SpringContracts

Design by Contract
for Java
with seamless integration into the Spring Framework

- DRAFT -

This is a draft version !

Content

Chapter 1 – Introduction

Motivation

Writing reliable and maintainable software is one of the biggest challenges in our industry. Especially if you have to rely on sources or integrate with third party software, not written by yourself. Even if you're working in a project with more than a couple of developers, where all the written source is accessible - at a certain point you have to 'trust' the code written by others in that you have to understand the conceptional model and work on them. This conceptional model is usually a lot more abstract than the code level is – and that for good reasons. It allows for thinking, communicating and modeling at a higher level, where a lot of implementation details are hidden. And important for development itself - it allows for an appropriate usage of a software component.
But this requires a sound and clear representation of that conceptional model that is shared between all participants of the development process.

What's in a name?

So what we need to (re-)use a software component in a proper way is a model of its effectiveness. We have to know about the conditions under which is it valid to use a component (i.e. are null values allowed as key for a java.util.Map?), about the effects of its methods (is poll() retrieving and also removing the first element of a java.util.Queue?) and about it's context it will run properly (can the size of a java.util.Stack ever be greater than its current capacity?).

A half typed language like Java can help to express such a model. For example, having a method findWithdraws() that allows only to pass a java.util.Date as an argument is a helpful hint about the correct usage of that method. You can't pass a BigDecimal to that method without having the compiler to beef about.
So part of that conceptional model we can trust is build upon types. Unfortenately it is not build upon class names or method names since they are only hollow words (and of the same reason the model is especially not based on any kind of documentation). It's easy – and you surely saw this more than one time – to give a method a name and let the method do a completely different job, at least a slightly different job you would infer from it's name (i.e. having getters that alter the state of an object).

Hence, names can't help in building reliable conceptional models. And even types can't do to a full extent. Back to the above mentioned method, a type can't say if it's allowed to pass future Dates or maybe only Dates within a specific time period. Moreover, a type can't give you information about the full effects of a method. So if you for example call a withdraw() method on a class teller, wether the argument nor the return type will give you information if the transaction will additional charge your balance with an administration fee, should you have used a foreign teller. You also don't know how the method will behave, if you call the method with an amount (as the argument) which is higher than the actual balance.

Trust me

Names and types can only give a hint, but can never give a trustfully and therefore no satisfying answer about the effectiveness of a software component. Let's revisit java.util.Queue. Of course we could rely on the method names. Of course we could rely on Javadoc. But its very simple to implement the 'contract' of this interface without following the stated documentation. You could implement poll() without removing the head of the queue and – the other way round – let have peek() do the job and release the first element. What we need to trust in the documented or named effects, is a way to validate that the implementation not only accomplishes the syntactical contract of an interface, but also complies the intended semantic behaviour.

Design by Contract

Contracts are a widely accepted way of expressing the terms and conditions under which a relationship between a supplier and a client who wants to use the suppliers services will accomplish.
Generally, the contracts conditions defines mutual obligations and benefits both sides agree on.

> *The supplier has to provide a certain product (obligation) and is entitled to expect that the client has paid its fee (benefit). The client must pay the fee (obligation) and is entitled to get the product (benefit).*
> *Both parties must satisfy certain obligations, such as laws and regulations, applying to all contracts.*
>
> http://en.wikipedia.org/wiki/Design_by_contract

Bertrand Meyer was the first one who adopts the idea of contracts to the area of software development in order to increase the semantic expressiveness of software components by specifying contracts on them and validate the associated conditions.

> *Similarly, if a routine from a class in object-oriented programming provides a certain functionality, it may:*
>
> *Impose a certain obligation to be guaranteed on entry by any client module that calls it: the routine's **precondition** — an obligation for the client, and a benefit for the supplier (the routine itself), as it frees it from having to handle cases outside of the precondition.*
>
> *Guarantee a certain property on exit: the routine's **postcondition** — an obligation for the supplier, and obviously a benefit (the main benefit of calling the routine) for the client.*
>
> *Maintain a certain property, assumed on entry and guaranteed on exit: the **class invariant**.*
>
> http://en.wikipedia.org/wiki/Design_by_contract

There are three contract elements which can be used to express the contract that goes with a software component (as a supplier of services) and any other classes who wants to use them (as clients of that software component).

