Java Specialists in Action

Dr Heinz Kabutz

The Java Specialists Newsletter

heinz@javaspecialists.co.za

http://www.javaspecialists.co.za

Java Specialists in Action

- Using dynamic proxies to write less code

Background – Who Am I ?

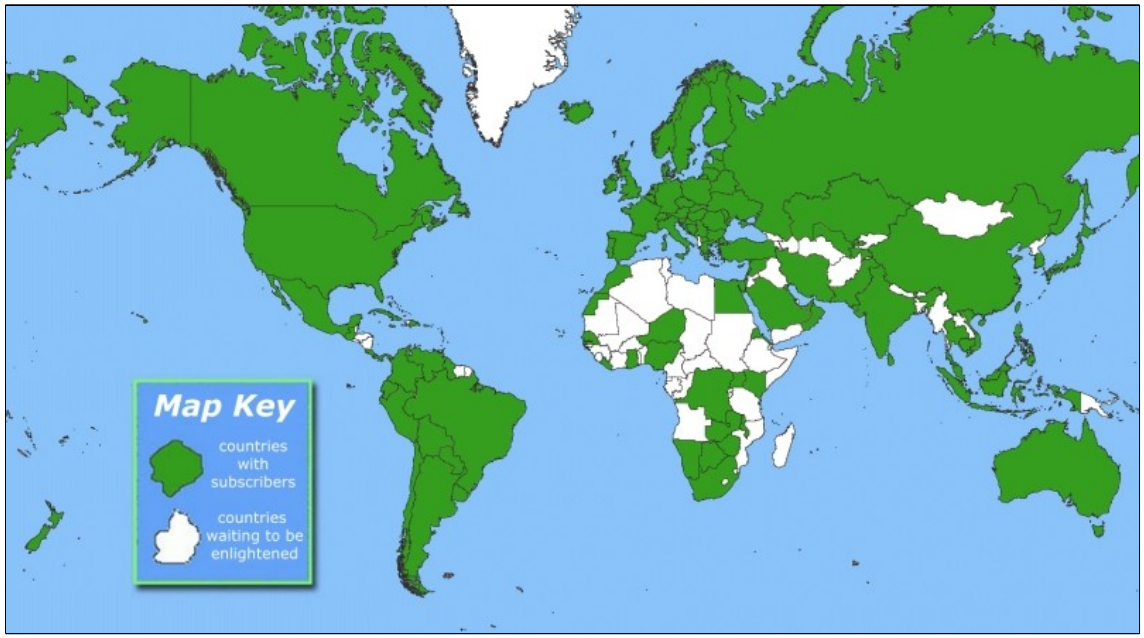
- **Heinz Kabutz**
  - Born in Africa, Cape Town
  - PhD Computer Science from University of Cape Town
    - University famous for first heart transplant
  - Relocating to a Greek island on 20th October 06
  - Java Champion

Background – What do I do ?

- Program on ordinary Java projects
  - Since 1997, several 500k+ LOC systems
- Java Code Reviews
  - Onsite interviews, Java quality inspection
  - 5 day consulting jobs
- Advanced Java Training
  - Design Patterns, Java 5, Introduction to Java
  - Now offered in Norway through Bouvet
    - http://www.bouvet.no/kurs

The Java Specialists' Newsletter

- Advanced topics
  - 30 000 readers in 112 countries



  - Please subscribe by sending an email to subscribe@javaspecialists.co.za

Questions

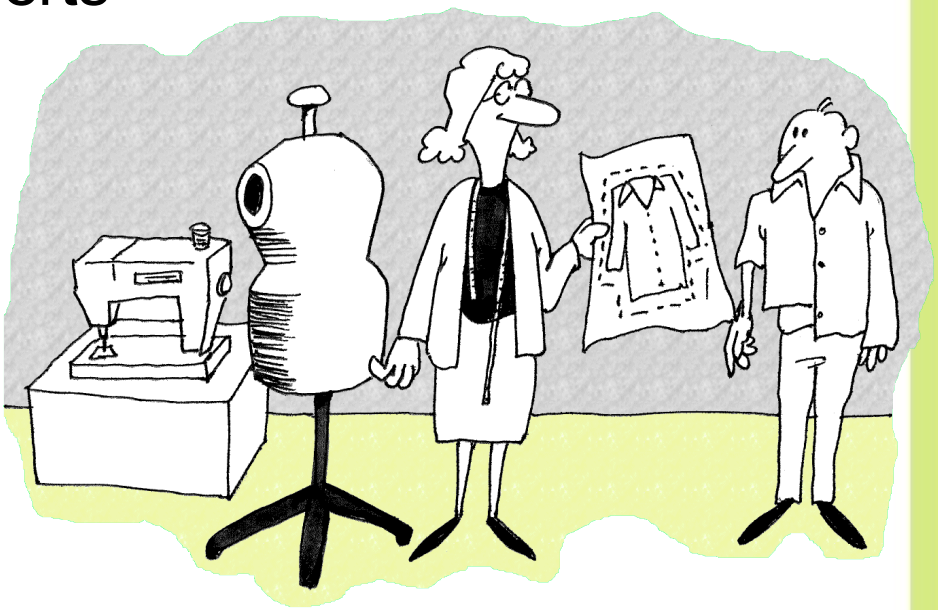
- Please interrupt me with questions!
  - Or write them down to ask at the end
- There *are* some stupid questions
  - They are the ones you didn't ask
  - Once you've asked them, they are not stupid anymore
- The more you ask, the more interesting the talk is

