FANCY ONE MORE JAVA CUP?

FRIDAY, APRIL 24, 2009

Bean properties without hard-coded names?

The problem

Java Beans specifications have been around for more than a decade and, although they were good for tools (IDE), their full capabilities (bound properties in particular) were not much used in 3nd party libraries until only recently (e.g. JSR-295 "Beans binding", JSR-303 "Beans Validation", Glazed Lists...)

I have never been a big fan of Bean properties for many reasons:

1. they incur a lot of boilerplate code (in particular for bound properties)
2. there's no way to refer to a bean property by anything else than its name, a hard-coded String, hence this doesn't bear refactoring
3. they offer no compile-time safety (for the same reason as above, e.g. how can you be sure that "SSID" property is a String, it might be an int, or even an instance of a custom class!)

OK, I know some points are easily worked around:

- Issue 1, for instance, is not a problem for most IDE: they will generate getter/setter for all your properties in no time; however, you have to remember that software maintenance cost increases with the number of lines of code to maintain (whether manually produced or automatically generated).
- Refactoring of properties (issue 2) can be correctly handled by your IDE as well (with just some little extra effort)

Nevertheless, I see no workaround to issue 3, no IDE -as far as I know- will check that a hard-coded name refers to a property of some given type!

For developers working on Swing applications in particular (but developers in other architectures may be concerned as well), beans usage is a must, so they face those issues everyday.

The solution

ABOUT ME

JEAN-FRANCOIS POILPRET
VIETNAM

VIEW MY COMPLETE PROFILE

SUBSCRIBE

Posts

Comments

BLOG ARCHIVE

▼ 2009 (9)
May (1)
April (1)
February (3)
January (4)

► 2008 (18)

MY LINKS

DesignGridLayout
GUTS: Guice-GUI Framework
HiveBoard

MY PREFERRED BLOGS

Pushing Pixels
Design, uninterrupted #73
2 days ago

I will further demonstrate a proof of concept of how we can solve issues 2 & 3 above. The complete prototype also includes a solution to issue 1 (i.e. transforming "normal" bean properties into bound properties) but I won't discuss it here because many solutions have been blogged about for a couple of years. There's even an example of that in cglib snippets.

Before getting to the solution, I'd like to describe how I came to it. It's quite simple. For unit testing, I like to use EasyMock. In EasyMock, here is how you create a mock and define its expectations:

```
import static org.easymock.EasyMock.expect;
import static org.easymock.EasyMock.createMock;
...
CustomerService mock = createMock(CustomerService.class);
expect(mock.getNextAppointment()).andReturn(new Date());
...
```

What happens here is that EasyMock generates a mock implementation of CustomerService interface so that any call to any method is recorded. Then the generic method expect(T) allows EasyMock to have compile-time safety in the andReturn(T) call.

Hence I had the thought of using the same kind of API to get access to bean property:

```
import static net.sf.beanutils.BeanUtils.mock;
import static net.sf.beanutils.BeanUtils.property;
import net.sf.beanutils.Property;
...
MyBean mock = mock(MyBean.class);
Property<MyBean, String> property = property(mock.getCustomerId());
System.out.println(property.name());
```

Here, I would introduce a specific generic class Property<T, U> that encapsulates description of a given property of a specific bean class.

Of course, in the example above, you've probably seen that -technically- this is different from EasyMock because EasyMock deals with *interfaces* while we have to deal with beans (*non abstract classes*). But there is an EasyMock extension that supports just that, it is based on cglib.

I have decided to also use cglib for my proof of concept, because its API seems quite easy and I found the provided examples quite straightforward.

First of all, let me introduce Bean<T> class, which is the main factory for Property instances of a given bean:

```
public class Bean<T>
{
    // The only way to get a Bean<T> instance is to use this factory
 method
    public static <T> Bean<T> create(Class<T> clazz) {...}

    // Initializes all members (uses cglib)
```

Jonathan Giles
JavaFX Your Way: Building JavaFX Applications with Alternative Languages
6 days ago

