Java Specialists in Action

Dr Heinz Kabutz

The Java Specialists' Newsletter

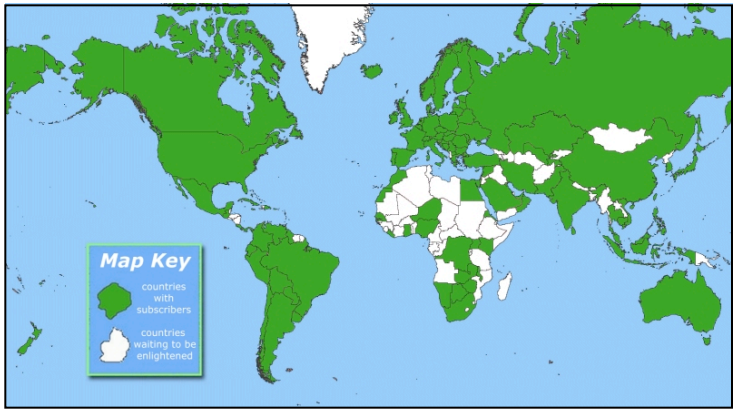
http://www.javaspecialists.co.za

Java Specialists in Action

- **Using dynamic proxies to write less code**

Background

- **Heinz Kabutz**
  - *The Java Specialists' Newsletter*
    - 20 000 readers in 111 countries
  - Sun Java Champion
  - Java programmer since 1997
    - Worked on large Java systems
      - 500 000 – 1 000 000 LOC
  - Taught Java to hundreds of developers
    - Java Patterns Course
    - Java 5 Delta Course
    - http://javaspecialists.co.za/courses



maximum solutions
java training
and home of The Java™ Specialists' Newsletter

Java Training Courses
Java Standard

Specialists in Object Orientation

Home | Java Newsletter | Java Courses | Inhouse Courses | Java Trainers | 10 Good Reasons | Make Enquiry

Welcome to The Java™ Specialists' Newsletter

Strategy Pattern with Generics [Issue 123] The Strategy Pattern is elegant in its simplicity. With this pattern, we should try to convert intrinsic state to extrinsic, to allow sharing of strategy objects. It gets tricky when each strategy object needs a different set of information in order to do its work. In this newsletter, we look at how we can use Java 5 Generics to pass the correct subtype of the context into each strategy object.

Java Courses

Imagine being trained in Java™ and object orientation by the authors of The Java™ Specialists' Newsletter and Java™ Champion Dr Heinz Kabutz. To add to our **inhouse Java courses** we are proud to announce the launch of our **Java Courses on Crete** with it's beautiful beaches, warm weather and great local food.

Latest Java News

Code Java at the speed of light. Modern IDEs have revolutionised the way in which we are able to churn out code. Sadly most programmers are held back... Dr Heinz Kabutz was honoured to be invited to present two sessions at TheServerSide **Java Symposium** from the 23rd to the 25th of March 2006 at Caesar's Palace in Las Vegas.

Newsletter Archive

Over 120 issues archived by date and topic. You are welcome to join our community of 20,000 java programmers in over 100 countries who receive **The Java Specialists' Newsletter** every month.

**"Dr Heinz Max Kabutz publishes an Advanced Java newsletter.
Not for the uninitiated, but I find something fascinating in every issue."**
Bruce Eckel, Author of Thinking in Java

Courselinks
- Java Intro
- Tiger
- Patterns
- Performance
- Schedules
- Inhouse

#1 Java Champ in Africa

Subscriber Countries

Subscribe to the Newsletter | Find out about our java courses | Site Map | Site Design by Catch22 Marketing

java newletter archive	java courses	student comments	java standard course	java 5 delta course	Google Site Map
java performance course	java for managers	delphi patterns	java champion	ruby courses	java trainers

Questions

- **Please please please please ask questions!**
- **There *are* some stupid questions**
  - **They are the ones you didn't ask**
  - **Once you've asked them, they are not stupid anymore**
- **Assume that if you didn't understand something that it was my fault**
- **The more you ask, the more interesting the talk will be**

Introduction to Topic

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

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**
  - **Readjusting of "thinking mistakes" by developers**