Introduction to Topic

- In this talk, we will look at:
  - Design Patterns
  - Dynamic Proxies in Java
  - Soft, Weak and Strong references
  - Some Java 5 features
- For additional free topics:
  - The Java™ Specialists' Newsletter
    - http://www.javaspecialists.co.za
  - And find out how
    `"hi there".equals("cheers!") == true`

Design Patterns

- Mainstream of OO landscape, offering us:
  - View into brains of OO experts
  - Quicker understanding of existing designs
    - e.g. Visitor pattern used by Annotation Processing Tool
  - Improved communication between developers
  - Readjust “thinking mistakes”



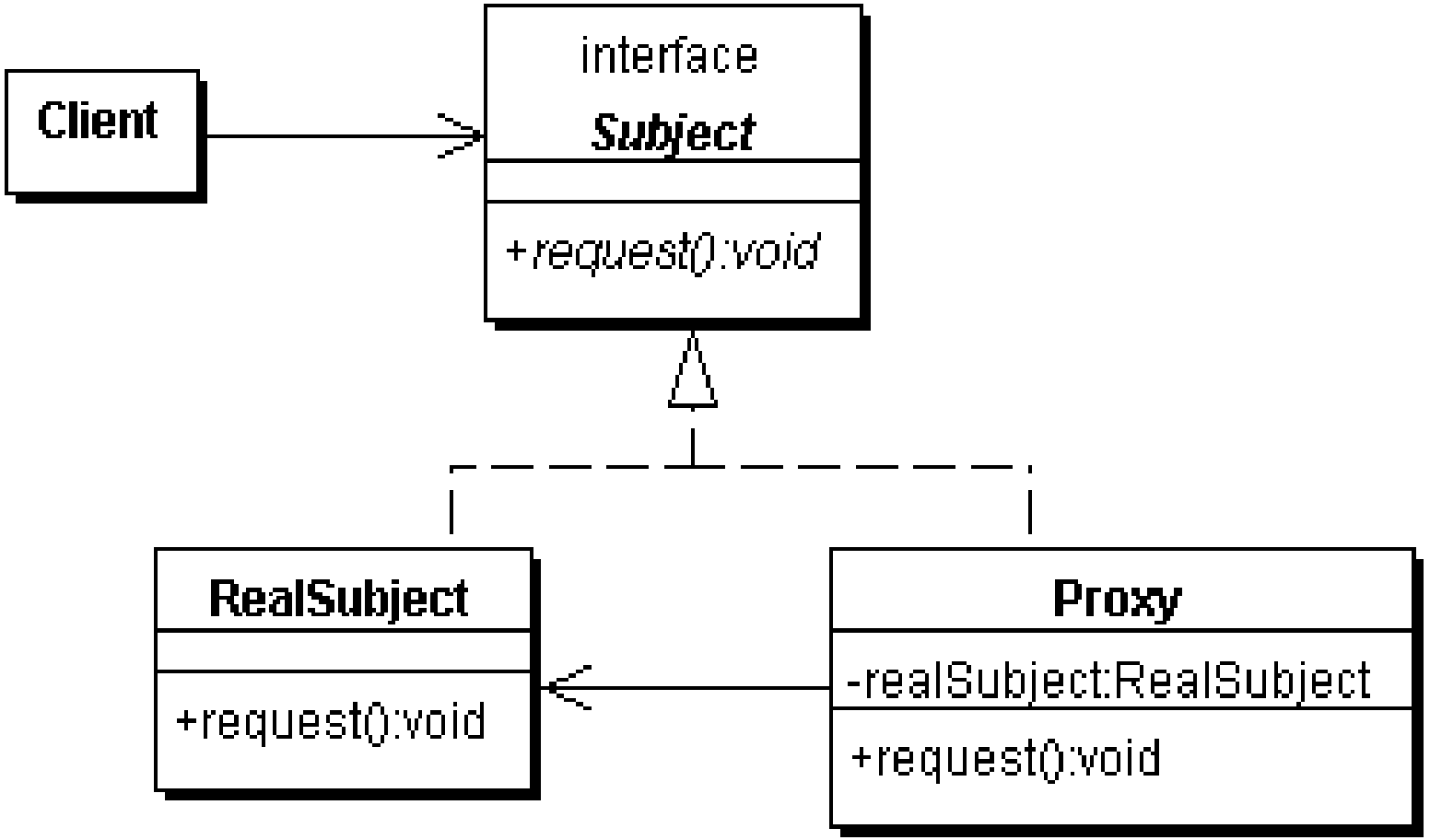
Vintage Wines

- Software Design is like good red wine
  - At first, quality of wine does not matter