Example – A contract for Stack

We will start with some simple contract specifications 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();
}
```

Preconditions

Preconditions refers to method calls. They specify the circumstances under which it is valid to call a suppliers method. The supplier can benefit from the fact, that the client have to keep the conditions. The client is obligated to call the supplier only if the required conditions are met.
If the client doesn't follow the claimed precondition, the supplier isn't forced to fulfil the rest of the methods contract (that is the benefit for the client, the supplier has to ensure under normal conditions).

Now let's consider the precoditions for the Stack's method **`push()`**.
There is at least one 'aggresive' Precondition we could state: It is not allowed to push another element onto the Stack, if the stack's maximum capacity is allready reached. Using the Stack's properties **`capacity`** and **`size`**, it's easy to specify such a requirement as a precondition:

'the current size of the stack is less than the stack's capacity' : **`size < capacity`**

As we have seen, it's allowed to refer to any of the class' properties (more generally said, the class' so called query methods) inside a condition. We will provide an detailed overview of all referable elements in the chapter about condions and the associated specification languages.

Postconditions

Postconditions refers to method calls, too. They specify the effects of the methods task activities, after the method has finished its job.
These effects are the clients benefit - the reason why the client called the method at all. From the supplier's point of view, the effects are the obligations the supplier has to ensure.

What are the postconditions for the **`push()`** method?
First of all, we could state, that the size of the Stack has increased by one. Secondly – and that's the essential part of the semantic behaviour of a Stack – the **`push()`** method has to ensure, that the actual pushed element is on top of the Stack.

'the size is increased by one – the newly added element' : **`size == old:size + 1`**

'the newly added element is on top of the Stack' : **`arg1 == top`**

We used again some of the properties of the Stack. Note, that we have to refer to the value of the property **`size`** before (the 'old' value) and after the methods execution (the new, current value), in order to express that the value has changed by the methods execution. A new element is introduced with arg1, which stands for the (first) element of the method.

Invariants

Invariants express the conception of a class, which has to be valid over its whole lifecycle. They state conditions that have to be always true for all objects of that class.

We could state that the **`size`** of a Stack is always equals or greater than zero, but this is trivial in that case.
A more valuable claim could be, that the **`size`** is always less or equals than the **`capacity`** of the Stack.

'the size of the stack is always less or equals than the stack's capacity' : **`size <= capacity`**

Note, that this invariant looks very similar to the precondition. But there's an important difference. Where the precondition states, that it's valid to push another element onto the Stack, if there's at least one free place (less), the invariant claims that the size can reach the maximum capacity (less or equals).

Overview

SpringContracts is a Java solution for 'Design by Contract' in that it allows to specify and validate contracts using invariants, pre- and postconditions and apply them to any arbitrary class or interface.

SpringContracts fits seamlessly into the Spring Framework – a very popular and widely used Enterprise Java Framework. With the potentials of Spring's inversion of control (ioc) container, there's a comprehensive and easy way to modularize and hence customize or extend SpringContracts behaviour due to your own needs.

Base technologies

The following technologies are mainly used in order to drive SpringContracts

- AOP in order to intercept method calls for contract validation
- Java 5 Annotations for an easy way of applying contract elements to classes, interfaces and methods (although there are other ways like external contract definition and assignment)
- A specification language interpreter, which evaluates the single conditions, expressed in that language.

General workflow

Following is a characteristic high level sequence description on whats happened when a method gets called – the base entrance point for contract validation.
A reference and short characterisation of the underlying classes or interfaces which are mainly responsible for a single task are given with each described activity:

- Intercept method call
  Using AOP, SpringContracts provides an aspect resp. method interceptor to hook into the control flow of an application, weaving in contract validation.

  Class **`org.springcontracts.dbc.interceptor.ContractValidationInterceptor`** can be used as a Spring AOP method interceptor or as an AspectJ AOP Aspect (see chapter 'Activating SpringContracts' for a more detailled description about these two AOP styles)

- Delegate to **`org.springcontracts.dbc.validation.ContractValidator`**
  Validation is delegated to a ContractValidator, which is a kind of supervisor for contract validation.
  The creation and delivery of the related contract in relation to the intercepted method is again delegated to a **`org.springcontracts.dbc.contractContractBuilder`**, which is a kind of Strategy, based on the specification language in use.
  The delivered contract is represented by a **`org.springcontracts.dbc.contract.Contract,`** which aggregates some **`org.springcontracts.dbc.contract.ContractElementAttribute.`**

- Detect all invariants of the class the intercepted method belongs to and all preconditions and postconditions of the called class.

  Detection of contract elements relevant to the called method is delegated to an implementation of **`org.springcontracts.dbc.source.ContractElementAttributeSource.`**
  In case of annotation based detection, the implementation to do the job is **`org.springcontracts.dbc.annotation.AnnotationContractAttributeSource,`** which will also detect contract elements on all superclasses and all direct or indirect implemented interfaces.

- Ask ALL found contract elements before method execution if the condition of that contract element is violated (of course, postconditions will always answer true, but do some preparation due to the validation after the methods execution, i.e. acquire the value of some class properties before the method call, in order to refer to them by using the 'old' keyword).

- Collect all violations and handle them
  Processing of potentially ocurring violations is delegated to an implementation of **`org.springcontracts.dbc.validation.ContractViolationHandler,`**
  The default implementation will throw a **`org.springcontracts.dbc.validation.ContractViolationException`**

- Execute the method

- Ask postconditions and invariants after method execution if the condition of that contract element is violated.

- Again, collect all Violations and handle them (see above).

- return to the methods caller (returning the methods return value)

Chapter 2 – Setup

Download current distribution

Download the current distribution *springcontracts-<current-version>-with dependencies.zip*
via http://sourceforge.net/projects/springcontracts.

The distribution will usually come with all needed third party libraries SpringContracts depends on, so you don't have to gather them all over the web.

Unpack the distribution

Unpack the zip file in a free folder of your choice.
You will find a new folder *springcontracts-<current-version>-with dependencies* with the following subfolders:

- /dist
  contains the current distribution of SpringContracts as a jar file (only class files)
- /doc
  contains documentation about SpringContracts
- /lib
  contains all third party libraries, SpringContracts depends on
- /src
  contains all source files of the current distribution
- /test
  contains all source files of all unit tests

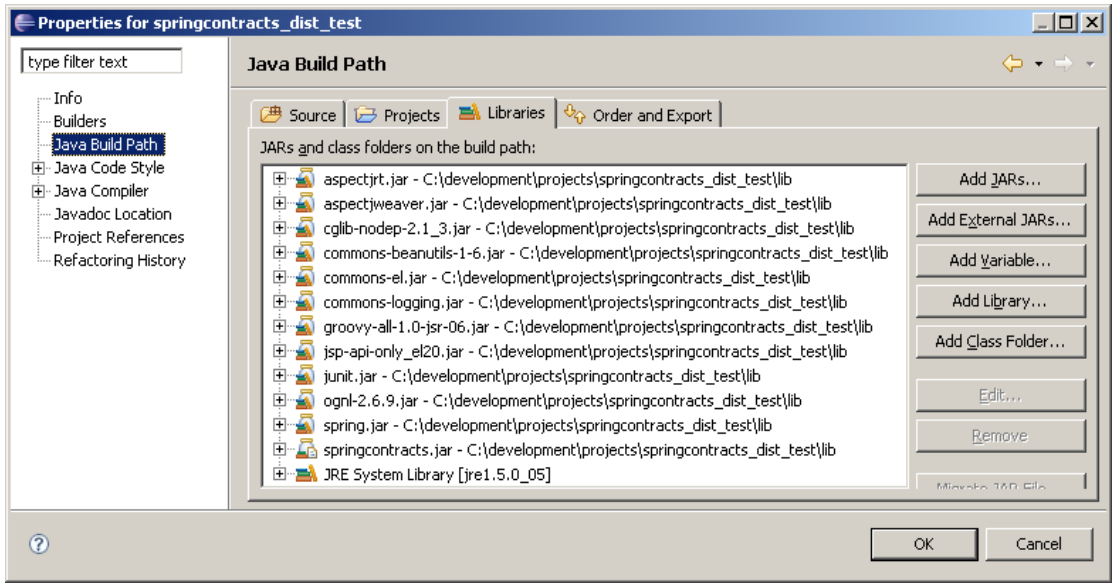
Read and agree to the licence

SpringContracts is an open source project under the Apache licence, version 2.0.
Please read the /licence.txt and agree to the licence.

Setup classpath

You have to extend the classpath, in that you add /dist/springcontracts.jar and all jars under /lib in order to use SpringContracts properly.

Following is an example on setting the classpath for a java project within eclipse (menu 'Project – Properties')



Using SpringContracts

Activating SpringContracts

To enable contract validation at runtime, there has to be an aspect, that intercepts method calls in order to validate all contract elements relevant to that method. This aspect is represented by class **`org.springcontracts.dbc.interceptor.ContractValidationInterceptor`**.
Because Spring provides a comprehensive foundation for an easy configuration style, we use Spring's ApplicationContext to register and setup that contract validation aspect.
It's completely sufficient to activate that aspect solely. In that case SpringContracts will run with a default behaviour, that is detecting contract elements on Annotations, evaluating conditions using Expression Language and throwing Exceptions on contract violations.

Since Spring 2.0 comes with full integration of AspectJ, there are now two options for using AOP within Spring - Spring AOP or AspectJ AOP.

Spring AOP

Using the proxy based Spring AOP model for method interception is simply done by declaring the aspect configuration (pointcut and advice type declaration) and the bean which contains the advice logic:

