Design By Contracts for Spring

v. 0.1.8-SNAPSHOT

Project Documentation

UniKnow **2015-04-09**

Table Of Content

1 Introduction DBC

Introduction Design By Contract.

The principle of *design by contract* was first introduced by Bertrant Meyer and the basic idea is modules in software systems should be designed to follow contracts. The contracts shall specify under which circumstances a method can be used. Most methods request some parameter(s) to be sent to the method when it is invoked. Sometimes it is necessary that the parameters have certain values for the method to be able to execute and produce correct output. The contract is all about specifying acceptable values for the parameters both before and after the execution of the method.

Contracts have suppliers and consumers. The supplier can be a method or a class that already is written and provides some functionality to other developers. The supplier code will be called from other parts of the system when it is needed. The code that calls the supplier will be the client.

In order to verify that the contract is followed by both the supplier and the client, constraints will be inserted during the design of the contract. An example of an constraint would be that a certain integer should have a positive value. Constraints are basically boolean expressions.

1.1 Preconditions and Postconditions.

A method contract is a set of constraints that are evaluated before and after the execution of one method. The precondition that is spoken of are the constraints that must be fulfilled before the code in a method is executed. A postcondition is a guaranteed state or value that the result must satisfy when the code has been executed. The pre- and postconditions function as a guarantee for both the client and supplier.

A simple contract is shown below. The client of the method must send a parameter of type `int` that is smaller than `MAX_VALUES`. In return the client can be certain that that the method returns the value placed at the requested postion in the array.

```
@Named
@Validated
public class SimpleMethodContract {

    /**
     * Max number of values that can be persisted.
     */
    public static final int MAX_VALUES = 10;

    private int values[] = new int[MAX_VALUES];

    public int getValue(@Max(MAX_VALUES - 1) @Min(0) final int index) {
        return values[index];
    }

}
```

If a method has a precondition which say that a `index` must be within a certain range (`0` .. `MAX_INDEX`), then every programmer must ensure that the method is invoked with a `index` which is within the range. That is the obligation of the client. A consequence of the constraint is that the method may start executing its code without verifying that the `index` is within the range (this is done by the framework), which is a clear benefit. In case of a postcondition the client knows that the

method returns a valid value (verified by framework) and that the instance is in a state according to the postconditions; e.g. there is no need for the client to verify the output.

As a summary you can say that if the client invoke a method with a satisfied precondition then the method promise to deliver an output that is satisfied according to the postcondition. On the other hand, if the postcondition is not satisfied, a `ValidationException` is raised at the client and it is there that the error must be fixed, not in the supplier. A `ValidationException` during postcondition means a fault in the supplier and it must be fixed in the invoked method. The constraints help the developer to find the error and to fix it in the right space.

1.2 Invariant

An invariant is a certain constraint that must be true for all methods that are grouped into an class. There could for example be an class which represents a house. A house can contain one of more rooms which have a certain area. Lets say there exist several methods that can change the characteristics of a house and a room. Then the house class will need an invariant which says that the summation of rooms areas must be the same as the total area of the house.

Invariants are used to ensure that the properties of the entity always have correct values andt that the present state is correct regardless of which method is invoked.

Assertions like preconditions and postconditions are used to verify the input and output of a single method. An invariant, must be verified on every call to a method of a class to ensure the correct properties.

In this example, the invariant guarantees that the `PositiveInteger`'s value is always greater than or equal to zero.

```
/**
 * Example of an invariant that guarantees that value is positive
 */
@Named
@Validated
@Scope(BeanDefinition.SCOPE_PROTOTYPE)
public class PositiveInteger {

    @Min(0)
    private int value = 0;

    public void add(int addedValue) {
        value += addedValue;
    }

    public void subtract(int subtractedValue) {
        value -= subtractedValue;
    }

    public int toInt() {
        return value;
    }
}
```

2 Constraints

Validation Constraints

The `dcb4spring` framework supports constraints in the form of annotations placed on a field, method or class of a spring managed bean.