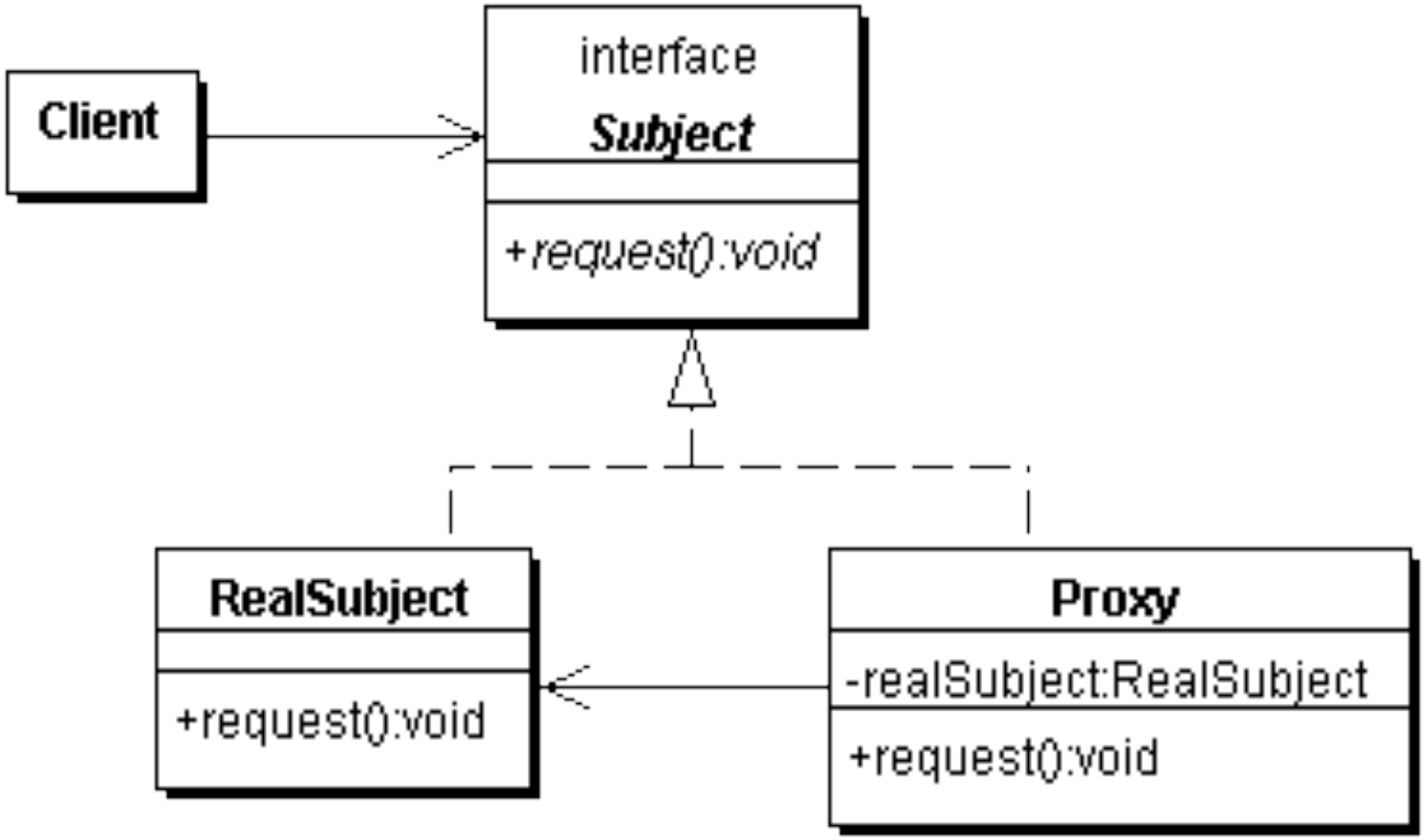
Vintage Wines

- **Design Patterns are like good red wine**
  - **You cannot appreciate them at first**
  - **As you study them you learn the difference between *plonk* and vintage, or bad and good designs**
  - **As you become a connoisseur you experience the various textures you didn't notice before**
- **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