```
<aop:config>
     <aop:aspect ref="contractValidationAspect">

       <aop:pointcut  id="contractValidatingMethods"
                      expression="execution(* *(..))"/>

       <aop:around pointcut-ref="contractValidatingMethods"
                   method="validateMethodCall"/>
     </aop:aspect>
</aop:config>

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

As you can see, the pointcut declaration in this example refers to at least all method calls.
It's a good idea to limit the pointcuts to only that bundle of classes, you really want to have validatet by SpringContracts, otherwise all other method calls gets advised, too.
For example, if all your 'contracted' classes reside under the package com.mycompany.*, it makes sense to narrow the pointcut definition to only all method calls of classes under this package resp. subpackages:

```
<aop:pointcut id="contractValidatingMethods"
              expression="execution(* com.mycompany..*(..))"/>
```

Note, that Spring AOP is proxy based and hence only advices method calls of beans, that are also declared in Spring's ApplicationContext.
Ad hoc instantiations of beans outside the control of Springs container get not adviced and hence not validated (since Spring doesn't control the lifecycle of such beans, it can't apply a MethodInterceptor to a newly created bean). Because proxy based interception is limited to method calls, constructors don't get advised, too.

As a conclusion, if you only want to validate 'contracted' beans that are declared within Spring's ApplicationContext and don't need constructor validation, Spring AOP is the right choice for you.