Alex Ruiz's Blog
JUnit: Custom ExpectedException rules...rule!
1 week ago

Andres Almiray's Weblog
Griffon's second Birthday
2 weeks ago

LABELS

Java (25)
DesignGridLayout (18)
Swing (14)
Layout (10)
GUI (7)
API (5)
open source (3)
Guice (2)
Plugin (2)
rant (2)
JSR-296 (1)
JavaFX (1)
Jazoon (1)
Maven (1)
Talk (1)
bean (1)
hooks (1)
new year resolutions (1)
opinion (1)
properties (1)

```
    protected Bean(Class<T> clazz)
    {
        _clazz = clazz;
        _properties = ReflectUtils.getBeanProperties(_clazz);
        _mockInterceptor = new MockInterceptor(_properties);

        // Create a mock immediately with cglib
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(clazz);
        enhancer.setCallback(_mockInterceptor);
        _mock = clazz.cast(enhancer.create());
    }

    public Class<T> type() {...}

    // Returns a T mock, used in conjunction with property() method
(as argument)
    public T mock() {...}

    // Returns a beans property reference without using its hard-cod
ed string name
    // This pattern will always survive bean refactoring (compile-sa
fe)!
    // The returned reference can be used to get/set property value
or get its
    // name (in a safe way)
    public<U> Property<T, U> property(U mockCall)
    {
        PropertyDescriptor property = _mockInterceptor.lastUsedPrope
rty();
        checkType(mockCall, property);
        return Property.create(property);
    }

    // Create a new bean that delegates to this one but makes all it
s
    // properties bound
    public T proxy(T source) {...}

    // Global cache of Bean objects (each class T should have only o
ne
    // Bean<T> instance)
    private static final Map<Class<?>, Bean<?>> _cache =
        new HashMap<Class<?>, Bean<?>>();

    private final Class<T> _clazz;
    private final PropertyDescriptor[] _properties;
    private final MockInterceptor _mockInterceptor;
    private final T _mock;
}
```

Only the relevant methods & API are shown above. The "meat" of the code is essentially in `Bean<T>` constructor and the `property()` method. The `proxy()` method is not shown here but is also interesting.

The second interesting piece of code is the `MockInterceptor` class, which is automatically called by cglib for any call to a method of `_mock`:

```
class MockInterceptor implements MethodInterceptor
{
    public MockInterceptor(PropertyDescriptor[] properties)
    {
        _properties = properties;
    }

    public PropertyDescriptor lastUsedProperty()
    {
        PropertyDescriptor property = _lastUsedProperty;
        _lastUsedProperty = null;
        return property;
    }
```

```
    public Object intercept(
        Object target, Method method, Object[] args, MethodProxy pro
xy)
        throws Throwable
    {
        _lastUsedProperty = null;
        // Check this is a getter
        for (PropertyDescriptor descriptor: _properties)
        {
            if (method.equals(descriptor.getReadMethod()))
            {
                _lastUsedProperty = descriptor;
                break;
            }
        }
        //TODO try to return something non-null when possible
        //TODO should we call super method if not abstract of course
)?
        return null;
    }

    private final PropertyDescriptor[] _properties;
    //FIXME should be in a ThreadLocal no?
    private PropertyDescriptor _lastUsedProperty = null;
}
```

The principles are quite simple actually: every time a *getter* of _mock is called, the matching java.beans.PropertyDescriptor is stored in _lastUsedProperty. The latest is used in Bean<T>.property() method to create a new Property<T, U> instance:

```
public class Property<T, U>
{
    static<T, U> Property<T, U> create(PropertyDescriptor descriptor
)
    {
        return new Property<T, U>(descriptor);
    }

    protected Property(PropertyDescriptor descriptor)
    {
        _descriptor = descriptor;
    }

    public Class<?> type() {...}
    public U get(T bean) {...}
    public void set(T bean, U value) {...}

    public String name()
    {
        return _descriptor.getName();
    }

    private final PropertyDescriptor _descriptor;
}
```

As you can see, Property<T, U> is merely a wrapper to a java.beans.PropertyDescriptor instance, with additional type information (T: type of the bean, U: type of the bean property) added as generic parameters, making it typesafe.

With this little design, here is what we have achieved:

```
import static net.sf.beanutils.Bean.*;
...
Bean<MyBean> helper = create(MyBean.class);
```

```
MyBean mock = mock();
Property<MyBean, String> prop1 = helper.property(mock.getMyFirstProp
erty());
Property<MyBean, Integer> prop2 = helper.property(mock.getMySecondPr
operty());
...
System.out.println(prop1.name()); // prints "myFirstProperty"
MyBean bean = new MyBean();
prop1.set(bean, "Something");
prop2.set(bean, 123);
prop1.set(bean, 123); // Compile-time error!
```

This is not *exactly* like the foreseen use shown at the beginning of this post, but that's not very far!

Evaluation & Limitations

All initial issues have been solved:

1. no boilerplate code for bound properties (performed by `Bean<T>.proxy(T source)` method)
2. no need for hard-coded property names thanks to `Property<T, U>.name()` that will return the right name
3. thanks to heavy use of Java 5 generics, this proof of concept provides compile-time safety

There are still some limitations with the current design:

1. this works only with classes that comply to java-beans specifications (but that was a pre-requisite of the proof of concept)
2. this won't work with final classes or final getters of bean classes (limitation of cglib)
3. the current prototype is not thread-safe (that one is very easy to fix, with a `ThreadLocal`)
4. write-only properties (no getter) are not supported (that could be added if needed)
5. type-safety may be not guaranteed in case of misuse of the API (e.g. not following the usage example shown above)

Next?

