SpringContracts

Quick Start

Inhaltsverzeichnis

Setup

Step 1 – Download

Visit http://sourceforge.net/projects/springcontracts and download the latest release of SpringContracts (springcontracts-0.11-with-dependencies.zip)

Step 2 – Unzip

Unzip springcontracts-0.11-with-dependencies.zip to a directory of your choice

Step 3 – Licence

Read licence.txt and readme.txt (located in the root directory of the unpacked zip)

Step 4 – Configure classpath

Add dist/springcontracts.jar and the required third party jars (located in directory /lib) to the classpath

Writing your first Contract

We will start with a very simple contract specification for a well known datatype, called '***Stack***'.
Let's define an interface for ***Stack*** which can be implemented by a various number of implementations.

```
public interface Stack {

      /**
       * @return the maximum number of elements, the stack can carry
       */
      public int getCapacity();

      /**
       * @return the current number of elements on the stack
       */
      public int getSize();

      /**
       * pushes an element on top of the stack, until the stack is full
       */
      public void push( Object element );

      /**
       * shows the current element on top of the stack without removing it.
       */
      public Object getTop();

      /**
       * @return the current element on top of the stack, removing it
       * from the stack
       */
      public Object pop();
}
```

Having specified the semantic behaviour of a Stack by documenting every single method, we though can't ensure, that any implementation of Stack will follow the intended specification (it's easy to provide an implementation of interface Stack with a completely different behaviour - for example by always delivering the current element at the bottom of the Stack when calling pop(). This implementation would truly statisfy the syntactic 'contract' of the interface, but not the intended semantic contract, since Javadoc can't force and respectively check, that every implementation of Stack fulfill the specified semantic contract).
To force that all implementations adhere to the semantic specification of Stack, we now will specify the semantic contract for it, using preconditions, postcondition and invariants.

Writing an invariant

Invariants describe the conceptual model of Stack in that they express conditions about attributes that won't change over the whole lifecycle of an object.
In our case, the current size of a Stack is always between zero and the capacity of the Stack. We now have to express this invariant in a formal way, so that it's checkable against the implementations of Stack at runtime.

Expressing Conditions with Expression Language

With the Expression Language (EL), SpringContracts provides a build in support for a language to specifiy such conditions. In our case, we would write the following condition for the mentioned invariant:

```
this.size >= 0 and this.size <= this.capazity
```

We use the keyword ***this*** to reference to the instance of Stack itself (also referencable in pre- and postconditions). Using EL-Syntax, we can refer to properties (which provides at least a getter, since we only read the current value of the property) by simply typing the property name.

Assigning Invariants to Classes or Interfaces

The simplest way to assign an Invariant to a class or interface is to use Java 5 annotations (there are also other ways of assignment, not discussed in this document. Please refer to the Reference for further information).
SpringContracts provides an annotation ***@Invariant***, which allows for easy association between the invariant's condition and the interface Stack. Since invariants relates to the behaviour of the whole instance of Stack, we annotate it at class, respectively interface level:

```
@Invariant( condition="this.size >= 0 and this.size <= this.capazity" )
public interface Stack { ... }
```

Implementing the 'contracted' Interface

Now let's test the invariant by writing an implementation for Stack that purposely violates the invariant, in that it allows pushing elements onto the stack beyond its capacity (assuming that pushing a new element to the stack increases its size).

```
import java.util.ArrayList;
import java.util.List;

public class UnboundSizeStackImpl implements Stack{

	private int capacity = 0;

	private List elements = new ArrayList();

	public UnboundSizeStackImpl( int capazity ){
		this.capacity = capazity;
	}

	public int getCapacity() {
		return capacity;
	}

	public int getSize() {
		return elements.size();
	}

	public Object getTop() {
		return elements.get( elements.size() - 1 );
	}


	// continued ...
```

```
    public Object pop() {
          Object elem = getTop();
          elements.remove( elements.size() - 1 );

          return elem;
    }

    public void push(Object element) {
          elements.add( element );
    }
}
```

Configuring Spring's Application Context

In order to validate contracts at runtime, we have to advice ***`UnboundSizeStackImpl`*** with an appropriate ***`ContractValidationAspect`*** (since the validation logic is hooked in, based on AOP).
This is a relatively easy task, configuring Spring's ApplicationContext with the provided ***`ContractValidationInterceptor`*** (and additionally activating AOP-Autoproxying):

```
  <aop:aspectj-autoproxy/>

  <aop:config>
    <aop:aspect ref="contractValidationAspect">
      <aop:pointcut id="allMethods" expression="execution(* *(..))"/>
      <aop:around pointcut-ref="allMethods" method="validateMethodCall"/>
    </aop:aspect>
  </aop:config>

  <bean id="contractValidationAspect"
  class="org.springcontracts.dbc.interceptor.ContractValidationInterceptor"/>
```

This configuration will run SpringContracts with default settings, using annotations, EL as specification language and throwing ***`ContractViolationExceptions`*** if a Violation of a contract element (i.e. invariants) occurs.
(You can configure these settings amongst others. Please refer to the Reference for a more detailled overview of SpringContracts config capabilities)
As you can see, the pointcut refers to all method calls of arbitrary classes. Having other classes or interfaces with contracts beside of Stack would automatically advice and validate them, too. No further configuration of the Aspect is needed.