AspectJ AOP

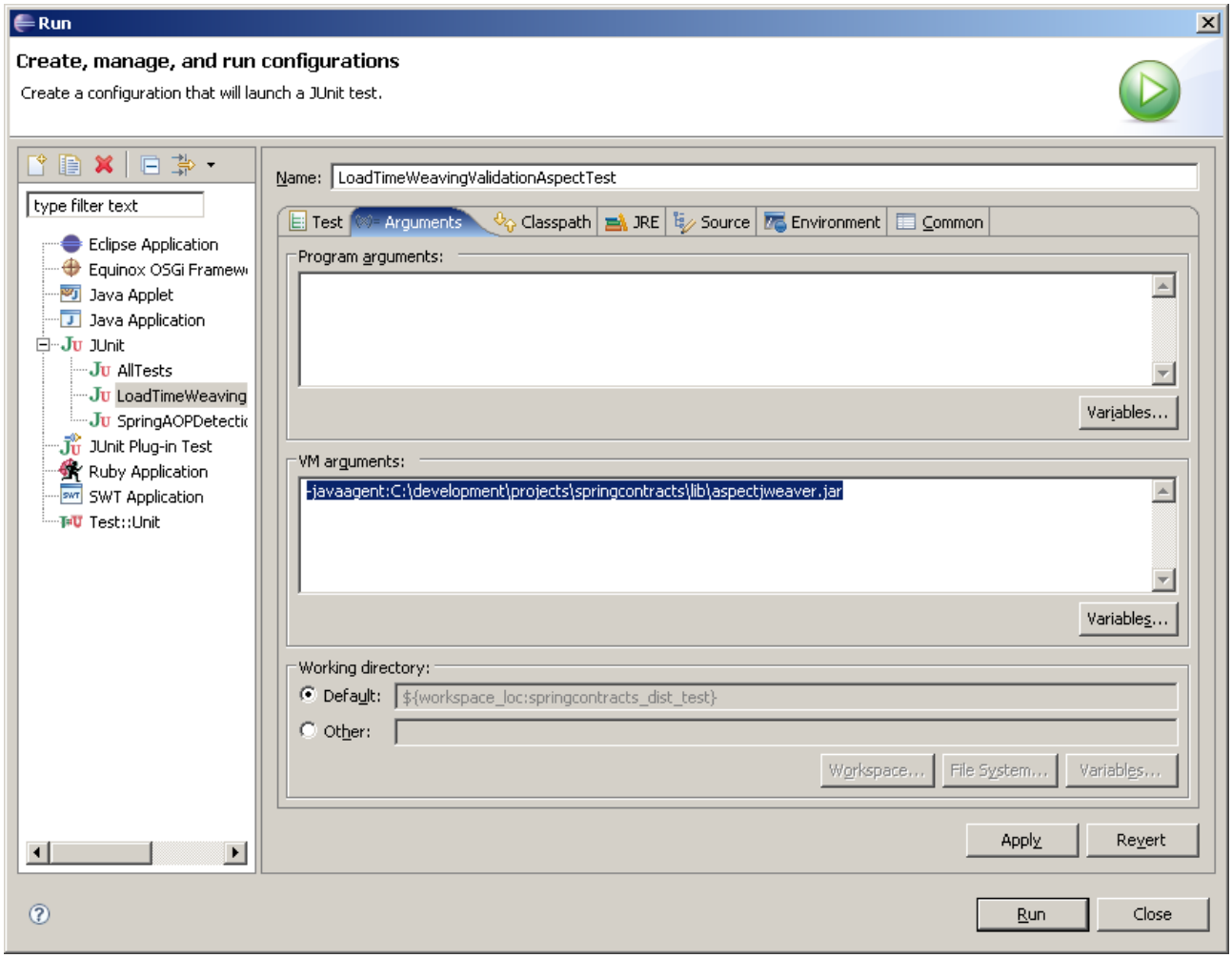
With the integration of AspectJ into Spring 2.0, it's not only possible to use AspectJ's pointcut model (which is also usable for pointcut declarations within Spring AOP), but also it's weaving mechanisms, still using Springs potentials for a customized configuration of SpringContracts due to specific needs.
Using AspectJ's weaving capabilities, it's possible to also advice constructors and beans which are created outside of Spring's ApplicationContext, using the new operator (this is often the case for domain objects or objects that are created by ORM tools for example).

Load Time Weaving (LTW)

One of AspectJ's weaving options is to weave in aspects at the time a class gets loaded into a VM by a classloader. In that case, the VM has to be startet with a special weaving agent, that gets called whenever a new class is to be loaded by the classloader. This is simply done by giving the VM a hint to the weaver agent at startup, pointing to aspectjweaver.jar.

```
java ... -javaagent:<path-to-lib>aspectjweaver.jar
```

Following is an example of how to configure the VM arguments within eclipse (menu 'Run – Run ...').



Configuring the Weaver Agent

The weaver agent has to be set up with the aspects, he ought to weave into the loading classes. Within that configuration goes the space of all classes he should consider for waeving.
These settings are configured in a file named ***aop.xml*** that the agent expects in a directory **/META-INF**:

```
<?xml version="1.0"?>
<aspectj>
  <aspects>
    <aspect name="org.springcontracts.dbc.interceptor.ContractValidationInterceptor"/>
  </aspects>
  <weaver>
    <exclude within="org.springframework..*"/>
    <exclude within="org.apache..*"/>
    <exclude within="org.springcontracts..*"/>
    <include within="com.mycompany..*"/>
  </weaver>
</aspectj>
```

It's again class **org.springcontracts.dbc.interceptor.ContractValidationInterceptor**, that represents the aspect to weave in. If using the class in this way, it's not necessary to declare any pointcut or advice type since ContractValidationInterceptor will do this for you. All method calls and constructor calls get automatically adviced within the classes that are specified as includes (or at least not specified as excludes) within **aop.xml**.
Once again it's a good idea to narrow the classes to advice, using the **include** tag.

```
  <include within="com.mycompany..*"/>
```