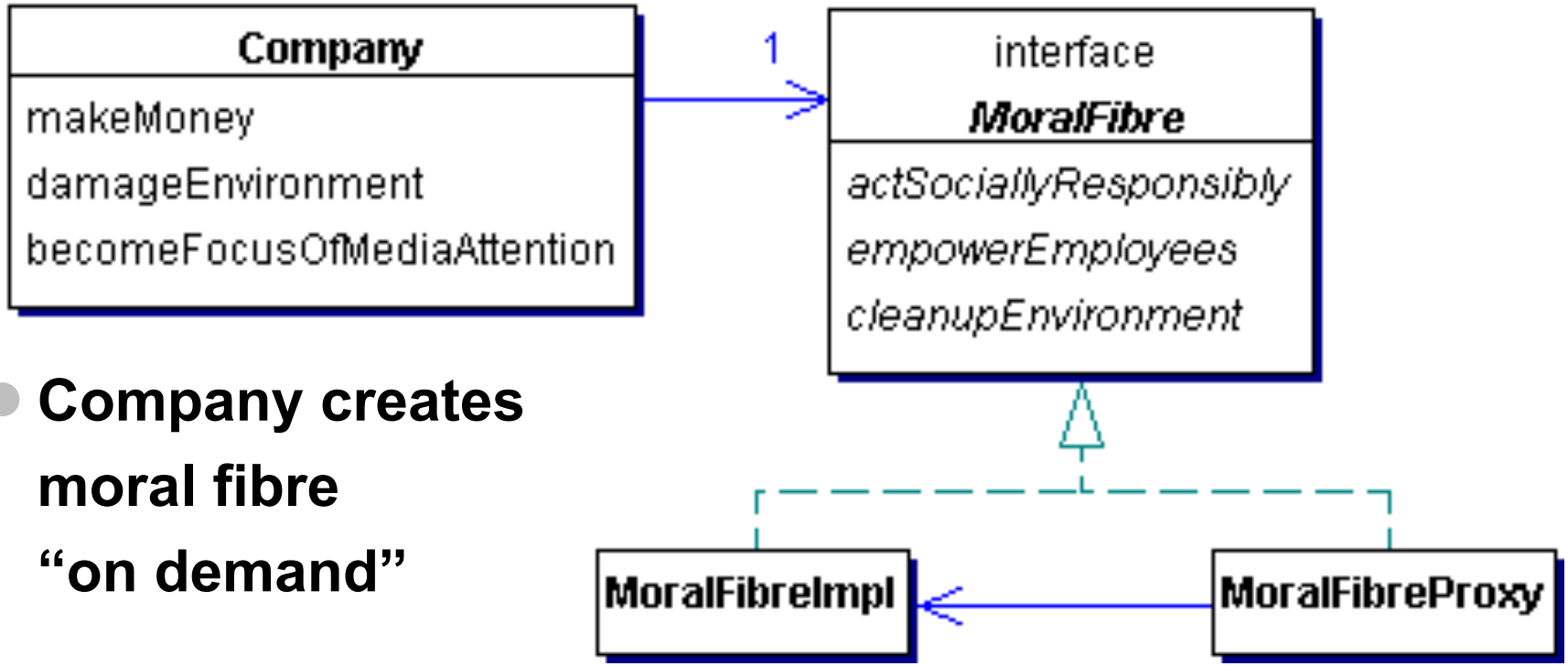
We will focus on this type

- **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 {
  // ...
  private final MoralFibre moralFibre; // set in constructor

  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 some 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(2000);
    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**

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();
}
```

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

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(); }
}

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 the generics here
      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

