Java Specialists in Action

Using dynamic proxies to write less code

Introduction

• Heinz Kabutz
• Java Programmer since 1997
• Publisher of The Java™ Specialists’ Newsletter
  • http://www.javaspecialists.co.za
• Read in 110 countries by about 20000 Java developers
  • Not for Java beginners 😄

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

- **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

We will focus on this type
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

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

Dynamic Proxies

● Allows you to write a method call handler
  ● Is invoked every time any method is called on interface
  ● Previous approach broken – what if toString() is called?

● 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

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

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

```java
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(); }
}
```

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

```java
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
```java
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();
}
```

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:

```java
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 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
  - However, no one else will have a pointer to this object
  - Thus, it is fairly safe to synchronize on “this”
- Allegedly double-checked locking idiom was broken pre-Java 5
  - I have personally not seen evidence to support this
Proxy Factory

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

```java
@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) { ... }
}
```

We will just show the main ProxyFactory method:

```java
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));
    }
}
```

```java
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);
    }
}
```
Weak Pointer is cleared when we don’t have a strong ref

```java
Company maxsol = new Company("Maximum Solutions", 1000000,
    virtualProxy(MoralFibre.class, WEAK));
SECONDS.sleep(2);
maxsol.damageEnvironment();
maxsol.becomeFocusOfMediaAttention();
// short term memory...
System.gc();
SECONDS.sleep(2);
maxsol.damageEnvironment();
maxsol.becomeFocusOfMediaAttention();
```

Soft Pointer more appropriate

```java
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
maxsol.damageEnvironment();
maxsol.becomeFocusOfMediaAttention();
```

Performance of Dynamic Proxies

![Performance of Dynamic Proxies Graph](image)
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
  - Similar to the "Personal Assistant" pattern
  - See [http://www.javaspecialists.co.za/archive/Issue034.html](http://www.javaspecialists.co.za/archive/Issue034.html)
  - 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:

```java
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:

```java
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:

```java
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://www.javaspecialists.co.za/archive/Issue108.html](http://www.javaspecialists.co.za/archive/Issue108.html)
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
Question