JSR 303: Bean Validation

1.0.Beta3.20090128
2009-01-28

Authors:

Bean Validation Expert Group

Emmanuel Bernard (Red Hat Middleware, LLC)

Steve Peterson

Table of Contents

Disclaimer

This specification is in public draft stage. The content of this specification is subject to change. Specific areas currently at work include:

- XML deployment descriptor
- Extension for method parameters validation (see Appendix C)
- Exception cases

The JSR 303 expert group is seeking for feedbacks from the community on this specification. Direct feedbacks to http://forum.hibernate.org/viewforum.php?f=26. If privacy is a concern, consider using jsr-303-comments@jcp.org.

1

Introduction

This document is the specification of the Java API for JavaBean validation in Java EE and Java SE. The technical objective of this work is to provide a class level constraint declaration and validation facility for the Java application developer, as well as a constraint metadata repository and query API.

1.1. Expert group

This work is being conducted as part of JSR-303 under the Java Community Process Program. This specification is the result of the collaborative work of the members of the JSR 303 Expert Group. These include the following present and former expert group members:

- Geert Bevin
- Emmanuel Bernard (Red Hat Middleware LLC) - Specification Lead
- Uri Boness
- Erik Brakkee (Ericsson AB)
- Ed Burns (Sun Microsystems, Inc.)
- Jason Carreira
- Robert Clevenger (Oracle - retired)
- Linda DeMichiel (Sun Microsystems, Inc.)
- Tim Fennel
- Bharath Ganesh (Pramati Technologies)
- Romain Guy (Google Inc.)
- Robert Harrop
- Jacob J. Hookom
- Bob Lee (Google Inc.)
- Craig R. McClanahan (Sun Microsystems, Inc.)
- Niall K. Pemberton

- Steve Peterson
- Dhanji R. Prasanna (individual initially and then Google Inc.)
- Gerhard Petracek
- Matt Raible
- Michael Nascimento Santos
- Sebastian Thomschke
- Jon Wetherbee (Oracle)

1.2. Specification goals

Validating data is a common task that occurs throughout an application, from the presentation layer to the persistence layer. Often the same validation logic is implemented in each layer, proving time consuming and error-prone. To avoid duplication of these validations in each layer, developers often bundle validation logic directly into the domain model, cluttering domain classes with validation code that is, in fact, metadata about the class itself.

This JSR defines a metadata model and API for JavaBean validation. The default metadata source is annotations, with the ability to override and extend the meta-data through the use of XML validation descriptors.

The validation API developed by this JSR is not intended for use in any one tier or programming model. It is specifically not tied to either the web tier or the persistence tier, and is available for both server-side application programming, as well as rich client Swing application developers. This API is seen as a general extension to the JavaBeans object model, and as such is expected to be used as a core component in other specifications. Ease of use and flexibility are influencing the design of this specification.

1.3. How this document is organized

This document describes each aspect of the bean validation specification in a separate chapter. One should remember that the specification is a consistent whole.

Chapter 2 describes how constraints are defined.

Chapter 3 describes how a JavaBean class is decorated with annotations to describe constraints.

Chapter 4 describes how to programmatically validate a JavaBean.

Constraint metadata request API describes how the metadata query API works.

In Appendix A, key concepts are summarized. Some reviewers have found that reading the terminology section first helps to better understand the specification.

JPA, JSF and EE integration proposals are present at Appendix D, Appendix E and Appendix F.

1.4. How to comment

The expert group is eager to receive feedbacks from readers. Feel free to contact us at http://forum.hibernate.org/viewforum.php?f=26. If privacy is a concern, consider using jsr-303-comments@jcp.org.

2

Constraint Definition

Constraints are defined by the combination of a constraint annotation and a constraint validation implementation. The constraint annotation is applied on types, methods, or fields.

Unless stated otherwise the default package name for the Bean Validation APIs is `javax.validation`.

2.1. Constraint annotation

A constraint on a JavaBean is expressed through one or more annotations. An annotation is considered a constraint definition if its retention policy contains `RUNTIME` and if the annotation itself is annotated with `javax.validation.Constraint`.

```
/**
 * Link between a constraint annotation and its constraint validation implementations.
 * <p/>
 * A given constraint annotation should be annotated by a @Constraint
 * annotation which refers to its list of constraint validation implementation.
 *
 * @author Emmanuel Bernard (emmanuel at hibernate.org)
 * @author Gavin King
 * @author Hardy Ferentschik
 */
@Documented
@Target({ ANNOTATION_TYPE })
@Retention(RUNTIME)
public @interface Constraint {
        /**
         * ConstraintValidator classes must reference distinct target types.
         * If two validators refer to the same type, an exception will occur
         *
         * @return aray of ConstraintValidator classes implementing the constraint
         */
        public Class<? extends ConstraintValidator<?,?>>[] validatedBy();
}
```

Constraint annotations can target any of the following `ElementTypes`:

- `FIELD` for constrained attributes
- `METHOD` for constrained getters
- `TYPE` for constrained beans
- `ANNOTATION_TYPE` for constraints composing other constraints

While other `ElementTypes` are not forbidden, the provider does not have to recognize and process constraints

placed on such types.

Since a given constraint definition applies to one or more specific Java types, the JavaDoc for the constraint annotation should clearly state which types are supported. Applying a constraint annotation to an incompatible type will raise a `UnexpectedTypeForConstraintException`. Care should be taken on defining the list of `ConstraintValidator`. The type resolution algorithm (see Section 3.5.3) could lead to exceptions due to ambiguity.

2.1.1. Constraint definition properties

A constraint definition may have properties that can be specified at the time the constraint is applied to a JavaBean. The properties are mapped as annotation elements. The annotation element names `message` and `groups` are considered reserved names; annotation elements starting with `valid` are not allowed; a constraint may use any other element name for its parameters.

2.1.1.1. message

Every constraint annotation must define a `message` element of type `String`.

```
String message() default "{constraint.myConstraintFailure}";
```

The `message` element value is used to create the error message. See Section 4.3 for a detailed explanation.

2.1.1.2. groups

Every constraint annotation must define a `groups` element that specifies which processing groups the constraint definition is associated with.

```
    Class<?>[] groups() default {};
```

The default value must be an empty array.

When using the constraint declaration, if no group is specified, the `Default` group is considered declared. The validation will be evaluated if:

- no group was specified at validation time (which defaults to the `Default` group) and the annotation declares implicitly or explicitly the `Default` group
- or if one of the groups specified at validation time matches one of the groups specified on the constraint annotation declaration

See Section 4.1.2 for more information.

Groups are typically used to control the order of Validator evaluation, or perform validation of the partial state of a JavaBean.

2.1.1.3. Constraint specific parameter

The constraint annotation definitions may define additional elements to parameterize the constraint. For example, a constraint that validates the length of a string can use an annotation element named `length` to specify the maximum length at the time the constraint is declared.

2.1.2. Examples

Example 2.1. @NotNull constraint definition

```
@Documented
@Constraint(validatedBy = NotNullConstraintValidator.class)
@Target({METHOD, FIELD, ANNOTATION_TYPE})
@Retention(RUNTIME)
public @interface NotNull {
    String message() default "{constraint.notNull}";
    Class<?>[] groups() default {};
}
```

Example 2.1 defines a not null constraint with a specific default message. The constraint Validator is implemented by `NotNullConstraint`.

Example 2.2. @Length constraint definition

```
@Documented
@Constraint(validatedBy = LengthConstraintValidator.class)
@Target({METHOD, FIELD, ANNOTATION_TYPE})
@Retention(RUNTIME)
public @interface Length {
    int min() default 0;
    int max() default Integer.MAX_VALUE;
    String message() default "{constraint.length}";
    Class<?>[] groups() default {};
}
```

Example 2.2 defines a length constraint. The constraint definition includes two optional properties that may be specified when the constraint is applied.