```
private static 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 = java.lang.reflect.Proxy.newProxyInstance(
  Thread.currentThread().getContextClassLoader(),
   new Class[]{ interface to implement },
     implementation of java.lang.reflect.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> subjectConstructor;
  public VirtualProxy(Class<? extends T> realSubjectClass,
                      Class[] constrParamTypes,
                      Object[] constrParams,
                      PointerType pointerType) {
    try {
      subjectConstructor = 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 = subjectConstructor.newInstance(
          constrParams);
        realSubjectPointer.set(realSubject);
      }
    }
    return method.invoke(realSubject, args);
  }
}
```

- **Whenever <u>any</u> 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**
  - **However, no one else will have a pointer to this object**
  - **Thus, it is fairly safe to synchronize on "this"**
- **Double-checked locking idiom was 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 very careful of using this!
public class ProxyFactory {
  public static <T> T virtualProxy(Class<T> subjectIntf) { ... }

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

  public static <T> T virtualProxy(Class<T> subjectIntf,
    Class<? extends T> realSubjectClass, PointerType type) { ... }

  public static <T> T virtualProxy(Class<T> subjectIntf,
    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> subjectInterface,
      Class<? extends T> realSubjectClass,
      Class[] constrParamTypes,
      Object[] constrParams, PointerType type) {
    return (T) Proxy.newProxyInstance(
        Thread.currentThread().getContextClassLoader(),
        new Class[]{subjectInterface},
        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**

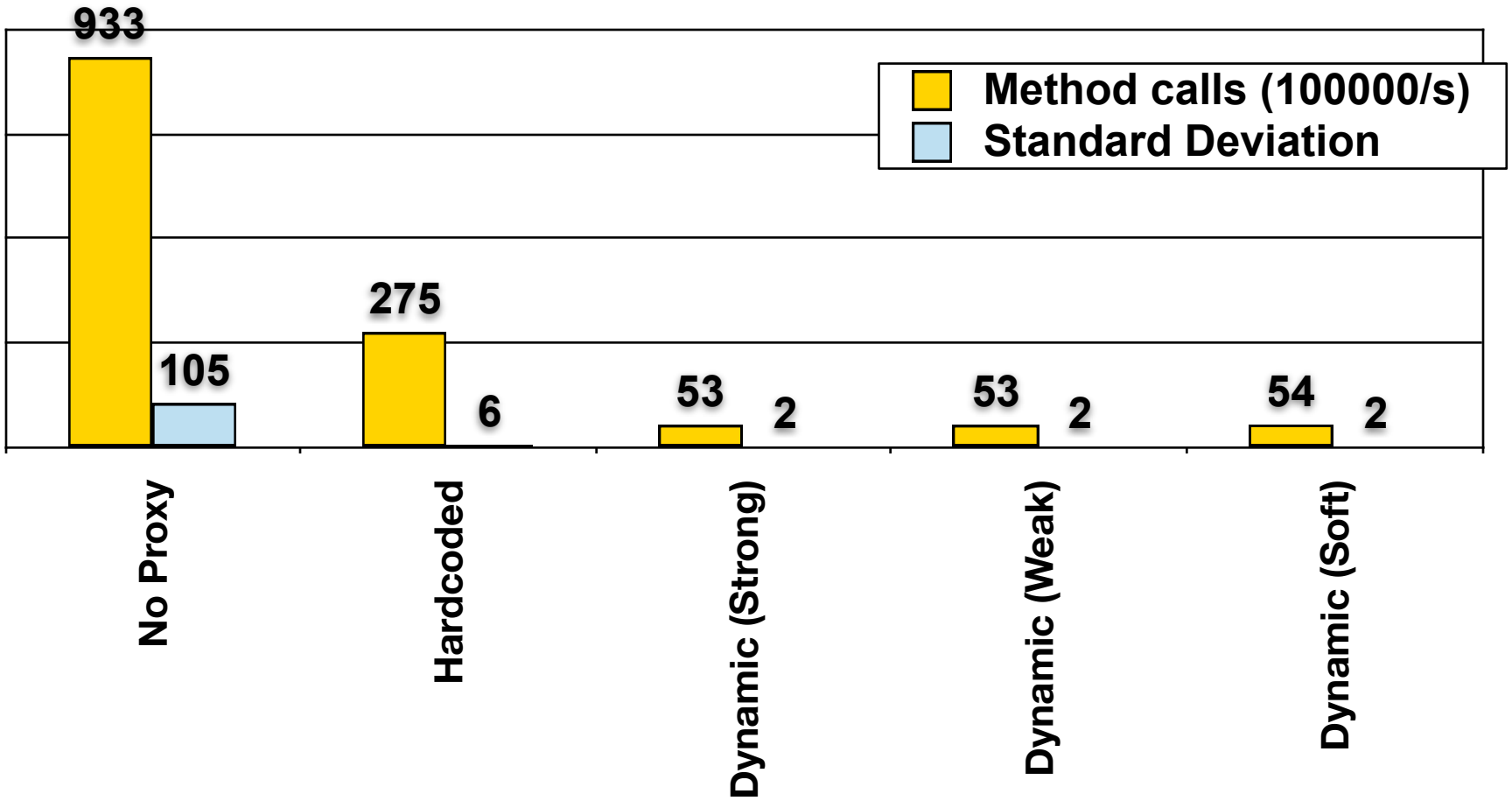
```
Company maxsol = new Company("Maximum Solutions", 1000000,
    virtualProxy(MoralFibre.class, SOFT));
 SECONDS.sleep(2);
 maxsol.damageEnvironment();
 maxsol.becomeFocusOfMediaAttention();