    - As long as it has the right effect
  - With experience, you discern difference
  - As you become a connoisseur you experience the various textures you didn't notice before
    - Grown on the north slope in Italy on clay ground
- Warning: Once you are hooked, you will no longer be satisfied with inferior designs

Proxy Pattern

- Intent [GoF95]
  - Provide a surrogate or placeholder for another object to control access to it.

Proxy Structure



Types of Proxies in GoF

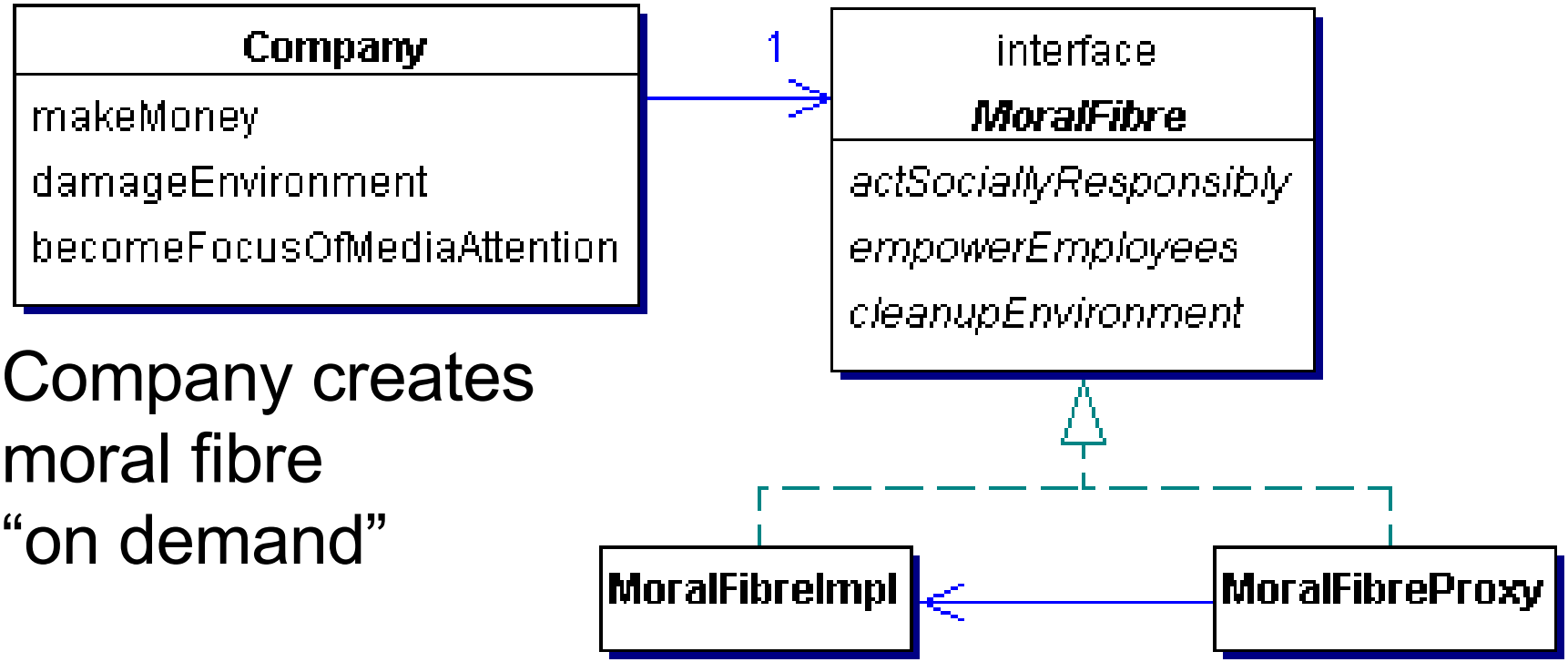
We will focus on this type

- Virtual Proxy
  - creates expensive objects on demand
- Remote Proxy
  - provides a local representation for an object in a different address space
- Protection Proxy
  - controls access to original object