Example 2.3. @Min constraint definition

```
@Documented
@Constraint(validatedBy = MinConstraintValidator.class)
@Target({METHOD, FIELD, ANNOTATION_TYPE})
@Retention(RUNTIME)
public @interface Min {
    int value();
    String message() default "{constraint.min}";
    Class<?>[] groups() default {};
}
```

Example 2.3 defines a min constraint: the `value` property must be specified when the constraint is applied.

2.2. Applying multiple constraints of the same type

It is often useful to apply the same constraint more than once to the same target, with different properties. A com-

mon example is the `@Pattern` constraint, which validates that its target matches a specified regular expression.

To support this, the bean validation provider treats regular annotations (annotations not annotated by `@Constraint`) but whose a `value` element has a return type of an array of constraint annotations in a special way. Each element in the `value` array are processed by the Bean Validation implementation as regular constraint annotations. This means that each constraint specified in the value element is applied to the target. The annotation must have retention `RUNTIME` and can be applied on a type, field, property or an other annotation.

Note to constraint designers. Each constraint annotation should be coupled with its corresponding multi-valued annotation.

```
@Documented
@Constraint(validatedBy = PatternValidator.class)
@Target({METHOD, FIELD, ANNOTATION_TYPE})
@Retention(RUNTIME)
public @interface Pattern {
    /** regular expression */
    String regex();

    /** Flags parameter for Pattern.compile() */
    int flags() default 0;

    String message() default "{constraint.pattern}";
    Class<?>[] groups() default {};
}

@Documented
@Target({METHOD, FIELD, ANNOTATION_TYPE})
@Retention(RUNTIME)
public @interface Patterns {
    Pattern[] value();}
```

Example 2.4. Multi-valued constraint

```
public class Engine {
    @Patterns( {
            @Pattern(regex = "^[A-Z0-9-]+$",
                message = "must contain alphabetical characters only"),
            @Pattern(regex = "^....-....-....$", message="must match ....-....-....")
            } )
    private String serialNumber;
}
```

In this example, both constraints (`^[A-Z0-9-]+$` and `^....-....-....$`) will be applied on the `serialNumber` field.

2.3. Constraint composition

The specification allows to compose constraints from other constraints.

Constraint composition is useful in several ways:

- Avoid duplication and facilitate reuse of more primitive constraints.

- Expose primitive constraints as part of a composed constraint in the metadata API and enhance tool awareness.

Composition is done by annotating a constraint annotation with constraint annotations.

Example 2.5. Composition consists of annotating the composed constraint

```
@NotNull
@Size(min=5, max=5)
@Constraint(validatedBy = FrenchZipcodeValidator.class)
@Documented
@Target({ANNOTATION_TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface FrenchZipcode {
    String message() default "Wrong zipcode";
    Class<?>[] groups() default {};
}
```

Annotating an element with `@FrenchZipcode` is equivalent to annotating it with `@NotNull`, `@Size(min=5, max=5)` and `@FrenchZipcode`. More formally, each constraint annotation hosted on a constraint annotation is applied to the target element and this recursively. Note that the main annotation and its constraint validation implementation is also applied. By default, each failing constraint generates an error report. Groups from the main constraint annotation are inherited by the composing annotations. Any `groups` definition on a composing annotation is ignored.

It is possible to ensure that composing annotations do not raise individual error reports. In this scenario, if one or more composing annotations are invalid, the main annotation is automatically considered invalid and the corresponding error report is generated. To mark a constraint as raising only the main constraint error report despite its use of composing constraints, use the `@ReportAsViolationFromCompositeConstraint` annotation.

Example 2.6. If any of the composing constraint fails, the error report corresponding to @FrenchZipcode is raised and none other.

```
@NotNull
@Size(min=5, max=5)
@ReportAsViolationFromCompositeConstraint
@Constraint(validatedBy = FrenchZipcodeValidator.class)
@Documented
@Target({ANNOTATION_TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface FrenchZipcode {
    String message() default "Wrong zipcode";
    Class<?>[] groups() default {};
}
```

The definition of `@ReportAsViolationFromCompositeConstraint` is as followed.

```
/**
 * A constraint annotation annotated with this annotation
 * will return the composed annotation error report if any of the composing annotations
 * fail. The error reports of each individual composing constraint is ignored.
 *
 * @author Emmanuel Bernard
 */
public @interface ReportAsViolationFromCompositeConstraint {
```

```
}
```

Composing annotations can define parameters and message (aside from `groups`) but these are fixed for a given main annotation.

Example 2.7. Composing annotations can use parameters. They are fixed for a given main annotation. All @FrenchZipcode constraints have a @Size restricted to 5.

```
@NotNull
@Size(min=5, max=5)
@Constraint(validatedBy = FrenchZipcodeValidator.class)
@Documented
@Target({ANNOTATION_TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface FrenchZipcode {
    String message() default "Wrong zipcode";
    Class<?>[] groups() default {};
}
```

It is possible to override parameters and messages defined on a composing annotation. A parameter from the main annotation is used to override one or more parameters of the composing annotations. Such a parameter is annotated with the `@OverridesParameter` annotation or its multivalued equivalent `@OverridesParameters`.

Example 2.8. Parameters from composing annotations can be overridden by parameters from the main annotation.

```
@NotNull
@Size
@Constraint(validatedBy = FrenchZipcodeValidator.class)
@Documented
@Target({ANNOTATION_TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface FrenchZipcode {
    String message() default "Wrong zipcode";
    Class<?>[] groups() default {};

    @OverridesParameters( {
        @OverridesParameter(constraint=Size.class, parameter="min"),
        @OverridesParameter(constraint=Size.class, parameter="max") } )
    int size() default 5;

    @OverridesParameter(constraint=Size.class, parameter="message")
    String sizeMessage() default "{beancheck.zipcode.size}";
}
```

The parameter value from the main annotation annotated with `@OverridesParameter` is applied to the parameter named after `OverridesParameter.parameter` of the composing constraint of type `OverridesParameter.constraint`. If not specified, the name of the targeted parameter equals the name of the parameter the `@OverridesParameter` is on The types of the overridden and overriding parameters must be equals.

Using Example 2.8,

```
@FrenchZipcode(size=9, sizeMessage="Zipcode should be of size {max}")
```

is equivalent to

```
@FrenchZipcode
```

using the following definition

```
@NotNull
@Size(min=9, max=9, message="Zipcode should be of size {max}")
@Constraint(validatedBy = FrenchZipcodeValidator.class)
@Documented
@Target({ANNOTATION_TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface FrenchZipcode {
    String message() default "Wrong zipcode";
    Class<?>[] groups() default {};
}
```

If a constraint type is used more than once as a composing constraint, the construct described in Section 2.2 is used. To select the appropriate composing constraint targeted, `OverridesParameter.index` is used. It represents the constraint index in the `value` array. If `index` is undefined, the single constraint declaration is targeted.

Example 2.9. @OverridesParameter annotation definition

```
/**
 * Mark a parameter as overriding the parameter of a composing constraint.
 * Both parameter must share the same type.
 *
 * @author Emmanuel Bernard
 */
public @interface OverridesParameter {
    /**
     * constraint type the parameter is overriding
     */
    Class<? extends Annotation> constraint();

    /**
     * name of constraint parameter overridden
     * Defaults to the name of the parameter hosting the annotation
     */
    String parameter();

    /**
     * index of the targetted constraint declaration when using
     * multiple constraints of the same type.
     * The index represents the index of the constraint in the value() array.
     *
     * By default, no index is defined and the single constraint declaration
     * is targeted
     */
    int index() default -1;
}
```

The following elements uniquely identify an overridden constraint parameter:

- `OverridesParameter.constraint`

- `OverridesParameter.parameter`
- `OverridesParameter.index`

Constraint designers are encouraged to make use of composition (recursively or not) based on the built-in constraints defined by the specification. The composing constraints are exposed through the Bean Validation metadata API (Section 5.5). This metadata is particularly useful for third-party metadata consumers like persistence frameworks generating database schemas (such as Java Persistence) or presentation frameworks.