Configuring the Validator Aspect

Again, if you want to use the default behaviour of SpringContracts, there's nothin more to do!
The aspect doesn't have to be mentioned in Spring's ApplicationContext, since the weaver agent is now responsible for creating and weaving the aspect.
Nevertheless, if you still want to customize SpringContracts behaviour, you can reference the aspect inside the ApplicationContext. This is easily done via the aspects factory method aspectOf, since there is only one single instance of **`org.springcontracts.dbc.interceptor.ContractValidationInterceptor`** in use:

```
<bean id="contractValidationAspect"
     class="org.springcontracts.dbc.interceptor.ContractValidationInterceptor"
     factory-method="aspectOf">

     <property name="contractValidator" ref="contractValidator"/>
</bean>

<bean id="contractValidator"
     class="org.springcontracts.dbc.validation.ContractValidator">

   <constructor-arg>
      <bean class="org.springcontracts.groovy.contract.GroovyContractBuilder">
         <property name="contractViolationHandler" ref="violationHandler"/>
      </bean>
   </constructor-arg>

   <constructor-arg>
      <bean class="org.springcontracts.dbc.annotation.AnnotationContractAttributeSource"/>
   </constructor-arg>
</bean>

<bean id="violationHandler" class="com.mycompany.logging.MyViolationHandler"/>
```

In this example, we refer to the aspect in order to use groovy as specification language and set up a customized **`org.springcontracts.dbc.validation.ContractViolationHandler`** (please note, that future releases will provide some NameSpaceHandler, so that it will be much more concise to customize SpringContracts)

If you want to customize SpringContracts and use LTW, there's one important Note to remember:
The aspect isn't configured until Spring's applicationContext is created. All method calls before the instantiation of Spring's ApplicationContext gets adviced by the aspect as well, but in an unconfigured state. Therefore it's important to instantiate the ApplicationContext at startup in order to work with a properly configured aspect (should you be in need of a customized behaviour).

Expressing conditions

Conditions are boolean expressions, used to specify the single preconditions, postconditions and invariants.
While preserving the constraints of the contract, they can refer to some elements of its surrounding context, like the instance and methods of the class they're applied to, or the single arguments or return value of a method call.

Conditions are expressed in a special specification language. Therefore, the power of the expressiveness of such conditions depends on the potentials of the underlying specification language. At the moment, SpringContratcs come with build in support for Expression Language (Apache Commons EL), Groovy and OGNL.
However, regardless of which specification language you choose, the keywords to refer to the surrounding context should remain the same.

this

Most of the time, conditions refer to the actual instance of a class a contract element is applied to or some members of that class' instance like properties or methods, in order to constrain their state.
The keyword **`this`** stands for the current instance of a class, a method gets intercepted for contract validation.
Depending on the underlying specification language, you can refer to the class' members via the **`this`** reference.

Revisiting the example's interface Stack, we now can form the invariant, now using **`this`** in order to refer the properties **`size`** and **`capacity`**:

```
this.size <= this.capacity
```

EL as well as Groovy and OGNL allow this kind of shortcut notation in order to refer to properties which offer a corresponding getter method. Of course you also could use the method notation, in order to refer to the getter method.

```
this.getSize() <= this.getCapacity()
```

Normally, EL doesn't allow method calls, but thanks to SpringContracts you can use simple (not nested at this time) method calls within EL expressions as well.

arg

Having preconditions and postconditions on method calls, you may want to refer to the actual arguments of that call using the keyword **`arg<n>`**.
The arguments of a method are numbered, beginning with number 1, so **`arg1`** refers to the first argument, **`arg2`** refers to the second argument and so on.

Looking at one part of methods push() postcondition, we now can state, that the first (and only) argument has to be on top of the Stack after the methods execution.

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

Note, that we used again the shortcut notation in order to refer to the property **`top`** of interface **`Stack`**.

Symbolic argument names

Sometimes you want to use an expressive argument name other than arg<n> when refering to arguments of a method. SpringContracts offers the possibility to map arguments to symbolic argument names (a so called arg binding) and refer to that argument by its symbolic name inside a condition.

```
newElement == this.top        ( with arg-Binding : arg1 -> newElement )
```

We will come back to that feature when looking in detail about how to apply contract elements to classes and methods with annotations.

args

As we will see, you sometimes want to make a statement about all arguments without worrying about the actual number of arguments (for example when stating, that all arguments dont have to be **`null`**).
In that case you can refer to the keyword **`args`**, which represents a collection of all arguments of a method.

We'll have a closer look at keyword **`args`** when introducing quantors.

return

When stating a postcondition, you sometimes want to consider the return value. Keyword **`return`** lets you refer to the actual return value of the called method after execution. Of course you can only refer to the return value in postconditions.