Approaches to writing proxies

- Handcoded
  - Only for the very brave … or foolish
- Autogenerated code
  - RMI stubs and skeletons created by rmic
- Dynamic proxies
  - Available since JDK 1.3
  - Dynamically creates a new class at runtime
  - Flexible and easy to use

Model for example



- Company creates moral fibre "on demand"

```
public class Company {
  // set in constructor ...
  private final MoralFibre moralFibre;

  public void becomeFocusOfMediaAttention() {
    System.out.println("Look how good we are...");
    cash -= moralFibre.actSociallyResponsibly();
    cash -= moralFibre.cleanupEnvironment();
    cash -= moralFibre.empowerEmployees();
  }

  @Override
  public String toString() {
    Formatter formatter = new Formatter();
    formatter.format("%s has $ %.2f", name, cash);
    return formatter.toString();
  }
}
```

```
public class MoralFibreImpl implements MoralFibre {
  // very expensive to create moral fibre!
  private byte[] costOfMoralFibre = new byte[900*1000];

  { System.out.println("Moral Fibre Created!"); }
  // AIDS orphans
  public double actSociallyResponsibly() {
    return costOfMoralFibre.length / 3;
  }
  // shares to employees
  public double empowerEmployees() {
    return costOfMoralFibre.length / 3;
  }
  // oiled sea birds
  public double cleanupEnvironment() {
    return costOfMoralFibre.length / 3;
  }
}
```

Handcoded Proxy

- Usually results in a lot of effort
- Good programmers have to be lazy
  - DRY principle
    - Don’t repeat yourself
- Shown just for illustration

```
public class MoralFibreProxy implements MoralFibre {
  private MoralFibreImpl realSubject;
  private MoralFibre realSubject() {
    if (realSubject == null) { // need synchronization
      realSubject = new MoralFibreImpl();
    }
    return realSubject;
  }
  public double actSociallyResponsibly() {
    return realSubject().actSociallyResponsibly();
  }
  public double empowerEmployees() {
    return realSubject().empowerEmployees();
  }
  public double cleanupEnvironment() {
    return realSubject().cleanupEnvironment();
  }
}
```

```
import static java.util.concurrent.TimeUnit.SECONDS;

public class WorldMarket0 {
  public static void main(String[] args) throws
  Exception {
    Company maxsol = new Company("Maximum Solutions",
        1000 * 1000, new MoralFibreProxy());
    SECONDS.sleep(2); // better than Thread.sleep();
    maxsol.makeMoney();
    System.out.println(maxsol);
    SECONDS.sleep(2);
    maxsol.damageEnvironment();
    System.out.println(maxsol);
    SECONDS.sleep(2);
    maxsol.becomeFocusOfMediaAttention();
    System.out.println(maxsol);
  }
}
```

```
Oh goodie!
Maximum Solutions has $ 2000000.00
Oops, sorry about that oilspill...
Maximum Solutions has $ 8000000.00
Look how good we are...
Moral Fibre Created!
Maximum Solutions has $ 7100000.00
```

Dynamic Proxies

- Handcoded proxy flawed
  - Previous approach broken – what if toString() is called?
  - Fixing synchronization problems would need to be done everywhere
- Allows you to write a method call handler
  - Is invoked every time any method is called on interface
- Easy to use
  - But, seriously underused feature of Java

But First, References

- We want to release references when possible
  - Soft, Weak and Strong references offer different benefits
  - Works in conjunction with proxies
  - However, references are not transparent

Strong, Soft and Weak References

- Java 1.2 introduced concept of soft and weak references
- Weak reference is released when no strong reference is pointing to the object
- Soft reference can be released, but will typically only be released when memory is low
  - Works correctly since JDK 1.4

Object Adapter Pattern – Pointers

- References are not transparent
- We make them more transparent by defining a Pointer interface
  - Can then be Strong, Weak or Soft

```
public interface Pointer<T> {
   void set(T t);
   T get();
}
```

Strong Pointer

- Simply contains a strong reference to object
- Will never be garbage collected

```
public class StrongPointer<T>
    implements Pointer<T> {
  private T t;
  public void set(T t) { this.t = t; }
  public T get()       { return t; }
}
```

Reference Pointer

- Abstract superclass to either soft or weak reference pointer

```
import java.lang.ref.Reference;
public abstract class RefPointer<T>
    implements Pointer<T> {
  private Reference<T> ref;
  protected void set(Reference<T> ref) {
    this.ref = ref;
  }
  public T get() {
    return ref == null ? null : ref.get();
  }
}
```