The list below lists all the built-in constraints.

Constraint	Description	Example
`@AssertFalse`	The value of the field or property must be `false`	`@AssertFalse boolean supported;`
`@AssertTrue`	The value of the field or property must be `true`	`@AssertTrue boolean supported;`
`@DecimalMax`	The value of the field or property must be a decimal value lower than or equal to the number in the value element. Supported types are `BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, `long`, and their respective wrappers. **Note** that `double` and `float` are not supported due to rounding errors	`@DecimalMax("30.00") BigDecimal discount;`
`@DecimalMin`	The value of the field or property must be a decimal value greater than or equal to the number in the value element. Supported types are `BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, `long`, and their respective wrappers. **Note** that `double` and `float` are not supported due to rounding errors	`@DecimalMin("5.00") BigDecimal discount;`
`@Digits`	The value of the field or property must be a number within a specified range. The `integer` element specifies the maximum integral digits for the number, and the `fraction` element specifies the maximum fractional digits for the number. Supported types are `BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int` and their respective wrapper types. `null` elements are considered valid.	`@Digits(interger=2, fraction=2) BigDecimal price;`
`@Email`	The value of the field or property must be a well-formed email address.	`@Email String emailAddress`
`@Future`	The value of the field or property must be a date in the future.	`@Future Date eventDate;`

`@Max`	The value of the field or property must be an integer value lower than or equal to the number in the value element	`@Max(10)` `int quantity;`
`@Min`	The value of the field or property must be an integer value greater than or equal to the number in the value element	`@Min(5)` `int quantity;`
`@NotBlank`	The value of the field or property must not be `null` or empty. The difference to `@NotEmpty` is that trailing whitespaces are getting ignored.	`@NotBlank` `String username;`
`@NotEmpty`	The value of the annotated `String`, `Collection`, `Map` or array must not be `null` or empty.	`@NotEmpty` `String username;`
`@NotNull`	The value of the field or property must not be `null`.	`@NotNull` `String userName;`
`@Null`	The value of the field or property must be `null`.	`@Null` `Object unusedValue`
`@Past`	The value of the field or property must be a date in the past	`@Past` `Date birthday;`

`@Pattern`	The value of the field or property must match the regular expression defined in the `regexp` element	`@Pattern(regexp="\\(\`
		`\d{3}\\)\\d{3}-\\d{4}")`
		`String phoneNumber;`

`@Size`	The size of the field or property is evaluated and must match the specified boundaries. If the field or property is `String`, the size of the string is evaluated. If the field or property is a `Collection`, `Map` or array, the size of the entity is evaluated. Use one of the optional `max` or `min` elements to specify the boundaries	`@Size(min=2, max=10)` `String identifier;`
`@URL`	THe value of the field or property must be a valid URL.	`@URL` `String url;`

See Using bean validation

3 Usage dbc4spring

Usage `dbc4java`

To use `dbc4java` within your project you need to include the following dependency within your project.

Classes for which its constraints need to be verified require to be annotated with `@Validated` and the constraints on fields, methods, etc are also added as annotations to the class/method.

```
@Named
@Validated
@Scope(BeanDefinition.SCOPE_PROTOTYPE)
public class PositiveInteger {

    @Min(0)
    private int value = 0;

    public void add(int addedValue) {
        value += addedValue;
    }

    public void subtract(int subtractedValue) {
        value -= subtractedValue;
    }

    public int toInt() {
        return value;
    }
}
```

In the example above we assure by the constraint `@Min(0)` that the integer is always positive.

3.1 Usage standalone

3.2 Usage within Spring

To assure that the constraints are validated at every method invocation the dbc4java spring configuration file need to be imported.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <!-- Import dbc4spring configuration file -->
    <import resource="classpath:validation-spring-config.xml" />

    <context:component-scan base-package="org.uniknow.agiledev.dbc4spring"/>
</beans>
```