 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];}
```

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

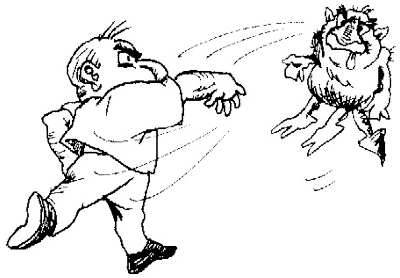


Analysis of Performance Results

- **Always look at performance in real-life context**
  - **In your system, how often does a method 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

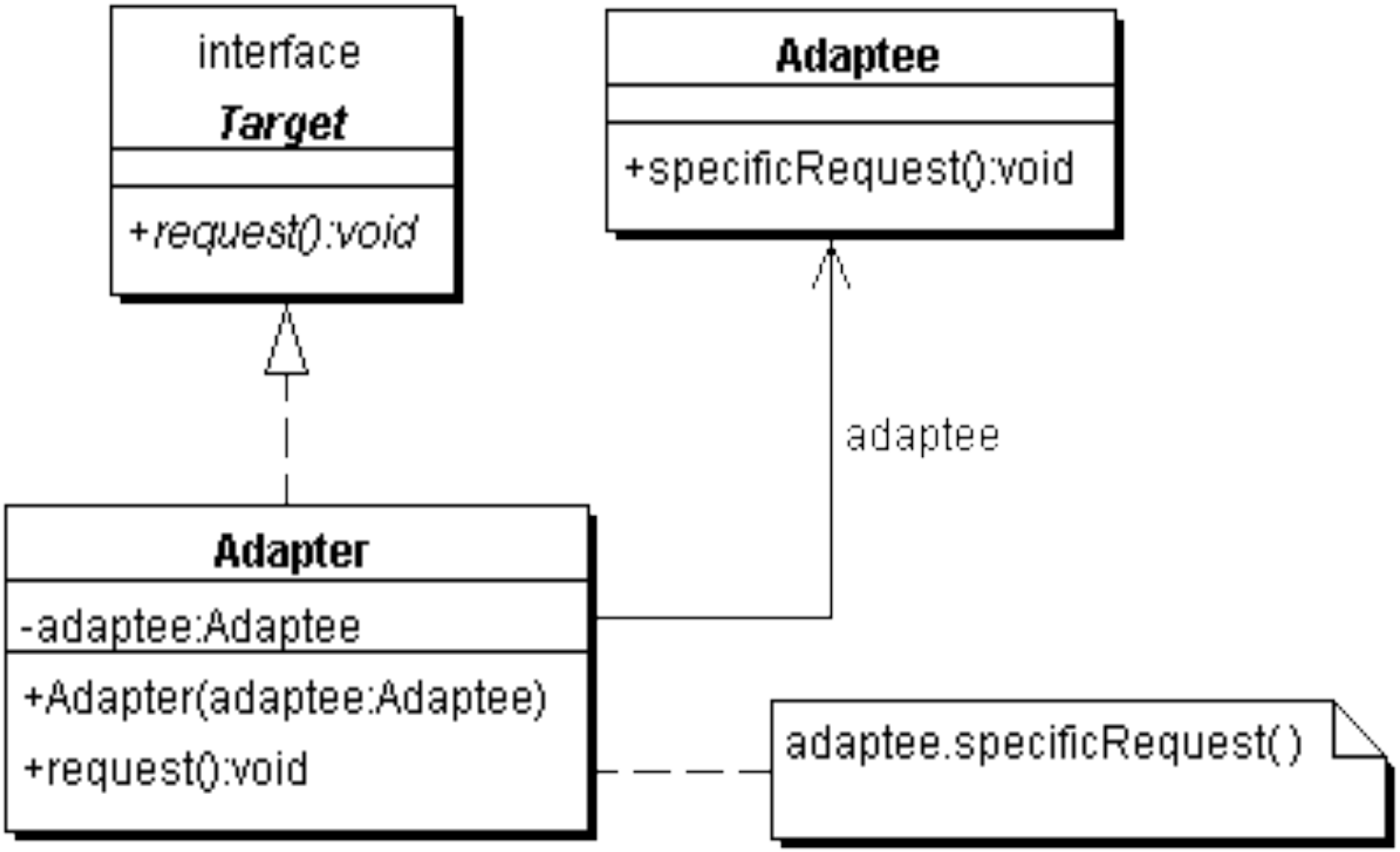
Further uses of Dynamic Proxy

- **Protection Proxy**
  - **Only route the call when caller has the correct security context**
    - **Similar to the "Personal Assistant" pattern**
- **Dynamic Decorator or Filter**
  - **We can add functions dynamically to an object**
  - **See http://javaspecialists.co.za/archive/newsletter.do?issue=034**
  - **Disclaimer: I tried to read it today, and don't understand it either**

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( // only declared methods
        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**
  - **http://javaspecialists.co.za/archive/newsletter.do?issue=108**

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**