2.4. Constraint validation implementation

A constraint validation implementation performs the validation of a given constraint annotation for a given type. The implementation classes are specified by the `validatedBy` element of the `@Contraint` annotation that decorates the constraint definition. The constraint validation implementation implements the `ConstraintValidator` interface.

Example 2.10. ConstraintValidator interface

```
/**
 * Defines the logic to validate a given constraint A
 * for a given object type T.
 * Implementations must comply to the following restriction:
 * T must resolve to a non parameterized type
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface ConstraintValidator<A extends Annotation, T> {
        /**
         * Validator parameters for a given constraint definition
         * Annotations parameters are passed as key/value into parameters
         * <p/>
         * This method is guaranteed to be called before any of the other Constraint
         * implementation methods
         *
         * @param constraintAnnotation parameters for a given constraint definition
         */
        void initialize(A constraintAnnotation);

        /**
         * Implement the validation constraint.
         * <code>object</code> state must not be changed by a Constraint implementation
         *
         * @param object object to validate
         * @param constraintValidatorContext context in which the constraint is evaluated
         *
         * @return false if <code>object</code> does not pass the constraint
         */
        boolean isValid(T object, ConstraintValidatorContext constraintValidatorContext);
}
```

Some restrictions apply on the generic type `T` (used in the `isValid` method). T must resolve in a non parameterized type:

- the type is not using generics

- because the raw type is used instead of the generic version

Warning

Should we support unbounded wildcards?

Here are some examples of valid definitions in Example 2.11.

Example 2.11. Valid ConstraintValidator definitions

```
//String is not making use of generics
public class SizeValidatorForString implements<Size, String> {...}

//Collection uses generics but the raw type is used
public class SizeValidatorForCollection implements<Size, Collection> {...}
```

And some invalid definitions in Example 2.12.

Example 2.12. Invalid ConstraintValidator definitions

```
//parameterized type
public class SizeValidatorForString implements<Size, Collection<String> {...}

//parameterized type using unbounded wildcard
public class SizeValidatorForCollection implements<Size, Collection<?>> {...}

//parameterized type using bounded wildcard
public class SizeValidatorForCollection implements<Size, Collection<? extends Address>> {...}
```

Note

This restriction is not a theoretical limitation and a future version of the specification will likely allow that in the future.

The life cycle of a constraint validation implementation instance is undefined. Compliant implementations are allowed to cache `ConstraintValidator` instances retrieved from the `ConstraintValidatorFactory` or request them every time it is needed.

Warning

Should we restrict further? Should a constraint implementation instance be retrieved from the `ConstraintValidatorFactory` before each use? Providing the constraint via the Metadata API was driving the undefined lifecycle rule but we no longer return the implementation via `ConstraintDescriptor`.

`initialize` is called by the Bean validation provider prior to any use of the constraint implementation.

The `isValid` method is evaluated by the Bean Validation provider each time a given value is validated. It returns `false` if the value is not valid, `true` otherwise. `isValid` implementations must be thread-safe.

If the value parameter references an object with an unanticipated type, an `UnexpectedTypeForConstraintException` should be raised. While Validator implementation can raise this exception themselves, constraint designer are encouraged to make use of specialized Validator implementation and delegate the type routing to the type matching algorithm described in Section 3.5.3. The constraint validation implementation is not allowed to change the state of the value passed to `isValid`.

Note

While not mandatory, it is considered a good practice to split the core constraint validation from the not null constraint validation (for example, an `@Email` constraint will return true on a null object, i.e. will not take care of the `@NotNull` validation)

`null` can have multiple meanings but is commonly used to express that a value does not make sense, is not available or is simply unknown. Those constraints on the value are orthogonal in most cases to other constraints. For example a String, if present, must be an email but can be null. Separating both concerns is a good practice.

The `ConstraintValidatorContext` object passed to the `isValid` method carries information and operations available in the context the constraint is validated to.

Example 2.13. ConstraintValidatorContext interface passed to ConstraintValidator.isValid()

```
/**
 * Provide contextual data and operation when applying a given constraint validator implementation
 *
 * @author Emmanuel Bernard
 */
public interface ConstraintValidatorContext {
        /**
         * Disable default error message and default ConstraintViolation object generation.
         * Useful to set a different error message or generate an ConstraintViolation based on
         * a different property
         *
         * @see #addError(String)
         * @see #addError(String, String)
         */
        void disableDefaultError();

        /**
         * @return the current unexpanded default message
         */
        String getDefaultErrorMessage();

        /**
         * Add a new error message. This error message will be interpolated.
         * <p/>
         * If isValid returns false, a ConstraintViolation object will be built per error message
         * including the default one unless #disableDefaultErrorMEssage() has been called.
         * <p/>
         * Aside from the error message, ConstraintViolation objects generated from such a call
         * contains the same contextual information (root bean, path and so on)
         * <p/>
         * This method can be called multiple time. One ConstraintViolation instance per
         * call is created.
         *
         * @param message new unexpanded error message
         */
        void addError(String message);
```

```
        /**
         * Add a new error message to a given sub property <code>property</code>.
         * This error message will be interpolated.
         * <p/>
         * If isValid returns false, a ConstraintViolation object will be built
         * per error message including the default one unless #disableDefaultErrorMEssage()
         * has been called.
         * If the constraint being validated is not a class-level constraint,
         * a ValidationException is raised.
         *
         * <p/>
         *
         * @param message new unexpanded error message
         * @param property property name the ConstraintViolation is targeting
         *
         * @throws ValidationException if the constraint is not set on a class-level
         */
        void addError(String message, String property);

}
```

The `ConstraintValidatorContext` interface allows to redefine the default message error generated when a constraint is not valid. By default, each invalid constraint leads to the generation of one error object represented by a `ConstraintViolation` object. This object is build from the default error message as defined by the constraint declaration and the context in which the constraint declaration is placed on (bean, property, attribute).

The `ConstraintValidatorContext` methods let the constraint implementation disable the default error object generation and create one or more custom ones. The unexpanded message passed as a parameter is used to build the `ConstraintViolation` object (the message resolution operation is applied to it). The property on which the error object is applied is defined as following:

- if the property is not overridden, the current context the constraint is declared on is used (bean or property)
- if the property is overridden, the current context the constraint is declared on the property passed as a parameter relative to the bean the constraint declaration is on
- if the property is overridden and the constraint is declared on a property, a `ValidationException` exception is raised
- if the overridden property is not present on the bean, a `ValidationException` is raised.

The property can be overridden by calling `ConstraintValidatorContext.addError(String, String)`.

2.4.1. Example

Example 2.14. Length constraint Validator

```
/**
 * Check that a string length is between min and max
 *
 */
public class LengthConstraintValidator implements ConstraintValidator<Length, Sting> {
    private int min;
```

```
    private int max;

    /**
     * Configure the constraint validator based on the elements
     * specified at the time it was defined.
     * @param constraint the constraint definition
     */
    public void initialize(Length constraint) {
        min = constraint.min();
        max = constraint.max();

    }

    /**
     * Validate a specified value.
     * returns false if the specified value does not conform to the definition
     */
    public boolean isValid(String value, ConstraintValidatorContext context) {
        if ( value == null ) return true;

        int length = value.length();
        return length >= min && length <= max;
    }
}
```

This constraint Validator checks that the string's length is between min and max. It also demonstrates some best practices:

- returns `true` on a null parameter

The next example shows how to use `ConstraintValidatorContext`.

