Dr. Winfried Grünewald
service@grnwld.de

γῆν ὁρῶ

Properties und Proxies

1 Einleitung

Die Java Entwicklungsumgebung bietet mit den Properties-Dateien [JPD] einen sehr einfachen Mechanismus, um eine Konfiguration zu definieren. Dabei werden im wesentlichen Schlüssel-Wert-Paare (Key/Value) in der Textdatei abgelegt, die durch Aufrufe von Methoden der Klasse `java.util.Properties` geladen werden können. Die Datentyp der Schlüssel wie auch der Werte ist `String`.

Verwendet man diesen Mechanismus, so werden die Schlüssel naturgemäß über viele Klassen verteilt. Dies erschwert die Überprüfung, ob ein Schlüssel noch benutzt wird oder ein nötiger Eintrag in der Konfigurationsdatei fehlt. Abhilfe geben am ehesten Listen aller Schlüssel z.B. als `String`-Konstanten mit der Abmachung, daß andere Schlüssel nicht verwendet werden dürfen.

```
public class Configuration{
   public final static String PROPERTY_EINS="property.eins";
   public final static String PROPERTY_ZWEI="property.zwei";
   ...
}
```

Sieht man von Namenskonventionen ab, so wird sich der Name der Konstanten und der Wert der Konstanten nur in seltenen Fällen unterscheiden. Dies läßt sich durch Aufzählungstypen [Enum] umgehen. Man kann ein Objekt eines Aufzählungstyps nach seinem Namen fragen, ihn gemäß Konvention in den eigentlichen Schlüssel konvertieren. Ganz von selbst wird die getroffene Abmachung kodiert. Man kann nur noch Elemente des Aufzählungstyps verwenden. Dadurch nimmt man aber in Kauf, auf genau eine solche Liste beschränkt zu sein und unerwünschte Projektabhängigkeiten sich nicht vermeiden lassen.

```
public enum Configuration{
   PROPERTY_EINS,
   PROPERTY_ZWEI,
   ...
}
```

Ein anderer noch gravierender Nachteil liegt in der Tatsache, daß die Werte immer vom Typ String sind. Dadurch muß der String-Wert bei der Verwendung konvertiert werden. Das Programm sollte aber an diesen Code-Stellen nicht mit den Konvertierungsdetails behelligt werden. Bei schlampiger Buchführung werden die Werte an unterschiedlichen Stellen verschieden implementiert, was meist zu ungewollten Ergebnissen führt. Damit muß diese Information ebenfalls in der Schlüsselliste kodiert werden. Dies läßt sich z.B. mit generischer Typisierung der Schlüssel erreichen [Generics].

```
public interface Key<T>{
   T convert( String value );
}
...
T getProperty( Key<T> key );
```

Durch die Angabe des Schüsseltyps kann typsicher auf die Werte zugegriffen werden. Um die Konvertierung kümmert sich entweder das Schlüsselobjekt selbst oder die umgebende Bibliothek (Framework). Theoretisch kann jeder Schlüssel einen anderen Algorithmus verwenden, den String-Wert in das passende Wert-Objekt zu wandeln. Es lassen sich auch leicht mehrere solche Listen anlegen und man muß diese Objekte benutzen.

```
public class Configuration{
   public final static Key<Integer> PROPERTY_EINS=
      new DefaultKey("property.eins", Converter.TO_INTEGER);
   public final static Key<Boolean> PROPERTY_ZWEI=
      new DefaultKey("property.zwei", Converter.TO_BOOLEAN);
   ...
}
```

Wieder stört die doppelte Angabe der Namen. Auch reicht pro Typ meist ein Algorithmus für die Konvertierung und man möchte nicht gezwungen sein, diesen für jeden Schlüssel explizit anzugeben. Die Konstanten lassen sich noch über das `java.lang.reflection` Framework mit den echten Property-Schlüssel versorgen, die Generic-Attribute sind aber zur Laufzeit nicht mehr vorhanden, und können nicht mehr genutzt werden, um den richtigen Konverter anzuziehen. Damit bleibt dieser Ansatz unbefriedigend.

2 Ansatz

Versuchen wir es anders. Der Java-Reflektion-Mechanismus erlaubt es, zur Laufzeit den Namen und den Rückgabetyp einer Methode zu bestimmen. Aus dem Namen erzeugen wir den Property-Schlüssel und über den Type des Rückgabewertes bestimmen wir den Konverter. Damit ergibt sich folgende Definition:

```
public interface Configuration{
   int propertyEins();
   boolean propertyZwei();
   ...
}
```

Diese Schnittstelle erfüllt also wieder unsere Forderung. Alle Schlüssel müssen aufgelistet sein. Mehrere solche Schnittstellen sind möglich und die ihre Verwendung ist typsicher. Auf den ersten Blick scheint der Wermutstropfen die Implementierung zu sein. Aber mit einem dynamischen Proxy [Proxy] läßt sich eine solche Implementierung sehr einfach und elegant generisch lösen.

```
public class ProxyConfigurationImplementation{
   private static class PropertyHandler implements InvocationHandler {
      public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
         String key = getKey( method );
         String value = getValue( key );
         return convert( method.getReturnType(), value );
      }
      ...
   }
   ...
   public static <Type> Type load( URL url, Class<Type> type ){
   ...
      return type.cast(Proxy.newProxyInstance( type.getClassLoader(),
         new Class<?>[]{type}, new PropertyHandler(...)));
   }
}
```