If you for example want to state, that the return value of a method is never **`null`**, you can do so.

```
not empty return
```

Note, that the above syntax is only available with EL.
If you decide to use OGNL or Groovy as specification language, you can use a syntax more similar to Java.

```
return != null
```

old

Properties or other elements of a called method (for example its arguments) can change their value by the effects of the methods execution (ok, altering arguments isn't a good idea).
In order to reflect that change, you can reference and 'catch' the value of such an element before the method executes and refer to that value after the method has finished his task. Of course this makes only sense in postconditions.
In this case you can prefix an arbitrary subexpression inside your condtition with keyword **`old:`**.
Keep in mind, that the resulting value of such a subexpression have to be cloneable! Otherwise SpringContracts isn't able to catch and hold the value of an element beyond the methods execution until the postcondition gets validatet.

As an example, we could state, that pushing an element onto the Stack will increase the size of that Stack by one.

```
this.size == old:this.size + 1
```

The subexpression **`this.size`** is prefixed with keyword **`old:`** on the right side of that equation, refering to the value of property **`size`** before the method call. Since the resolution of that subexpression results in a value of type **`int`** , representet as a **`java.lang.Integer`** inside the interpreter (and hence is **`Cloneable`**), we can fetch that value before the method call and keep it for validation after the method call.
Because the same subexpression on the left side isn't prefixed, it represents the current value of property **`size`**. Since we apply the above expression within a postcondition, it represents the effect of the method call on property **`size`**.

First order logic

Since all conditions represent a boolean expression over all, expressing the semantic behaviour of a software component, it's very concise to have some constructs due to first order logic. SpringContracts provides some operators, that can be used within conditions.

All quantor

All quantor makes a statement that has to be true for all elements of a collection. Therefore it's only legal to apply an quantor to a subexpression that represents a java.util.Collection or an Array at least.
In order to refer to the single elements inside the quantor statement, you have to project the elements to a temporary variable.

Let's say we have an interface **`Bank`** that holds a set of **`Account`** objects, offering a corresponding getter which returns a collection of that accounts.
`Account` itself offers a property **`balance`** of type **`int`**, which represents the current balance of that account.
We now could state an invariant for a certain type of bank, that all of its managed accounts always have to have a balance that is greater than zero.

```
all( account : this.accounts,  account.balance > 0 )
```

We refer to the collection of accounts using the shortcut notation **`this.account`** and project them to the temporary variable **`account`**. We than refer to an account in the following statement (starting after the colon). This statement has to be true for every single account within the collection.

Another often used example constrains the arguments of a method not to be null. Given the keyword **`args`**, which represents a collection of all arguments (no matter of the actual number of arguments), we can refer to them using the all quantor.

```
all( argument : args,  argument != null )           // OGNL style !
```

Note, that it's possible to nest quantors to an arbitrary level of nesting.

```
all( account : this.accounts, all( transaction : account.transactions, ... ) )
```

Exist quantor

The exist quantor has the same syntax and domain like the all quantor, but with a different semantic when it comes to the resolution of the whole expression. The expression is true, if there is at least one element in the 'quantored' collection, that fulfils the quantor's statement.

For example we could equally express the above example of stating that all arguments don't have to be null using negation and the exist quantor

```
!exist( argument, args,  argument == null )
```

Of course it's allowed to nest quantors in arbitrary combined ways.

```
all( account : this.accounts, exist( transaction : account.transactions, ... ) )
```

Implication

Implication is a boolean operation that is most of the time used in a wrong way, like 'IF A THEN B'.
In most scenarios this may be all you want express, but this is only half of the truth, as implication is a shortcut for 'NOT A OR B' (means for example, that if A is not given (false) and B is given (true) the whole implication results also to true).
Nevertheless, SpringContracts offers the usage of an implication operator **==>** (which is internally transformed into the above mentioned boolean expression). Note that you have to put both antecedent and consequent in parentheses, in order to let SpringContracts detect the boundaries of the expression properly.

Applying contract elements

TODO

Annotations

TODO

```
import org.springcontracts.dbc.annotation.Invariant;
import org.springcontracts.dbc.annotation.Postcondition;
import org.springcontracts.dbc.annotation.Precondition;

@Invariant( condition="this.balance >= -500" )
public interface Account {

   public int getBalance();

   @Precondition( bindArgs="arg1=amount",
                  condition="amount > 0 and amount <= this.balance" )

   @Postcondition(bindArgs="arg1=amount",
                  condition="return == amount and this.saldo == old:this.saldo - amount")
   public int withdraw( int amount );

   @Precondition( bindArgs="arg1=amount",
                  condition="amount > 0" )

   @Postcondition(bindArgs="arg1=amount",
                  condition="this.saldo == old:this.saldo + amount" )
   public void deposit( int amount );

   @Precondition( bindArgs="arg1=amount, arg2=receiver",
                  condition="amount > 0 and amount <= this.saldo and not empty receiver")

   @Postcondition(bindArgs="arg1=amount, arg2=receiver",
                  condition="this.saldo == old:this.saldo - amount
                             and receiver.saldo == old:receiver.saldo + amount" )
   public void transfer( int amount, Account receiver );
}
```