Example 2.15. Use of ConstraintValidatorContext

```
/**
 * Check that a string length is between min and max
 * Error messages are using either key:
 *  - constraint.length.min if the min limit is reached
 *  - constraint.length.max if the max limit is reached
 */
public class FineGrainedLengthConstraintValidator
             implements ConstraintValidator<Length, String> {
    private int min;
    private int max;

    /**
     * Configure the constraint validator based on the elements
     * specified at the time it was defined.
     * @param constraint the constraint definition
     */
    public void initialize(Length constraint) {
        min = constraint.min();
        max = constraint.max();
    }

    /**
     * Validate a specified value.
     * returns false if the specified value does not conform to the definition
     */
    public boolean isValid(String value, ConstraintValidatorContext context) {
```

```
        if ( value == null ) return true;
        int length = value.length();

        //remove default error
        context.disableDefaultError();

        if (length < min) {
            //add min specific error
            context.addError( "{constraint.length.min}" );
            return false;
        }
        if (length > max) {
            //add max specific error
            context.addError( "{constraint.length.max}" );
            return false;
        }
        return true;
    }
}
```

The default error message is disabled and replaced by a specific error message depending on the type of constraint violation detected. In this case, only one error report is returned at a given time but a constraint validation implementation can return several error reports.

2.5. The ConstraintValidatorFactory

Constraint validation implementation instances are created by a `ConstraintValidatorFactory`.

```
/**
 * Instantiate a <code>ConstraintValidator</code> instance from its class.
 * The <code>ConstraintValidatorFactory</code> is <b>not</b> responsible
 * for calling {@link ConstraintValidator#initialize(java.lang.annotation.Annotation)}.
 *
 * @author Dhanji R. Prasanna
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface ConstraintValidatorFactory {

        /**
         * @param key The class of the constraint validator to instantiate.
         *
         * @return An constraint validator instance of the specified class.
         */
        <T extends ConstraintValidator<?,?>> T getInstance(Class<T> key);
}
```

The default `ConstraintValidatorFactory` provided by the Bean Validation provider implementation uses the public constraint no-arg constructor. A custom constraint Validator factory can be provided for example to benefit from dependency injection control in constraint implementations. Any constraint implementation relying on `ConstraintValidatorFactory` behaviors specific to an implementation (dependency injection, no no-arg constructor and so on) are not considered portable, hence great care should be given before walking that path.

3

Constraint declaration and validation process

The Bean Validation specification defines a framework for declaring constraints on JavaBean classes, fields and properties.

Constraints are declared for classes, and evaluated against instances or graphs of instances.

3.1. Requirements on classes to be validated

Objects that are to be validated must fulfill the following requirements.

Properties to be validated must follow the method signature conventions for JavaBeans read properties, as defined by the JavaBeans specification.

Static fields and static methods are excluded from validation.

Constraints can be applied to interfaces and superclasses.

The target of an annotation definition can be a field, property, or type, provided that:

- the constraint definition supports the specified target (`java.lang.annotation.Target`)
- the constraint supports the declared type of the target (see Section 3.5.3).

3.1.1. Object validation

Constraint declarations can be applied to a class or an interface. Applying a constraint to a class or interface expresses a validation over the state of the class or interface.

3.1.2. Field and property validation

Constraint declarations can be applied on both fields and properties for the same object type. The same constraint should however not be duplicated between a field and its associated property (the constraint validation would be applied twice). It is recommended for objects holding constraint declarations to adhere a single state access strategy (either annotated fields or properties).

When a field is annotated with a constraint declaration, field access strategy is used to access the state validated by such constraint.

When a property is annotated with a constraint declaration, property access strategy is used to access the state validated by such constraint.

When using field access strategy, the bean validation provider accesses the instance variable directly. When using the property access strategy, the bean validation provider accesses the state via the property accessor method. It is required that the class follow the method signature conventions for JavaBeans read properties (as defined by the JavaBeans `Introspector` class) for constrained properties when constrained properties are used. In this case, for every constraint property of type T, there is a getter method, `get<Property-name>`. For `boolean` properties, `is<Property-name>` is an alternative name for the getter method. Specifically, if `getX` is the name of the getter method, where X is a string, the name of the persistent property is defined by the result of `java.beans.Introspector.decapitalize(X)`.

The fields or methods visibility are not constrained.

3.1.3. Graph validation

In addition to supporting instance validation, validation of graphs of object is also supported. The result of a graph validation is returned as a single list of constraint violations.

Consider the situation where bean X contains a field of type Y. By annotating field Y with the `@Valid` annotation, the Validator will validate the contents of Y when X is validated. The exact type of the field of type Y (subclass, implementation) is determined at runtime. The constraint definitions for this particular type are used. This ensures proper polymorphism behavior.

Collection-valued or array-valued fields and properties may also be decorated with the `@Valid` annotation. This causes the contents of the collection or array to be validated. The following types are supported:

- any array of object
- `java.util.Collection`
- `java.util.Set`
- `java.util.List`
- `java.util.Map`

Each object stored in the array or collection is validated. For `Map`, the value of each entry is validated (the key is not validated). Like regular references, it's type is determined at runtime and the constraint definitions for this particular type are used.

The `@Valid` annotation is applied recursively. A conforming implementation avoids infinite loops by validating an object the first time it is encountered in the graph, and ignores it if it is encountered through at different path.

Warning

Question: this rule does not guarantee a full deterministic behavior if an object is present several times in the graph. Should we adjust the rule to ensure deterministic behavior at the risk of validating an object more than one time, but still preventing infinite loop?

3.2. Constraint declaration

Constraint declarations are placed on classes or interfaces primarily through annotations. A constraint annotation (see Section 2.1), can be applied to a type, on any of the type's fields or on any of the JavaBeans-compliant properties.

When a constraint is defined on a type, the type instance being validated is passed to the constraint Validator. When a constraint is defined on a field, the value of the field is passed to the constraint Validator. When a constraint is defined on a getter, the result of the getter invocation is passed to the constraint Validator.

3.3. Inheritance (interface and superclass)

A constraint declaration can be placed on an interface. For a given class, constraint declarations held on superclasses as well as interfaces are evaluated by the Validator. Rules are formally described in Section 3.4.5.

The effect of constraint declarations is cumulative. Constraints declared on a superclass getter will be validated along with any constraints defined on an overridden version of the getter.

3.4. Group and group sequence

A group defines a subset of constraints. Instead of validating all constraints for a given object graph, only a subset is validated depending on the group targeted. Each constraint declaration defines the list of groups it belongs to. If no group is explicitly declared, a constraint belongs to the `Default` group.

Groups are represented by interfaces.

Example 3.1. Definition of groups

```
/**
 * Validation group checking a user is billable
 */
public interface Billable {}

/**
 * customer can buy without harrassing checking process
 */
public interface BuyInOneClick {
}
```

A constraint can belong to one or more groups.

Example 3.2. Assign groups to constraints

```
/**
 * User representation
 */
public class User {
    @NotNull
    private String firstname;

    @NotNull(groups = Default.class)
```

```
    private String lastname;

    @NotNull(groups = {Billable.class, BuyInOneClick.class})
    private CreditCard defaultCreditCard;
}
```

During the validation call, one or more groups are validated. All the constraints belonging to this set of group is evaluated on the object graph. In Example 3.2, `@NotNull` is checked on `defaultCreditCard` when either the `Billable` or `BuyInOneClick` group is validated. `@NotNull` on `firstname` and on `lastname` are validated when the `Default` group is validated. Reminder: constraints held on superclasses and interfaces are considered.