Wir erzeugen eine Instanz, die die Schnittstelle `Configuration` erfüllt, indem wir die Methode load rufen. Wird nun eine Methode unserer Schnittstelle aufgerufen, so delegiert sie die Abarbeitung an die invoke-Methode des internen InvocationHandlers. Diesem wird automatisch über das Reflection-Object method, der Methodenname übergeben. Jetzt läßt sich leicht der eigentliche Property-Name bestimmen und der Wert aus der Konfigurationsdatei laden. Da wir den Rückgabetyp der Methode ebenfalls kennen, können wir den String-Wert konvertieren und an den Aufrufer zurückgeben.

3 Default-Werte

In den Konfigurationsdateien werden üblicherweise nicht alle Werte explizit gesetzt. Nicht angegebene Konfigurationsvariablen werden mit Default-Werten belegt. Im Ansatz mit dem Strings aus der Einleitung kann eigentlichen nur an der Verwendungstelle im Code dieser Defaultwert mit der Konstanten in Zusammenhang gebracht werden. Dabei helfen Konventionen auch recht wenig. In der Variante mit dem Aufzählungstypen fällt es uns sehr leicht, die Konstruktor für den Aufzählungstypen zu erweitern und so den Default-Wert in unserer Aufzählung direkt anzugeben. Auch im Ansatz mit den Generics ist dies gut möglich.

Der Ansatz mit den Interface-Methoden läßt keine so einfache Möglichkeit zu. Aber mit Annotationen lassen sich die Methoden auch um diese Information erweitern.

```
...
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Property {
   String value();
}
```

Wieder im InvocationHandler kann der Default-Wert der Annotation entnommen werden. Falls also kein Wert aus der Konfigurationdatei geladen werden kann, wird die Annotation der Methode nach dem Default-Wert gefragt. Ist keine solche Annotation vorhanden, so wird der Default-Wert des Typs genommen.

```
...
   public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable {
      ...
      Property property = method.getAnnotation( Property.class );
      value = (property!=null) ? property.value():
         getDefault( method.getReturnType() );
      ...
...
```

4 Andere Erweiterungen

Analog zu den Default-Wert-Annotationen lassen sich auch Abweichungen von der Regel zur Generierung der Property-Schlüssel aus den Methodennamen angeben. Man erweitert die Annotation um eine weitere Eigenschaft. Der angegebene String wird als eigentlicher Schlüssel verwendet. Um diesen String nicht immer angeben zu müssen wird ein Default-Wert für die Annotationseigenschaft gesetzt, die als Schlüssel nicht verwendet werden kann.

```
...
public @interface Property {
   String value() default "";
   String key() default "";
   ...
}
```

Unterschiedliche Konverter könnte man in einem Enum definieren. Dann lassen sie sich ebenfalls über eine Annotation bestimmen. Wieder ist ein Default-Konverter nötig, um nicht alle Konverter explizit setzen zu müssen.

5 Anhang

```
package de.grnwld.examples;

import java.io.IOException;
import java.io.InputStream;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.net.URL;
import java.util.Properties;

public class ProxyConfigurationImplementation {

   private static class PropertyHandler implements
      InvocationHandler {

      private Properties properties;

      public PropertyHandler(Properties properties) {
         this.properties = properties;
      }

      public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
         Property property = method.getAnnotation( Property.class );
         String key = getKey( property, method );
         String value = getValue( property, key );
         return convert( method.getReturnType(), value );
      }

      private String getValue(Property property, String key) {
         String value = properties.getProperty( key );
         if( value!=null){
            return value;
         }
         return (property!=null) ? property.value(): null;
      }
```

```
      private String getKey( Property property, Method method ){
         if( property!=null ){
            String key = property.key();
            if( !"".equals(key) ){
               return key;
            }
         }
        return method.getName();
      }

      private Object convert(Class<?> returnType, String value) {
        if( value==null ){
            return null;
        } else if( returnType.equals(Boolean.TYPE) ){
            return "true".equalsIgnoreCase(value);
        } else if( returnType.equals(Integer.TYPE) ){
            return Integer.decode(value);
        } else if( returnType.equals(String.class) ){
            return value;
        }
        throw new UnsupportedOperationException(
            "Cannot convert String to "+returnType.getName());
      }
   }

   public static <Type> Type load( URL url, Class<Type> type )
      throws IOException {
      if( url==null || type==null ){
         throw new IllegalArgumentException();
      }
      Properties properties = new Properties();
      InputStream inputStream = url.openStream();
      try{
         properties.load( inputStream);
         return type.cast(Proxy.newProxyInstance( type.getClassLoader(),
            new Class<?>[]{type},
            new PropertyHandler(properties)));
      } finally {
         inputStream.close();
      }
   }
}
```

Literaturverzeichnis

[JPD] http://de.wikipedia.org/wiki/Java-Properties-Datei

[Proxy] http://download.oracle.com/javase/1.5.0/docs/guide/reflection/proxy.html

[Generics] http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

[Enum] http://download.oracle.com/javase/tutorial/java/javaOO/enum.html