Soft and Weak Reference Pointers

- Contains either soft or weak reference to object
- Will be garbage collected later

```
import java.lang.ref.SoftReference;
public class SoftPointer<T>
    extends RefPointer<T> {
  public void set(T t) {
    set(new SoftReference<T>(t));
  }
}

import java.lang.ref.WeakReference;
public class WeakPointer<T> extends RefPointer<T> {
  public void set(T t) {
    set(new WeakReference<T>(t));
  }
}
```

Using Turbocharged enums

- We want to define enum for these pointers
- But, we don't want to use switch
  - Switch and multi-conditional if-else are anti-OO
  - Rather use inheritance, strategy or state patterns
- Enums allow us to define abstract methods
  - We implement these in the enum values themselves

```
public enum PointerType {
  STRONG { // these are anonymous inner classes
    public <T> Pointer<T> make() { // note generics
      return new StrongPointer<T>();
    }
  },
  WEAK {
    public <T> Pointer<T> make() {
      return new WeakPointer<T>();
    }
  },
  SOFT {
    public <T> Pointer<T> make() {
      return new SoftPointer<T>();
    }
  };

  public abstract <T> Pointer<T> make();
}
```

PointerTest Example

```
public void test(PointerType type) {
  System.out.println("Testing " + type + " Pointer");
  MyObject obj = new MyObject(type.toString());
  Pointer<MyObject> pointer = type.make();
  pointer.set(obj);
  System.out.println(pointer.get());
  obj = null;
  forceGC();
  System.out.println(pointer.get());
  forceOOME();
  System.out.println(pointer.get());
  System.out.println();
}
```

Danger – References

- References put additional strain on GC
- Only use with large objects
- Memory space preserving measure
  - But can severely impact on performance
- Even empty finalize() methods can cause OutOfMemoryError
  - Additional step in GC that runs in separate thread

Defining a Dynamic Proxy

- We make a new instance of an interface class using java.lang.reflect.Proxy:

```
Object o = Proxy.newProxyInstance(
  Thread.currentThread().getContextClassLoader(),
  new Class[]{ interface to implement },
  implementation of InvocationHandler
);
```

- The result is an instance of ***interface to implement***

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

public class VirtualProxy<T> implements InvocationHandler {
  private final Pointer<T> realSubjectPointer;
  private final Object[] constrParams;
  private final Constructor<? extends T> subjectConstr;

  public VirtualProxy(Class<? extends T> realSubjectClass,
                      Class[] constrParamTypes,
                      Object[] constrParams,
                      PointerType pointerType) {
    try {
      subjectConstr = realSubjectClass.
        getConstructor(constrParamTypes);
      realSubjectPointer = pointerType.make();
    } catch (NoSuchMethodException e) {
      throw new IllegalArgumentException(e);
    }
    this.constrParams = constrParams;
  }
```

```
  public Object invoke(Object proxy, Method method,
                       Object[] args) throws Throwable {
    T realSubject;
    synchronized (this) {
      realSubject = realSubjectPointer.get();
      if (realSubject == null) {
        realSubject = subjectConstr.newInstance(
          constrParams);
        realSubjectPointer.set(realSubject);
      }
    }
    return method.invoke(realSubject, args);
  }
}
```

- Whenever any method is invoked on the proxy object, it gets the real subject from the Pointer and creates it if necessary

A word about synchronization

- We need to **synchronize** whenever we check the value of the pointer
  - Otherwise several realSubject objects could be created
- We can synchronized on “this”
  - No one else will have a pointer to the object
- Double-checked locking idiom broken pre-Java 5
  - It now works if you make the field **`volatile`**
  - Easier to get **`synchronized`** correct than **`volatile`**

Proxy Factory

- To simplify our client code, we define a Proxy Factory:

```
@SuppressWarnings("unchecked") // be careful of this!
public class ProxyFactory {
  public static <T> T virtualProxy(Class<T> subject) {
    //  figure out realSubject class and delegate ...
  }

  public static <T> T virtualProxy(Class<T> subject,
    PointerType type) { ... }

  public static <T> T virtualProxy(Class<T> subject,
    Class<? extends T> realSubjectClass,
    Class[] constrParamTypes,
    Object[] constrParams, PointerType type) { ... }
}
```

Proxy Factory

- We will just show the main ProxyFactory method:
  - The other methods send default values to this one

```
public class ProxyFactory {
  public static <T> T virtualProxy(Class<T> subject,
      Class<? extends T> realSubjectClass,
      Class[] constrParamTypes,
      Object[] constrParams, PointerType type) {
    return (T) Proxy.newProxyInstance(
        Thread.currentThread().getContextClassLoader(),
        new Class[] { subject },
        new VirtualProxy<T>(realSubjectClass,
            constrParamTypes, constrParams, type));
  }
}
```