Because Spring has to recognize the beans in order to advice them, we also configure ***`UnboundSizeStackImpl`***:

```
  <bean id="stack" class="quickstart.stack.UnboundSizeStackImpl">
    <constructor-arg><value>3</value></constructor-arg>
  </bean>
```

That's it!
We now have a simple Example ready for rumbling in Spring ...

Testing Contract Validation

The only thing left is to test our example. Let's write a UnitTest (assumed that Spring's ApplicationContext is stored in a file named ***beanConfig.xml***, located in package ***quickstart.stack***):

```
import junit.framework.TestCase;

import org.springcontracts.dbc.validation.ContractViolationException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class UnboundSizeStackImplTest extends TestCase {

  private static final String BEAN_CONFIG_SOURCE =
                                  "/quickstart/stack/beanConfig.xml";

  ApplicationContext appContext = null;

  public void setUp() throws Exception{
    appContext = new ClassPathXmlApplicationContext( BEAN_CONFIG_SOURCE );
  }


  public void testInvariantOnStack() throws Exception{
    assertNotNull( appContext );

    Stack stack = (Stack) appContext.getBean( "stack" );
    assertNotNull( stack );

    stack.push( "element1" );
    stack.push( "element2" );
    stack.push( "element3" );

    try{
      stack.push( "element4" );
      fail();
    }
    catch( ContractViolationException cve ){
    }
  }
}
```

Note that the contract (invariant) allows to push a maximum number of elements onto the stack, restricted by its capacity (which is set to a maximum of three elements inside the Bean's config at Spring's ApplicationContext). Pushing the fourth element onto the stack violates the invariant (since invariants have to be valid at every obervable moment in the objects lifecycle, and therefore also before and after a method call).

Preconditions and Postconditions

In order to complete the contract for Stack, we should constrain the single methods.
Preconditions allow for specifying any requirements the method longs for, expressing under which circumstances it's valid to call a method. This part of the contract has normally to be fulfilled by the client.
For example, we could require, that push() is never called with null (a so called physical constraint):

```
not empty arg1
```

Using EL again, this time we refer to the first (and in this case only) argument of method push() via keyword ***arg1***.

Postconditions are used to describe the observable impacts of a method. They ensure to clients, that a method has a specific effect (for that reason the client calls the method). Postconditions has usually to be fulfilled by the service provider (that is the method itself).
For example, we could ensure, that the actual pushed element is always that one on top of the Stack:

```
arg1 == this.top
```

Since this is a quick start, we do not discuss the whole contract for Stack at this place.
Instead, here's the full contract of Stack (you maybe could found some missing constraints, depending how restrictive your contract should be):

```
  @Invariant( condition = "this.size >= 0 and this.size <= this.capazity" )
  public interface Stack { ... }

  /**
   * @return the maximum number of elements, the stack can carry
   */
  @Postcondition( condition = "return > 0" )
  public int getCapacity();

  /**
   * @return the current number of elements on the stack
   */
  public int getSize();

  /**
   * pushes an element on top of the stack, until the stack is full
   */
  @Precondition( condition = "arg1 not empty  and  this.size < this.capacity" )
  @Postcondition( condition = "arg1 == this.top  and  this.size == old:this.size + 1" )
  public void push( Object element );

  /**
   * shows the current element on top of the stack without removing it.
   */
  @Postcondition(
     condition = "( this.size > 0 ) ==> ( not empty return and this.top == old:this.top )
                   and ( this.size == 0 ) ==> ( empty return )" )
  public Object getTop();

  /**
   * @return the current element on top of the stack, removing it
   * from the stack
   */
  @Postcondition(
     condition = "(old:this.size>0) ==> (return == old:this.top and this.size == old:this.size - 1)
                  and ( this.size == 0 ) ==> ( empty return )" )
  public Object pop();
}
```

Summary

As you might have seen, SpringContracs can add value, increasing the semantic expressiveness of software.
You can annotate your classes or interfaces with invariants, preconditions and postconditions, that helps you to express the required behaviour, which will reach out beyond syntactical correctness.
This kind of specification is more than a plain documentation in that it can be validated at runtime. The violation of a contract results in an Exception. This is the default behaviour, which can be – among other parts of SpringContracts – easily configured using Spring's capabilities. This said, contracts are specs n' checks.
The single conditions itself are expressed by a formal language. At this moment, SpringContracts provides build in support for EL (further releases will provide support for other languages such as MVEL or OGNL, which can be switched without any difficulty, again using the potentials of Spring).

We have covered only the core features of SpringContracts in this quick start.
For a more detailled look of the capabilities, please refer to the SpringContracts Reference.

Enjoy!

Note

If you have some suggestions or critical comments due to this quick start or SpringContracts at all, please don't hesitate and write an email to mario.gleichmann@gmx.de

Any feedback is appreciated !