External contract definition

TODO

```
<bean id="contractValidationAspect"
      class="org.springcontracts.dbc.interceptor.ContractValidationInterceptor">
   <property name="contractValidator">
      <bean class="org.springcontracts.dbc.validation.ContractValidator">
        <constructor-arg>
           <bean class="org.springcontracts.el.contract.ElContractBuilder" />
        </constructor-arg>
        <constructor-arg ref="externalContractAttributeSource" />
      </bean>
   </property>
</bean>
```

```
<bean id="externalContractAttributeSource"
      class="org.springcontracts.dbc.source.ext.ExternalContractElementAttributeSource">
   <constructor-arg>
      <list>
         <bean class="org.springcontracts.dbc.source.ext.ExtTypeContract">
            <constructor-arg>
               <value>org.springcontracts.dbc.detection.spring.extcontractconfig.CalculatorImpl
               </value>
            </constructor-arg>
            <property name="invariants">
               <list>
                  <bean class="org.springcontracts.dbc.source.ext.ExtCondition">
                     <property name="condition">
                        <value>this.mode &gt; 0 and this.mode &lt; 6</value>
                     </property>
                     <property name="message"><value>calc mode between 1 and 5</value></property>
                  </bean>
               </list>
            </property>
            <property name="methodContracts">
               <list>
                  <bean class="org.springcontracts.dbc.source.ext.ExtMethodContract">
                     <constructor-arg><value>public int add( int, int )</value></constructor-arg>
                     <property name="preconditions">
                        <list>
                           <bean class="org.springcontracts.dbc.source.ext.ExtCondition">
                              <property name="condition">
                                 <value>arg1 &gt; 0 and arg2 &gt; 0</value>
                              </property>
                              <property name="message"><value>positive args</value></property>
                           </bean>
                        </list>
                     </property>
                     <property name="postconditions">
                        <list>
                           <bean class="org.springcontracts.dbc.source.ext.ExtCondition">
                              <property name="condition"><value>return &gt; 0</value></property>
                              <property name="message"><value>positive result</value></property>
                           </bean>
                        </list>
                     </property>
                  </bean>
               </list>
            </property>
         </bean>
         ...
      </list>
   </constructor-arg>
</bean>
```

Custom Namespace for external Contract definition

TODO

```
<?xml version="1.0" encoding="UTF-8"?>
<beans  xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xmlns:ext-contract="http://springcontracts.sourceforge.net/schema/ext-contract"
     xsi:schemaLocation=
      "http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop.xsd
       http://springcontracts.sourceforge.net/schema/ext-contract
       http://springcontracts.sourceforge.net/schema/ext-contract/ext-contract.xsd">
   ...
   <bean id="externalContractAttributeSource"
     class="org.springcontracts.dbc.source.ext.ExternalContractElementAttributeSource">
     <constructor-arg>
       <list><ref bean="contract1"/></list>
     </constructor-arg>
   </bean>

   <ext-contract:contract id="contract1"
      type="org.springcontracts.dbc.detection.spring.extcontractconfig.CalculatorImpl">

     <ext-contract:invariant condition="this.mode &gt; 0 and this.mode &lt; 6"
                             message="calc mode between 1 and 5" />

     <ext-contract:method signature="public int add( int, int )">
       <ext-contract:precondition  condition="arg1 &gt; 0 and arg2 &gt; 0"
                                   message="positive args"/>

       <ext-contract:postcondition condition="return &gt; 0" message="positive result"/>
     </ext-contract:method>

     <ext-contract:method
       signature="public java.lang.Integer sub( java.lang.Integer, int )">
       <ext-contract:precondition condition="arg1 &gt; arg2"
                                   message="minuend greater subtrahend"/>

       <ext-contract:postcondition condition="return &gt; 1"
                                    message="result greater 1"/>
     </ext-contract:method>
   </ext-contract:contract>
...
```

TODO Example: Applying external contract elements to standard classes
(i.e. don't allow null value keys for java.util.Map)

```
<ext-contract:contract id="mapContract" type="java.util.Map">

   <ext-contract:method
      signature="public java.lang.Object put( java.lang.Object, java.lang.Object )">

      <ext-contract:precondition condition="not empty arg1  and  not empty arg2"
                                 message="key and value not null"/>
   </ext-contract:method>

</ext-contract:contract>
```

Writing custom contract definition styles

TODO - ContractElementAttributeSource

TODO Example: ContractElementAttributeSource for property-file based contract definitions
(i.e. invariant.full.qualified.classname=this.size < this.capacity #invariant
precondition.full.qualified.classname.methodname(int,int)= arg1 > 1 #precondition
postcondition.full.qualified.classname.methodname(int,int)=this.size == old:this.size + 1 #postcondition)

TODO Example: ContractElementAttributeSource for alternative annotations
(@require, @ensure)

Registering custom AnnotationAttributeSource

TODO Description: Registering via Spring's ApplicationContext

TODO Example: Registering property-file based AnnotationAttributeSource

Messages

TODO – Define Messages using annotations
TODO – Refer to surrounding context (called class properties / methods, args, return value)

```
@Invariant( condition="this.size >= 0"
            message="Size always greater zero (current size: {this.size})")

@Precondition( condition="arg1 > arg2"
               message="first argument {arg1} should be greater than second {arg2}")

@Postcondition( condition="return.size() > 0"
                message="returned collection not empty (current size: {return.size})")
```

TODO Example: Extend property-file based ContractElementAttributes, supporting messages

Named Constraints

TODO - Definition

Simple named constraints

TODO (syntax)

TODO Example: &allArgsNotNull

TODO Example: &property:, &hasValue: -> condition: &property:name &hasValue:'Hans'