```
import static com.maxoft.proxy.ProxyFactory.virtualProxy;
import static java.util.concurrent.TimeUnit.SECONDS;

public class WorldMarket1 {
  public static void main(String[] args) throws Exception {
     Company maxsol = new Company("Maximum Solutions",
        1000 * 1000, virtualProxy(MoralFibre.class));
    SECONDS.sleep(2);
    maxsol.makeMoney();
    System.out.println(maxsol);
    SECONDS.sleep(2);
    maxsol.damageEnvironment();
    System.out.println(maxsol);
    SECONDS.sleep(2);
    maxsol.becomeFocusOfMediaAttention();
    System.out.println(maxsol);
  }
}
```

```
Oh goodie!
Maximum Solutions has $ 2000000.00
Oops, sorry about that oilspill...
Maximum Solutions has $ 8000000.00
Look how good we are...
Moral Fibre Created!
Maximum Solutions has $ 7100000.00
```

- Weak Pointer is cleared when we don't have a strong ref

```
Company maxsol = new Company("Maximum Solutions",
   1000000, virtualProxy(MoralFibre.class, WEAK));
SECONDS.sleep(2);
maxsol.damageEnvironment();
maxsol.becomeFocusOfMediaAttention();
```

Oops, sorry about that oilspill...
Look how good we are...
Moral Fibre Created!
Oops, sorry about that oilspill...
Look how good we are...
Moral Fibre Created!

```
// short term memory...
System.gc();
SECONDS.sleep(2);
maxsol.damageEnvironment();
maxsol.becomeFocusOfMediaAttention();
```

- Soft Pointer more appropriate

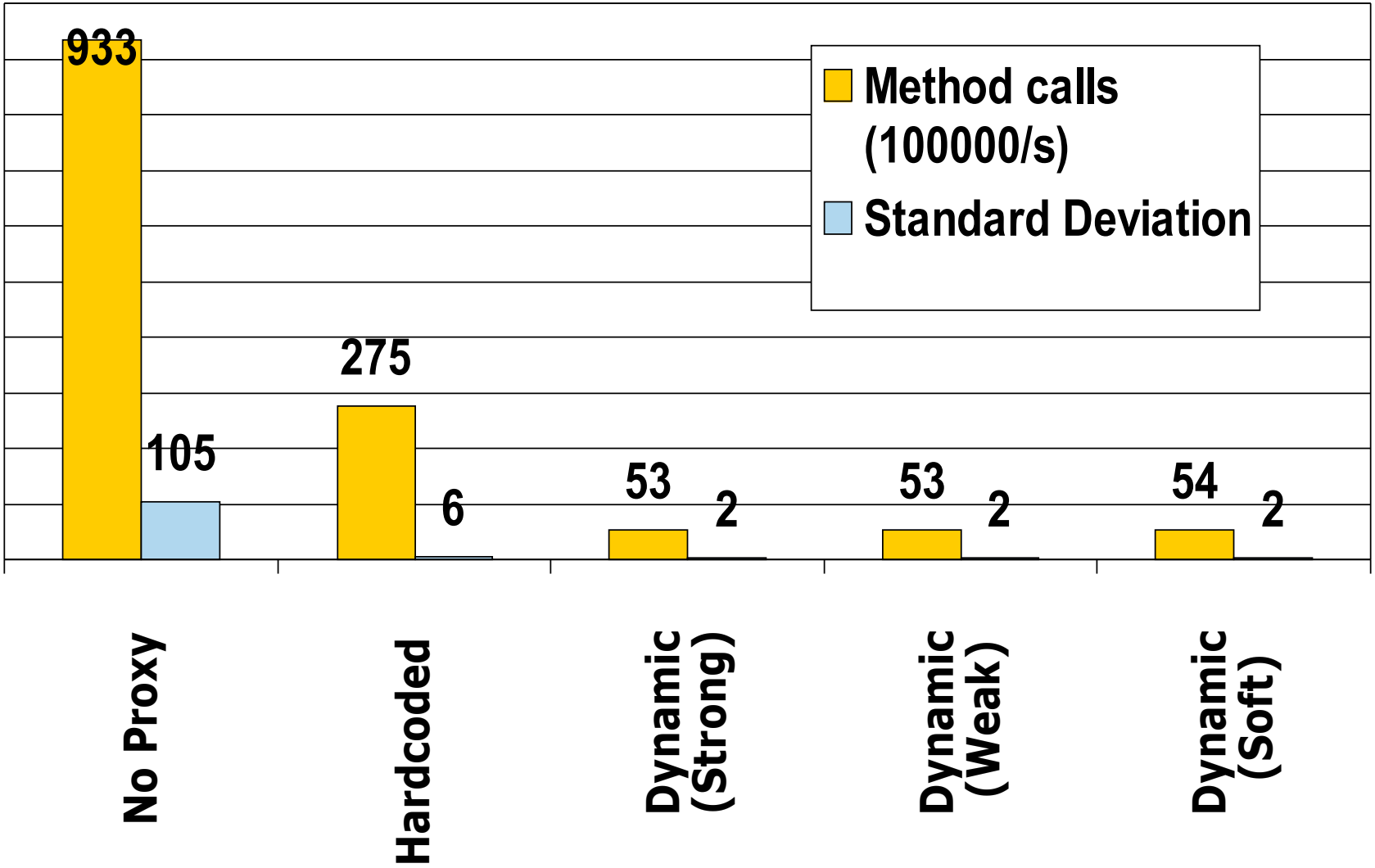
```
Company maxsol = new Company("Maximum Solutions", 1000000,
      virtualProxy(MoralFibre.class, SOFT));
SECONDS.sleep(2);
maxsol.damageEnvironment();
maxsol.becomeFocusOfMediaAttention();