`Default` is a group predefined by the specification

```
package javax.validation.groups;

/**
 * Default Bean Validation group
 *
 * @author Emmanuel Bernard
 * TODO should it be named DefaultGroup?
 */
public interface Default {
}
```

3.4.1. Group inheritance

In some situations, a group is a super set of one or more groups. This can be described by Bean Validation. A group can inherit one or more groups by using interface inheritance.

Example 3.3. Groups can inherit other groups

```
/**
 * Customer can buy without harrassing checking process
 */
public interface BuyInOneClick extends Default, Billable {
}
```

For a given interface `Z`, constraints marked as belonging to the group `Z` (ie where the annotation `groups` property contains the interface `Z`) or any of the super interfaces of `Z` (inherited groups) are considered part of the group `Z`.

In the following example:

Example 3.4. Use of a inherited group

```
/**
 * User representation
 */
public class User {
    @NotNull
    private String firstname;

    @NotNull(groups = Default.class)
```

```
    private String lastname;

    @NotNull(groups = {Billable.class})
    private CreditCard defaultCreditCard;
}
```

validating the group `BuyInOneClick` will lead to the following constraints checking:

- `@NotNull` on `firstname` and `lastname`
- `@NotNull` on `defaultCreditCard`

because `Default` and `Billable` are subinterfaces of `BuyInOneClick`.

3.4.2. Group sequence

By default, constraints are evaluated in no particular order and this regardless of which groups they belong to. It is however useful in some situations to to control the order of constraints evaluation. There are often scenarios where a preliminary set of constraints should be evaluated prior to other constraints. Here are two examples:

- The second group depends on a stable state to run properly. This stable state is verified by the first previous group.
- The second group is a heavy consumer of time, CPU or memory and its evaluation should be avoided if possible.

To implement such ordering, a group can be defined as a sequence of other groups. Each group in a group sequence must be processed sequentially in the order defined by `@GroupSequence.sequence` when the group defined as a sequence is requested. Processing a group is defined in Section 3.5 ; if one of the groups processed in the sequence generates one or more constraint violation, the groups following in the sequence must not be processed. This ensure that a set of constraint is evaluated only if another set of constraint is valid.

Groups defining a sequence and groups composing a sequence must not be involved in a cyclic dependency either directly or indirectly, either through cascaded sequence definition or group inheritance.

Warning

Should we constraint group defining a sequence? Ie. can group defining a sequence be used in constraint declarations explicitly? Think about implications in the resolution algorithm.

To define a group as a sequence, the interface must be annotated with the `@GroupSequence` annotation.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface GroupSequence {
    Class<?>[] sequence();
}
```

Here is a usage example

Example 3.5. Make use of group sequence

```
@ZipCodeCoherenceChecker(groups = Address.HighLevelCoherence.class)
public class Address {
    @NotNull @Size(max = 50)
    private String street1;

    @ZipCode
    private String zipcode;

    @NotNull @Size(max = 30)
    private String city;

    /**
     * check conherence on the overall object
     * Needs basic checking to be green first
     */
    public interface HighLevelCoherence {}

    /**
     * check both basic constraints and high level ones.
     * high level constraints are not cheked if basic constraints fail
     */
    @GroupSequence(sequence = {Default.class, HighLevelCoherence.class})
    public interface Complete {}
}
```

In Example 3.5, when the `Address.Complete` group is validated, all constraints belonging to the `Default` group are validated. If any of them fail, the validation skips the `HighLevelCoherence` group. If all `Default` constraints pass, `HighLevelCoherence` constraints are evaluated.

Note

A given constraint can belong to two groups ordered by a sequence. In this case, the constraint is evaluated as part of the first group and ignored in the subsequent group(s). See Section 3.5 for more informations.

3.4.3. Redefining the Default group for a class

In Example 3.5, validating the `Default` group does not validate `HighLevelCoherence` constraints. To ensure a complete validation, a user must use the `Complete` group. This breaks some of the encapsulation you could expect. You can work around this by redefining what the `Default` group means for a given class. To redefine `Default` for a class, place a `@GroupSequence` annotation on the class ; this sequence expresses the sequence of groups that does substitute `Default` for this class.

Example 3.6. Redefining Default group for Address

```
@GroupSequence(sequence={Address.class, HighLevelCoherence.class})
@ZipCodeCoherenceChecker(groups = Address.HighLevelCoherence.class)
public class Address {
    @NotNull @Size(max = 50)
    private String street1;

    @ZipCode
    private String zipcode;
```

```
    @NotNull @Size(max = 30)
    private String city;

    /**
     * check conherence on the overall object
     * Needs basic checking to be green first
     */
    public interface HighLevelCoherence {}
}
```

In Example 3.6, when an address object is validated for the group `Default`, all constraints belonging to the group `Default` and hosted on `Address` are evaluated. If none fails, all `HighLevelCoherence` constraints present on `Address` are evaluated. In other words, when validating the `Default` group for `Address`, the group sequence defined on the `Address` class is used.

Since sequences cannot have circular dependencies, using `Default` in the declaration of a sequence is not an option. Constraints hosted on a class `A` and belonging to the `Default` group (by default or explicitly) implicitly belong to the group `A`.

A sequence defined on a class `A` (ie. redefining the `Default` groups for the class) must contain the group `A`. In other words, the default constraints hosted on a class must be part of the sequence definition.

3.4.4. Implicit grouping

It is possible to implicitly group some constraints in the same group without explicitly listing such a group in the constraint declaration. Every constraint hosted on an interface `Z` and part of the `Default` group (implicitly or explicitly) belongs to the group `Z`. This is useful to validate the partial state of an object based on a role represented by an interface.

Example 3.7. Example of interface / group hosting constraints

```
/**
 * Auditable object contract
 */
public interface Auditable {
    @NotNull String getCreationDate();
    @NotNull String getLastUpdate();
    @NotNull String getLastModifier();
    @NotNull String getLastReader();
}

/**
 * Represents an order in the system
 */
public class Order implements Auditable {
    private String creationDate;
    private String lastUpdate;
    private String lastModifier;
    private String lastReader;

    private String orderNumber;