The current proof of concept just demonstrated what was feasible. Some possible next steps would be:

1. fix limitations #3 & #4 above (quite easy)
2. add possibility to add/remove property listeners directly in `Property<T, U>` (would ease creation of new frameworks such as binding or validation)
3. provide a work-around to limitation #2 (like logging a warning that

some methods are final)
4. check that generated proxy can work with Hibernate and Beans-bindings
5. create an Open Source project somewhere (I thought about SourceForge which I am familiar with)

For step #5, I am not sure I am willing to start such a project alone (I already have a bunch of other OSS projects currently and I never can find enough time for them), so if someone is interested, send me a note!

You can find the complete project (maven 2 and eclipse) here.

Any comments are appreciated. Have fun with this prototype!

POSTED BY JEAN-FRANCOIS POILPRET AT 6:27 AM 
LABELS: BEAN, JAVA, PROPERTIES

---

8 COMMENTS:

willcodejavaforfood said...

Nice work,

Something ocurred to me and please correct me if I am wrong. When using bound properties the traditional way there sure is a lot of boiler plate code setting everything up but once done you dont need to do much when reading/writing properties. It seems like your proof of concept has somewhat reversed this. Very little setup but the boiler plate code comes later when you need to access the properties. IE all the creating of Property objects.

APRIL 27, 2009 11:17 PM

Anonymous said...

Are you aware of the Bean properties project?

https://bean-properties.dev.java.net/

APRIL 28, 2009 3:29 AM

Jean-Francois Poilpret said...

@willcodejavaforfood:
Thanks for your encouragements. But I think you didn't follow my post fully (I should probably have written it differently to make it clearer;-)).

This prototype solves 3 issues.

The bound properties boilerplate code is just a "small" one (because many people have solutions already). And the prototype just needs to use one line of code per bean instance for solving that specific issue.

This prototype solves 2 other issues NOT solved by any boilerplate code, and this is the main point of my post!

@Anonymous:
Yes I'm aware of bean-properties (I have never used it but I like its principles a lot).

As I see it, the main problem with it is that it is new way to write beans, not compliant with Java Beans specifications. I think that makes it difficult to integrate with the zillion frameworks/libraries out there, all based on "pure" Java Beans.

In addition, if you make beans-properties more java beans compliant (the project offers some tools for that), you still don't solve the initial issue of hard-coding your property names for use by those 3rd-party libs, which was the main point of this post;-)

Cheers

APRIL 28, 2009 6:05 AM

Jean-Francois Poilpret said...

@willcodejavaforfood:
to clarify further, if you don't care about type safety, you can adapt the source code to remove Property class, and just keep Bean.

You can even add a BeanUtils static utility class with just the following methods (sorry for using [], chevrons are not allowed in comments):

static [T] T mock(Class[T] beanClass);
static[U] String name(U mockCall);
static [T] T proxy(T bean);

2 first methods are used to get the name of the property (you need to statically import them):
String name = name(mock(Bean1.class).getProp());

The last method is used to create a proxy that converts properties to bound properties (and add the methods for adding/removing

listeners).

Maybe I'll add such a BeanUtils class to the proto.

The added value of Property class is that it can help further creation of libraries for binding, for example.

Hope this helped clarifying my previous point.

APRIL 28, 2009 7:16 AM

Anonymous said...

Maybe I'm missing something, but what's the benefit of having to write

```
import static net.sf.beanutils.Bean.*;
...
Bean<MyBean> helper = create(MyBean.class);
MyBean mock = mock();
Property<MyBean, String> prop1 =
helper.property(mock.getMyFirstProperty());
Property<MyBean, Integer> prop2 =
helper.property(mock.getMySecondProperty());
...
System.out.println(prop1.name()); // prints "myFirstProperty"
MyBean bean = new MyBean();
prop1.set(bean, "Something");
prop2.set(bean, 123);
prop1.set(bean, 123); // Compile-time error!
```

instead of

```
MyBean bean = new MyBean();
bean.setMyFirstProperty("Something");
bean.setMySecondProperty(123);
bean.setMyFirstProperty(123); // Compile-time error!
```

APRIL 28, 2009 3:22 PM

Anonymous said...

@Jean-Francois Poilpret: You should type > to get > and < to get <

APRIL 28, 2009 4:24 PM

Jean-Francois Poilpret said...

@Anonymous(1): in the example you give, you're right, there's no point doing it my way!
The point is when you want to build or use a beans-binding library (or other that needs to access bean properties):

For building a beans-binding library, you would pass Property<T, U> along with the T instance to let the library handle the binding.

For using an existing beans-binding library, you would use Property<T, U>.name() to pass the name of your properties rather than a hard-coded string of the property name.

@Anonymous(2) [not sure it's the same person]: thanks for the tip;-)

APRIL 28, 2009 5:57 PM

willcodejavaforfood said...

@Jean-Francois Poilpret: OK now I understand the context in which to use it. I thought at first it was instead of using get/set and found it interesting but couldnt see much practical use, but using together with a beans-binding library makes a lot more sense.

Good thing Anonymous asked that question comparing the two.

APRIL 28, 2009 10:20 PM

POST A COMMENT

Comment as: [redacted]

[redacted]

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)