System.gc(); // ignores soft pointer
SECONDS.sleep(2);
maxsol.damageEnvironment();
maxsol.becomeFocusOfMediaAttention();

forceOOME(); // clears soft pointer
SECONDS.sleep(2);
maxsol.damageEnvironment();
maxsol.becomeFocusOfMediaAttention();
}
private static void forceOOME() {
  try {byte[] b = new byte[1000000000];}
  catch (OutOfMemoryError error) { System.err.println(error); }
}
```

Oops, sorry about that oilspill...
Look how good we are...
Moral Fibre Created!
Oops, sorry about that oilspill...
Look how good we are...
java.lang.OutOfMemoryError:
Java heap space
Oops, sorry about that oilspill...
Look how good we are...
Moral Fibre Created!

Performance of Dynamic Proxies

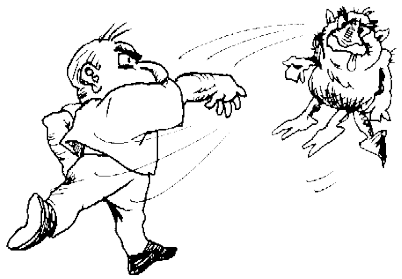


Analysis of Performance Results

- Consider performance in real-life context
  - How often does a method need to get called per second?
  - What contention are you trying to solve – CPU, IO or memory?
    - Probably the wrong solution for CPU bound contention
- Big deviation for “No Proxy” – probably due to HotSpot compiler inlining method call.

Virtual Proxy Gotchas

- Be careful how you implement equals()
  - Should always be *symmetric (from JavaDocs)*:
    - For any non-null reference values x and y, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`
- Exceptions
  - General problem with proxies
    - Local interfaces vs. remote interfaces in EJB
  - Were checked exceptions invented on April 1st ?



Checkpoint

- We've looked at the concept of a *Virtual Proxy* based on the GoF pattern
- We have seen how to implement this with dynamic proxies (since JDK 1.3)
- We have also looked at Soft and Weak refs
- Lastly, we were unsurprised that dynamic proxy performs worse than handcoded proxy

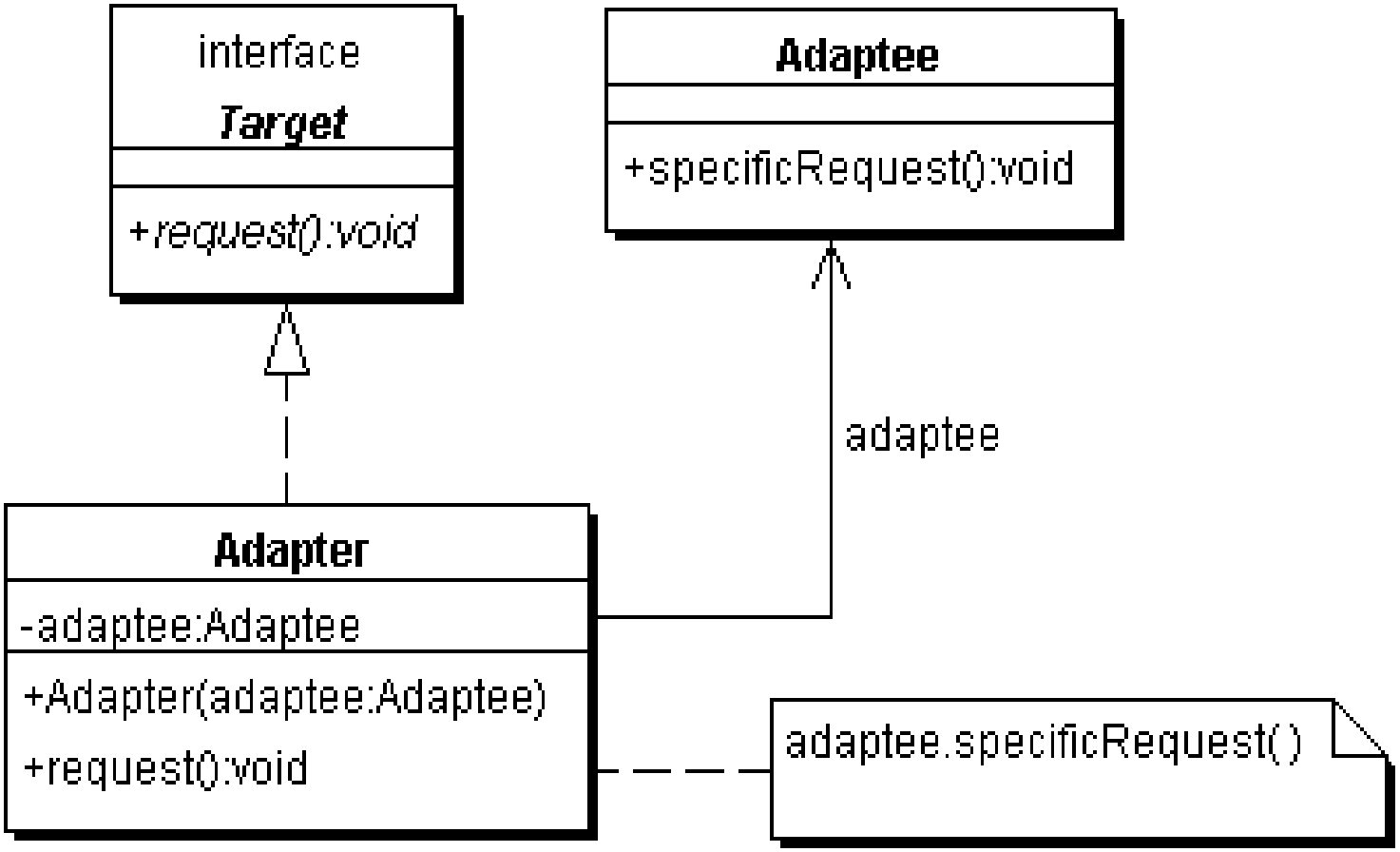
Further uses of Dynamic Proxy

- Protection Proxy
  - Only route call when caller has correct security context
    - Similar to the “Personal Assistant” pattern
- Dynamic Decorator or Filter
  - We can add functions dynamically to an object
  - See newsletter # 34
  - Disclaimer: a bit difficult to understand

Dynamic Object Adapter

- Based on Adapter pattern by GoF
- Plain Object Adapter has some drawbacks:
  - Sometimes you want to adapt an interface, but only want to override some methods
  - E.g. java.sql.Connection
- Structurally, the patterns Adapter, Proxy, Decorator and Composite are almost identical

Object Adapter Structure (GoF)



- We delegate the call if the adapter has a method with this signature
- Objects adaptee and adapter can be of any type