    public String getCreationDate() {
        return this.creationDate;
    }
```

```
    public String getLastUpdate() {
        return this.lastUpdate;
    }

    public String getLastModifier() {
        return this.lastModifier;
    }

    public String getLastReader() {
        return this.lastReader;
    }

    @NotNull @Size(min=10, max=10)
    public String getOrderNumber() {
        return this.orderNumber;
    }
}
```

When an order object is validated on the `Default` group, the following constraints are validated: `@NotNull` on `getCreationDate`, `getLastUpdate`, `getLastModifier`, `getLastReader`, `getOrderNumber` and `@Size` on `getOrderNumber` as all belong to the `Default` group.

When an order object is validated on the `Auditable` group, the following constraints are validated: `@NotNull` on `getCreationDate`, `getLastUpdate`, `getLastModifier`, `getLastReader`. Only the constraints present on `Auditable` (and any of its super interfaces) and belonging to the `Default` group are validated when the group `Auditable` is requested. It allows the caller to validate that a given object can be safely audited even if the object state itself is not valid.

3.4.5. Formal definitions

The formal rules defining groups are as followed. *Text in italic are comments about the rules.*

For every class `X`:

- A. For each superclass `Y` of `X`, the group `Y` contains all constraints of the group `Y` of `Y`

  this rule prepares formal concepts for recursive discovery

- B. The group `X` contains the following constraints:

  group `X` is a group used on sequences redefining the default group on a class (see Section 3.4.3)

  1. every constraint declared by the class `X` which does not explicitly declare a group or declare the group `Default` explicitly.

     all `Default` constraints hosted on `X`

  2. every constraint declared by any interface implemented by `X` and not annotated `@GroupSequence` which does not explicitly declare a group or declare the group `Default` explicitly.

     all `Default` constraints hosted on interfaces of `X`: constraints are inherited by the class hierarchy

3. if `X` has a direct superclass `Y`, every constraint in the group `Y`

   all `Default` constraints hosted on the superclasses of `X`: constraints are inherited by the class hierarchy

- C. If `X` has no `@GroupSequence` annotation, the group `Default` contains the following constraints:

  this rule defines which constraints are evaluated when validating `Default` on `X`.

  1. every constraint in the group `X`
  2. if `X` has a direct superclass `Y`, every constraint in the group `Default` of `Y`

     this rule is necessary in case `Y` redefines the group `Default`

- D. If `X` does have a `@GroupSequence` annotation, the group `Default` contains every constraint belonging to every group declared by the `@GroupSequence` annotation.

  this rule describes how a class can redefine the group `Default` for itself (see Section 3.4.3)

  - the `@GroupSequence` annotation must declare the group `X`

- E. For every interface `Z`, the group `Z` contains the following constraints:

  this rule defines how non `Default` groups are defined

  1. every constraint declared by the interface `Z` which does not explicitly declare a group or declare the group `Default` explicitly.

     all `Default` constraints hosted on `Z`: this rule formally defines implicit grouping per interface (see Section 3.4.4)

  2. every constraint declared by any superinterface not annotated `@GroupSequence` of the interface `Z` which does not explicitly declare a group

     all `Default` constraints hosted on interfaces of `Z`: groups can be inherited (see Section 3.4.1)

  3. every constraint declared by the class `X` which explicitly declares the group `Z`

     every constraint hosted by `X` and marked as belonging to the group `Z`

  4. every constraint declared by any interface implemented by `X` and not annotated `@GroupSequence` which explicitly declares the group `Z`

     every constraint hosted by any interface of `X` and marked as belonging to the group `Z`

  5. if `X` has a direct superclass `Y`, every constraint in the group `Z` of `Y`

     every constraint hosted by any superclass of `X` and marked as belonging to the group `Z`

- F. For every interface `Z` annotated `@GroupSequence`, the group `Z` contains every constraint belonging to every

group declared by the `@GroupSequence` annotation.

defines the composition side of group sequence but does not define the ordering behavior of sequence (see Section 3.4.2)

3.5. Validation routine

For a given group to validate, the validation routine applied on a given bean instance is expected to execute the constraint validations:

- for all traversable fields, execute all field level validations (including the ones expressed on interfaces and superclasses) matching the targeted group unless the given validation constraint has already been processed during this validation routine (as part of a previous group match)
- for all traversable getters, execute all getter level validations (including the ones expressed on interfaces and superclasses) matching the targeted group unless the given validation constraint has already been processed during this validation routine (as part of a previous group match)
- execute all class level validations (including the ones expressed on interfaces and superclasses) matching the targeted group unless the given validation constraint has already been processed during this validation routine (as part of a previous group match)
- for all traversable associations, execute all cascading validations (see Section 3.5.1) including the ones expressed on interfaces and superclasses

Traversable fields, getters and associations are defined in Section 3.5.2.

Note that it implies that a given validation constraint will not be processed more than once per validation. More than one group can be validated during the same routine loop call provided that the ordering rules defined by sequence definitions are respected.

The object validation routine is described as such. For each constraint declaration and in the previously defined order:

- determine for the constraint declaration, the appropriate `ConstraintValidator` to use (see Section 3.5.3).
- execute the `isValid` operation (from the constraint validation implementation) on the appropriate data (see Section 2.4)
- if `isValid` returns true, continue to the next constraint,
- if `isValid` returns false, the Bean Validation provider populates `ConstraintViolation` object(s) according to the rules defined in Section 2.4 and appends these objects to the list of returned invalid violations.

Bean Validation is a fully polymorphic framework. Constraints are gathered according to the object type determined at runtime.

3.5.1. Object graph validation

The `@Valid` annotation on a given association (i.e. object reference or collection / array of objects), dictates the Bean Validator implementation to apply recursively the bean validation routine on (each of) the associated object(s). This mechanism is recursive: an associated object can itself contain cascaded references. The Bean Validation implementation must ignore the cascading operation if the associated object instance has already been validated by the current validation routine, thus preventing infinite loops.

The `ConstraintViolation` objects, built when a failing constraint on an associated object is found, reflects the path to reach the object from the root validated object (See Section 4.2).

3.5.2. Traversable property

In some cases, the state of some properties should not be accessed. For example, if a property loaded by a Java Persistence provider is a lazy property or a lazy association, accessing its state would trigger a load from the database. An undesired behavior.

Bean Validation offers a way to control which property can and cannot be accessed via the `TraversableResolver` contract.

```
/**
 * Contract determining if a property can be accessed by the Bean Validation provider
 * This contract is called for each property either validated or traversed.
 *
 * A traversable resolver implementation must me thread-safe.
 *
 * @author Emmanuel Bernard
 */
public interface TraversableResolver {
    /**
     * Determine if a property can be traversed by Bean Validation.
     *
     * @param traversableObject object hosting <code>traversableProperty</code>.
     * @param traversableProperty name of the traversable property.
     * @param rootBeanType type of the root object passed to the Validator.
     * @param pathToTraversableObject path from the root object to the <code>traversableProperty</code>
     *        (using the path specification defined by Bean Validator).
     * @param elementType either <code>FIELD</code> or <code>METHOD</code>.
     *
     * @return <code>true</code> if the property is traversable by Bean Validation, <code>false</code> ot
     */
    boolean isTraversable(Object traversableObject,
                          String traversableProperty,
                          Class<?> rootBeanType,
                          String pathToTraversableObject,
                          ElementType elementType);
}
```

`traversableObject` is the object instance being evaluated.

`traversableProperty` is the name of the property hosted by the `traversableObject` being considered for traversal. The name of a property is defined in Section 3.1.2.

`rootBeanType` is the class of the root being validated (and passed to the `validate` method).

`pathToTraversableObject` is the path from the `rootBeanType` down to the `traversableObject` (it is an empty string if the `rootBeanPath` is the `traversableObject`). The path is described following the conventions described in Section 4.2 (`getPropertyPath`).

`elementType` is the `java.lang.annotation.ElementType` the annotation is placed on. It can be either `FIELD` or `METHOD`. Any other value is not expected.

The Bean Validation provider must not access the state of a property, nor validate its constraints if the property is not traversable. A property is traversable if `TraversableResolver` returns true for this property.

The example assumes the following object graph

```
public class Country {
    @NotNull private String name;
    @Length(max=2) private String ISO2Code;
    @Length(max=3) private String ISO3Code;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getISO2Code() {
        return ISO2Code;
    }

    public void setISO2Code(String ISO2Code) {
        this.ISO2Code = ISO2Code;
    }

    public String getISO3Code() {
        return ISO3Code;
    }

    public void setISO3Code(String ISO3Code) {
        this.ISO3Code = ISO3Code;
    }
}

public class Address {
    @NotNull @Length(max=30)
    private String addressline1;
    @Length(max=30)
    private String addressline2;
    @Length(max=11)
    private String zipCode;
    @NotNull @Valid
    private Country country;

    private String city;

    public String getAddressline1() {
        return addressline1;
    }

    public void setAddressline1(String addressline1) {
        this.addressline1 = addressline1;
    }

    public String getAddressline2() {
        return addressline2;
    }

    public void setAddressline2(String addressline2) {
        this.addressline2 = addressline2;
```