Defining named constraints

TODO

TODO Example: Definition of named constraints in Spring's ApplicationContext

```
<bean id="contractValidator" class="org.springcontracts.dbc.validation.ContractValidator">
   <constructor-arg>
      <bean class="org.springcontracts.el.contract.ElContractBuilder">
         <property name="namedConstraintResolver" ref="namedConstraintResolver"/>
      </bean>
   </constructor-arg>
   <constructor-arg>
      <bean class="org.springcontracts.dbc.annotation.AnnotationContractAttributeSource"/>
   </constructor-arg>
</bean>

<bean id="namedConstraintResolver"
      class="org.springcontracts.dbc.contract.enhancements.NamedConstraintResolver">
   <property name="namedConstraints">
      <list>
         <bean class="org.springcontracts.dbc.contract.enhancements.NamedConstraint">
            <constructor-arg index="0"><value>&amp;greater_zero</value></constructor-arg>
            <constructor-arg index="1"><value>&gt; 0</value></constructor-arg>
         </bean>
         <bean class="org.springcontracts.dbc.contract.enhancements.NamedConstraint">
            <constructor-arg index="0"><value>&amp;all_args_set</value></constructor-arg>
            <constructor-arg index="1"><value>all(arg:args, not empty arg)</value>
            </constructor-arg>
         </bean>
         <bean class="org.springcontracts.dbc.contract.enhancements.NamedConstraint">
            <constructor-arg index="0"><value>&amp;not_equal(x,y)</value></constructor-arg>
            <constructor-arg index="1"><value>x != y</value></constructor-arg>
         </bean>
      </list>
   </property>
</bean>
```

Refering named contraints

TODO

TODO Example: Usage of above defined named constraints in (i.e. pre-) conditions

```
public class Grid {

   @Precondition( condition="&all_args_set and arg2 &greater_zero and arg2 < 100" )
   public void placeCard( Integer cardNumber, int pos ){
   }

   @Precondition( condition="arg1 > 0 and &not_equal( arg1, arg2 ) and arg2 < 100" )
   public void move( int fromPos, int toPos ){
   }
}
```

Projection

TODO

TODO Example: Definition &alwaysPositive(x)

TODO Example: Definition &allGreater(collection, property, value)
-> all(elem : collection, elem.property > value)
-> condition: &allGreater(this.adresses, streetNumber, 20)

Writing a custom NamedContraintResolver

TODO

Registering custom NamedContraintResolver

TODO (registering in Spring's ApplicationContext)

Specification Language

TODO

Switching specification language

TODO

TODO Example: Switching to OGNL / Groovy by configuring Spring's ApplicationContext

```
<bean id="contractValidationAspect"
      class="org.springcontracts.dbc.interceptor.ContractValidationInterceptor">
   <property name="contractValidator">
      <bean id="contractValidator"
            class="org.springcontracts.dbc.validation.ContractValidator">
        <constructor-arg>
           <bean class="org.springcontracts.groovy.contract.GroovyContractBuilder"/>
        </constructor-arg>
        <constructor-arg>
           <bean class="org.springcontracts...AnnotationContractAttributeSource"/>
        </constructor-arg>
      </bean>
   </property>
</bean>
```

Writing a custom ContractBuilder for your preferred language

TODO

TODO Example: Contract Builder for JRuby

Contract violations

SpringContracts comes with a default behaviour on contract violations in that a ContractViolationException is thrown.

Generally this will be an instance of ContractViolationCollectionException, which will collect all single ViolationExceptions, which occured during the check before or after a method execution. Every violation of a single precondition will result in a separate PreconditionViolationException, every invariant violation will result in separate InvariantViolationException and everyl violation of a single postcondition will result in PostconditionValidationExceptions.

Note, that violations based on preconditions and violations based on postconditions will never be collected in the same ContractViolationCollectionException.
If the contract validation before a methods execution results at least in an InvariantContractException or a PreconditionException, an ContractViolationCollectionException is thrown an the method will never be called.

If the method was executed, all preconditions were statisfied. Hence validation after method execution can only result in PostconditionExceptions or InvariantContractExceptions.

TODO - ContractViolationException

Writing a custom ContractViolationHandler

TODO

TODO Example: LoggingContractViolationHandler

```
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.ApplicationEventPublisherAware;

public class ApplicationEventPublishingViolationHandler
   implements ContractViolationHandler, ApplicationEventPublisherAware {

   private ApplicationEventPublisher publisher = null;

   public void handleViolation(ContractViolationException violationException) {
      if( publisher != null ){
         publisher.publishEvent( new ContractViolationEvent( this, violationException ) );
      }
   }

   public void setApplicationEventPublisher( ApplicationEventPublisher publisher ) {
      this.publisher = publisher;
   }
}
```

Registering a custom ContractViolationHandler

TODO

```
  <bean id="contractValidator"
        class="org.springcontracts.dbc.validation.ContractValidator">

    <constructor-arg>
      <bean class="org.springcontracts.el.contract.ElContractBuilder">
        <property name="contractViolationHandler" ref="violationHandler"/>
      </bean>
    </constructor-arg>

    <constructor-arg>
      <bean class="org.springcontracts...AnnotationContractAttributeSource"/>
    </constructor-arg>
  </bean>

<bean id="violationHandler"
class="org.springcontracts.dbc.validation.ApplicationEventPublishingViolationHandler"/>
```

Misc examples

TODO