```
public Object invoke(Object proxy, Method method,
                     Object[] args) throws Throwable {
  try {
    // find out if the adapter has this method
    Method other = adaptedMethods.get(
      new MethodIdentifier(method));
    if (other != null) { // yes it has
      return other.invoke(adapter, args);
    } else { // no it does not
      return method.invoke(adaptee, args);
    }
  } catch (InvocationTargetException e) {
    throw e.getTargetException();
  }
}
```

- The ProxyFactory now gets a new method:

```
public class ProxyFactory {
   public static <T> T adapt(Object adaptee,
                             Class<T> target,
                             Object adapter) {
    return (T) Proxy.newProxyInstance(
      Thread.currentThread().getContextClassLoader(),
        new Class[]{target},
        new DynamicObjectAdapter<T>(
          adapter, adaptee));
  }
}
```

- Client can now adapt interfaces very easily

```
import static com.maxoft.proxy.ProxyFactory.*;

// ...

Connection con = DriverManager.getConnection("...");
Connection con2 = adapt(con, Connection.class,
  new Object() {
    public void close() {
      System.out.println("No, do not close connection");
    }
  });
```

- For additional examples of this technique, see The Java Specialists' Newsletter # 108
  - http://www.javaspecialists.co.za

Benefits of Dynamic Proxies

- Write once, use everywhere
- Single point of change
- Elegant coding on the client
  - Esp. combined with static imports & generics
- Slight performance overhead
  - But view that in context of application

Demo

- Short demonstration using Dynamic Virtual Proxy for new interface

Conclusion

- Thank you very much for listening to me ☺
- In my experience, Dynamic Proxies are easy to use
- Look for applications where they are appropriate

Java Specialists in Action

Dr Heinz Kabutz

The Java Specialists Newsletter

heinz@javaspecialists.co.za

http://www.javaspecialists.co.za