```
    }

    public String getZipCode() {
        return zipCode;
    }

    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }

    @Length(max=30) @NotNull
    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public Country getCountry() {
        return country;
    }

    public void setCountry(Country country) {
        this.country = country;
    }
}
```

and assumes the validation operation is applied on an address object. When the Bean Validation provider is about to check constraints of `ISO3Code`, it calls the `TraversableResolver` instance to ensure that the `ISO3Code` property is traversable with the following parameter values:

- `traversableObject`: country. The instance returned by `address.getCountry()`.
- `traversableProperty`: "ISO3Code". The name of the property of `traversableObject` being verified.
- `rootBeanType`: Address.class. The type of the root object being validated.
- `pathtoTraversableObject`: "country". The path from address to the country instance.
- `elementType`: `ElementType.FIELD`. The country property is annotated on the field.

Example 3.8. Java Persistence aware TraversableResolver

```
public class JPATraversableResolver implements TraversableResolver {

    boolean isTraversable(Object traversableObject,
                          String traversableProperty,
                          Class<?> rootBeanType,
                          String pathToTraversableObject,
                          ElementType elementType) {

        return Persistence.isLoaded(traversableObject, traversableProperty);
    }
}
```

The traversable resolver used by default in a Bean Validation behaves as followed:

- if Java Persistence is available in the runtime environment, a property is considered traversable if Java Persistence considers the property as loaded. A typical implementation will use `Persistence.isLoaded(Object, String)` to implement such contract.
- if Java Persistence is not available in the runtime environment, all properties are considered traversable.

See Section 4.4 to know how to pass a custom `TraversableResolver`.

3.5.3. ConstraintValidator resolution algorithm

A constraint is associated to one or more `ConstraintValidator` implementations. Each `ConstraintValidator<A, T>` accepts the type `T`. The `ConstraintValidator` executed depends on the type declared by the target hosting the constraint. For a given constraint evaluation, a single `ConstraintValidator` is considered.

If the constraint is hosted on a class or an interface, the targeted type is the class or the interface. If the constraint is hosted on a class attribute, the type of the attribute is the targeted type. If the constraint is hosted on a getter, the return type of the getter is the targeted type.

The rules written below describe formally the following statement: the `ConstraintValidator` chosen to validate a declared type `T` is the one where the type supported by the `ConstraintValidator` is a supertype of `T` and where there is no other `ConstraintValidator` whose supported type is a supertype of `T` and not a supertype of the chosen `ConstraintValidator` supported type.

When validating a constraint A placed on a target declaring the type `T`, the following resolution rules apply.

- Primitive types are considered equivalent to their respective primitive wrapper class.
- A `ConstraintValidator<A, U>` is said to be *compliant* with `T` if `T` is a subtype of `U` (according to the Java Language Specification 3rd edition chapter 4.10 Subtyping [1]). Note that `T` is a subtype of `U` if `T = U`.
- If no `ConstraintValidator` compliant with `T` is found amongst the `ConstraintValidators` listed by the constraint `A`, a `UnexpectedTypeForConstraintException` is raised.
- A `ConstraintValidator<A, U>` compliant with `T` is considered *strictly more specific* than a `ConstraintValidator<A, V>` compliant with `T` if `U` is a strict subtype of `V`. `U` is a strict subtype of `V` if `U` is a subtype of `V` and `U != V` (according to theJava Language Specification 3rd edition chapter 4.10 Subtyping [2]).
- A `ConstraintValidator<A, U>` compliant with `T` is considered maximally specific if no other `ConstraintValidator<A, V>` compliant with `T` is strictly more specific than `ConstraintValidator<A, U>`.
- If more than one maximally specific `ConstraintValidator` is found, a `AmbiguousConstraintUsageException` is raised.

Note

While the Java compiler itself cannot determine if a constraint declaration will lead to a `UnexpectedTypeForConstraintException` or a `AmbiguousConstraintUsageException`, rules can be statically checked. A tool such as an IDE or a Java 6 annotation processor can apply these rules and prevent a compilation in case of ambiguity. The specification encourages Bean Validation provider to provide such a tool to their users.

[1] http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.10
[2] http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.10

Let's see a couple of declaration their respective `ConstraintValidator` resolution. Assuming the following definitions:

```
[...]
@Constraint(validatedBy={
    SizeValidatorForCollection.class,
    SizeValidatorForSet.class,
    SizeValidatorForSerializable.class })
public @interface Size { ...}

public class SizeValidatorForCollection implements ConstraintValidator<Size, Collection> { ... }
public class SizeValidatorForSet implements ConstraintValidator<Size, Set> { ... }
public class SizeValidatorForSerializable implements ConstraintValidator<Size, Serializable> { ... }

public interface SerializableCollection extends Serializable,  Collection {}
```

The following resolutions occur.

Table 3.1. Resolution of ConstraintValidator for various constraints declarations

Declaration	Resolution
`@Size Collection getAddresses() { ... }`	`SizeValidatorForCollection`: direct match
`@Size Collection<?> getAddresses() { ... }`	`SizeValidatorForCollection`: `Collection` is a direct supertype of `Collection<?>`
`@Size Collection<Address> getAddresses() { ... }`	`SizeValidatorForCollection`: `Collection` is a direct supertype of `Collection<Address>`
`@Size Set<Address> getAddresses() { ... }`	`SizeValidatorForSet`: direct supertype of `Set<Address>`
`@Size SortedSet<Address> getAddresses() { ... }`	`SizeValidatorForSet`: `Set` is the closest supertype of `SortedSet<Address>`
`@Size SerializableCollection getAddresses() { ... }`	`AmbiguousConstraintUsageExceptionSerializableCollection` is a subtype of both `Collection` and `Serializable` and neither `Collection` nor `Serializable` are subtypes of each other.
`@Size String getName() { ... }`	`UnexpectedTypeForConstraintException` none of the `ConstraintValidator` types are supertypes of `String`.

3.6. Examples

The first example demonstrates how beans, fields and getters are annotated to express some constraints.

Example 3.9. Place constraint declarations on the element to validate

```
@ZipCodeCityCoherenceChecker
```

```
public class Address {
    @NotNull @Length(max=30)
    private String addressline1;

    @Length(max=30)
    private String addressline2;

    private String zipCode;

    private String city;

    public String getAddressline1() {
        return addressline1;
    }

    public void setAddressline1(String addressline1) {
        this.addressline1 = addressline1;
    }

    public String getAddressline2() {
        return addressline2;
    }

    public void setAddressline2(String addressline2) {
        this.addressline2 = addressline2;
    }

    public String getZipCode() {
        return zipCode;
    }

    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }

    @Length(max=30) @NotNull
    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }
}
```

During the validation routine execution on an `Address` object,

- `addressline1` field value is passed to the `@NotNull` as well as `Length` constraint validation implementation.
- `addressline2` field value is passed to the `@Length` constraint validation implementations.
- `getCity` value is passed to the `Length` and `@NotNull` constraint validation implementations
- `@ZipCodeCohereceChecker` is a constraint whose validation implementation's `isValid` method receives the `Address` object

The second example demonstrates object graph validation

Example 3.10. Define object graph validation

