


How to use this **like a mock** ?

```
Service service = AdapterBuilder.<Service>newBuilder()  
    .withTarget(Service.class)  
    .withOriginal(null)  
    .withDoWork_A((txt) -> txt + "_part")  
    .build();
```


generated DynamicObjectAdapterBuilder @SvenRuppert

To much boring code to write

but we want to have a type-safe version

if wrong... compile must break

What could we
change now ?

remember

```
public interface Service {  
    String doWork_A(String txt);  
    String doWork_B(String txt);  
}
```

```
public interface ServiceAdapter_A {  
    String doWork_A(String txt);  
}
```

```
public interface ServiceAdapter_B {  
    String doWork_B(String txt);  
}
```

functional interface



generated DynamicObjectAdapterBuilder @SvenRuppert

use Annotation Processing

create a marker like

@DynamicObjectAdapterBuilder

@Retention(RetentionPolicy.*SOURCE*)

@Target(ElementType.*TYPE*)

public @interface DynamicObjectAdapterBuilder {
}

public class AnnotationProcessor **extends** AbstractProcessor

generated DynamicObjectAdapterBuilder @SvenRuppert

use Annotation Processing

with every compile you will get actual

Functional-Interfaces for the Adapters

generated AdapterBuilder

to avoid the latency of the dynamic Proxy

we could implement **static ObjectAdapter**

the basic idea is the same as we had
with the **DynamicObjectAdapter**

split the interface into Functional Interfaces

create for every Functional Interface an attribute

at runtime: if an attribute
is available invoke the method

otherwise use the delegator

StaticObjectAdapter

@SvenRuppert

```
public interface Service {  
    String doWorkA(String txt);  
    String doWorkB(String txt);  
}
```

functional interface



```
public interface Service_DoWorkA_Adapter{  
    String doWorkA(final String txt);  
}  
public interface Service_DoWorkB_Adapter{  
    String doWorkB(final String txt);  
}
```


StaticObjectAdapter

@SvenRuppert

```
public class ServiceStaticObjectAdapter implements Service {  
    private Service service;  
    private Service_DoWorkA_Adapter doWorkAAdapter;  
    private Service_DoWorkB_Adapter doWorkBAdapter;  
  
    public String doWorkA(final String txt) {  
        if (this.doWorkAAdapter != null) return this.doWorkAAdapter.doWorkA(txt);  
        return service.doWorkA(txt)  
    }  
}
```

We can generate
this !

generated StaticObjectAdapter

@SvenRuppert

with AnnotationProcessing

split the interface with **@StaticObjectAdapter**

write functional interfaces

generate the StaticObjectAdapter

mark the Adapter with **@IsObjectAdapter**

prepare for.....

Proxy Deep Dive - StaticVirtualProxy - at Runtime

some words about :

how it works..and what you can do with this..

@SvenRuppert

generate Static Proxies at Runtime

@SvenRuppert

this is based on the
Newsletter Nr **180** and Nr **181**
from **Dr. Heinz Kabutz**

<http://www.javaspecialists.eu/archive/Issue180.html>

<http://www.javaspecialists.eu/archive/Issue181.html>

generate Static Proxies at Runtime

@SvenRuppert

questions are:

How to compile In-Memory ?
What is the right ClassLoader ?

How to compile In-Memory ?

you will need the **JavaCompiler** from tools.jar

a holder for the SourceCode

```
public class GeneratedJavaSourceFile extends SimpleJavaFileObject {
```

```
    private CharSequence javaSource;
```

the generated source code



```
    public GeneratedJavaSourceFile(String className, CharSequence javaSource) {  
        super(URL.create(className + ".java"), Kind.SOURCE);  
        this.javaSource = javaSource;  
    }
```

```
    public CharSequence getCharContent(boolean ignoreEncodeErrors)  
        throws IOException {  
        return javaSource;  
    }  
}
```


How to compile In-Memory ?

a holder for the ByteCode

```
public class GeneratedClassFile extends SimpleJavaFileObject {  
    private final ByteArrayOutputStream outputStream = new ByteArrayOutputStream();  
  
    public GeneratedClassFile() {  
        super(URI.create("generated.class"), Kind.CLASS);  
    }  
  
    public OutputStream openOutputStream() {  
        return outputStream;  
    }  
  
    public byte[] getClassAsBytes() {  
        return outputStream.toByteArray();  
    }  
}
```

the generated ByteCode

How to compile In-Memory ?

a ForwardingJavaFileManager

will force the
JavaCompiler to write
to this instance

```
public class GeneratingJavaFileManager  
    extends ForwardingJavaFileManager<JavaFileManager> {
```

```
    private final GeneratedClassFile gcf;
```

```
    public GeneratingJavaFileManager(  
        StandardJavaFileManager sjfm,  
        GeneratedClassFile gcf) {
```

the generated ByteCode

```
        super(sjfm);  
        this.gcf = gcf;  
    }
```

```
    public JavaFileObject getJavaFileForOutput(  
        Location location, String className,  
        JavaFileObject.Kind kind, FileObject sibling)
```

```
        throws IOException {  
        return gcf;  
    }  
}
```


How to compile In-Memory ?

the compile task itself

```
GeneratedJavaSourceFile gjsf = new GeneratedJavaSourceFile(className, javaSource);  
DiagnosticCollector<JavaFileObject> dc = new DiagnosticCollector<>();  
JavaCompiler jc = ToolProvider.getSystemJavaCompiler();  
StandardJavaFileManager stdFileManager = jc.getStandardFileManager(dc, null, null);  
GeneratedClassFile gcf = new GeneratedClassFile();  
GeneratingJavaFileManager fileManager  
    = new GeneratingJavaFileManager(stdFileManager, gcf);  
  
JavaCompiler.CompilationTask task  
    = jc.getTask(null, fileManager, dc, null, null, singletonList(gjsf));  
  
return task.call();
```


What is the right `ClassLoader` ?

could be: **public** `Unsafe.defineClass()`

or: **private static** `Proxy.defineClass0()`

both versions are
depending on the
selected JVM

but this solution will have
no roundtrip over the hard disc

will create and compile
Proxies at runtime

prepare for.....

Proxy Deep Dive - Summary

what we reached, what we could do now

@SvenRuppert

we got type-safe ProxyBuilder

..type-safe generated DynamicObjectAdapter

..at runtime we could change the chain of Proxies

..could be used for
(Dynamic)

Dependency Injection Implementations

If you are interested...

have a look at **RapidPM** on github

rapidpm-proxybuilder

rapidpm-dynamic-cdi

rapidpm-microservice

or contact me ;-) @SvenRuppert

Thank You !!!