```
public class Country {
    @NotNull
    private String name;
    @Length(max=2) private String ISO2Code;
    @Length(max=3) private String ISO3Code;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getISO2Code() {
        return ISO2Code;
    }

    public void setISO2Code(String ISO2Code) {
        this.ISO2Code = ISO2Code;
    }

    public String getISO3Code() {
        return ISO3Code;
    }

    public void setISO3Code(String ISO3Code) {
        this.ISO3Code = ISO3Code;
    }
}

public class Address {
    @NotNull @Length(max=30)
    private String addressline1;
    @Length(max=30)
    private String addressline2;
    @Length(max=11)
    private String zipCode;
    @NotNull @Valid
    private Country country;

    private String city;

    public String getAddressline1() {
        return addressline1;
    }

    public void setAddressline1(String addressline1) {
        this.addressline1 = addressline1;
    }

    public String getAddressline2() {
        return addressline2;
    }

    public void setAddressline2(String addressline2) {
        this.addressline2 = addressline2;
    }

    public String getZipCode() {
        return zipCode;
    }

    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }
```

```
    @Length(max=30) @NotNull
    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public Country getCountry() {
        return country;
    }

    public void setCountry(Country country) {
        this.country = country;
    }
}
```

During the validation routine execution on an `Address` object, constraints on `addressLine1`, `addressLine2`, `zipCode` and `country` are processed as well as the validation of the `Country` object itself, more specifically `country.name` is checked for `@NotNull`, `ISO2Code` and `ISO3Code` are checked for `@Length`.

Assuming that `@NotEmpty` is defined as such

```
@Documented
@NotNull
@Size(min=1)
@Constraint(validatedBy = NotEmptyConstraintValidator.class)
@Target({ METHOD, FIELD })
@Retention(RUNTIME)
public @interface NotEmpty {
    ...
}
```

The third example demonstrates superclass, inheritance and composite constraints.

Example 3.11. Use inheritance, constraints on superclasses and composite constraints

```
public interface Person {
    @NotEmpty
    String getFirstName();

    String getMiddleName();

    @NotEmpty
    String getLastName();
}

public class Customer implements Person {
    private String firstName;
    private String middleName;
    private String lastName;
    @NotNull
    private String customerId;
    @Password(robustness=5)
    private String password;

    public String getFirstName() {
        return firstName;
```

```
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getMiddleName() {
        return middleName;
    }

    public void setMiddleName(String middleName) {
        this.middleName = middleName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getCustomerId() {
        return customerId;
    }

    public void setCustomerId(String customerId) {
        this.customerId = customerId;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}

public class PreferredGuest extends Customer {
    @CreditCard
    private String guestCreditCardNumber;

    public String getGuestCreditCardNumber() {
        return guestCreditCardNumber;
    }

    public void setGuestCreditCardNumber(String guestCreditCardNumber) {
        this.guestCreditCardNumber = guestCreditCardNumber;
    }
}
```

When validating a `PreferredGuest` the following constraints are processed:

- `@NotEmpty`, `@NotNull` and `@Size(min=1)` on `firstName`
- `@NotEmpty`, `@NotNull` and `@Size(min=1)` on `lastName`
- `@NotNull` on `customerId`, `@Password` on `password`
- `@CreditCard` on `guestCreditCardNumber`

The fourth example demonstrates the influence of group sequence.

Example 3.12. Use groups and group sequence to define constraint ordering

```
@GroupSequence(sequence={First.class, Second.class, Last.class})
public interface Complete {}

public class Book {
    @NotEmpty(groups=First.class)
    private String title;

    @Length(max=30, groups=Second.class)
    private String subtitle;

    @Valid
    @NotNull(groups=First.class)
    private Author author;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getSubtitle() {
        return subtitle;
    }

    public void setSubtitle(String subtitle) {
        this.subtitle = subtitle;
    }

    public Author getAuthor() {
        return author;
    }

    public void setAuthor(Author author) {
        this.author = author;
    }
}

public class Author {
    @NotEmpty(groups=Last.class)
    private String firstName;

    @NotEmpty(groups=First.class)
    private String lastName;

    @Length(max=30, groups=Last.class)
    private String company;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }
```

```
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getCompany() {
        return company;
    }

    public void setCompany(String company) {
        this.company = company;
    }
}
```

Assuming the validation of the `Complete` group on the following book instance:

```
Author author = new Author();
author.setLastName( "Baudelaire" );
author.setFirstName( "" );
Book book = new Book();
book.setAuthor( author );
```

the validation routine will return the following failure:

- `@NotNull` failure (from `@NotEmpty`) on the `title` field

As both `title` and `author.lastname` are checked as part of the `First` group. If the instance is updated:

```
book.setTitle( "Les fleurs du mal" );
author.setCompany("Some random publisher with a very very very long name");
```

the validation routine will return the following failures:

- `author.firstName` fails to pass the `@Size(min=1)` (from `NotEmpty`) constraint
- `author.company` fails to pass the `Length` constraint

As the `First` and `Second` groups pass without failure, the `Last` group is going through validation.

4

Validation APIs

The default package for the Bean Validation APIs is `javax.validation`

4.1. Validator API

The main Bean Validation API is the `javax.validation.Validator` interface.

A `Validator` instance is able to validate instances of beans and their associated objects if any. It is recommended to leave the caching of `Validator` instances to the `ValidatorFactory`. `Validator` implementations are thread-safe.

```
/**
 * Validate bean instances
 * Implementations of this interface must be thread-safe
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 * @todo Should Serializable be part of the definition?
 */
public interface Validator {
    /**
     * validate all constraints on object
     *
     * @param object object to validate
     * @param groups groups targeted for validation
     *               (default to {@link javax.validation.groups.Default})
     *
     * @return constraint violations or an empty Set if none
     *
     * @throws IllegalArgumentException e if object is null
     */
    <T> Set<ConstraintViolation<T>> validate(T object, Class<?>... groups);

    /**
     * validate all constraints on &lt;code&gt;propertyName&lt;/code&gt; property of object
     *
     * @param object object to validate
     * @param propertyName property to validate (ie field and getter constraints)
     * @param groups groups targeted for validation
     *               (default to {@link javax.validation.groups.Default})
     *
     * @return constraint violations or an empty Set if none
     *
     * @throws IllegalArgumentException e if object is null or if propertyName is not present
     */
    <T> Set<ConstraintViolation<T>> validateProperty(T object,
                                                     String propertyName,
                                                     Class<?>... groups);

    /**
     * validate all constraints on <code>propertyName</code> property
     * if the property value is <code>value</code>
```

```
     * <p/>
     * TODO express limitations of ConstraintViolation in this case
     *
     * @param propertyName property to validate
     * @param value property value to validate
     * @param groups groups targeted for validation
     *               (default to {@link javax.validation.groups.Default})
     *
     * @return constraint violations or an empty Set if none
     * @throws IllegalArgumentException e if propertyName is not present
     */
    <T> Set<ConstraintViolation<T>> validateValue(Class<T> beanType,
                                                  String propertyName,
                                                  Object value,
                                                  Class<?>... groups);

    /**
     * Return the descriptor object describing bean constraints
     * The returned object (and associated objects including ConstraintDescriptors)
     * are immutable.
     *
     * @param clazz class type evaluated
     */
    BeanDescriptor getConstraintsForClass(Class<?> clazz);
}
```

`getConstraintsForClass` is described in Constraint metadata request API.

4.1.1. Validation methods

`<T> Set<ConstraintViolation<T>> validate(T object, Class<?>... groups)` is used to validate a given object. This method implements the logic described in Section 3.5. A `Set` containing all `ConstraintViolation` objects representing the failing constraints is returned, an empty `Set` is returned otherwise.

`<T> Set<ConstraintViolation<T>> validateProperty(T object, String propertyName, Class<?>... groups)` validates a given field or property of an object. The property name is the JavaBeans property name (as defined by the JavaBeans `Introspector` class). This method implements the logic described in Section 3.5 and applies it only to the given property. `@Valid` is not honored by this method. This method is useful for partial object validation.

`<T> Set<ConstraintViolation<T>> validateValue(Class<T> beanType, String propertyName, Object value, Class<?>... groups)` validates the property referenced by *`propertyName`* present on `beanType` or any of its superclasses, if the property value were *`value`*. This method implements the logic described in Section 3.5 and apply it only to the given property and the given value. `@Valid` is not honored by this method. This method is useful for ahead of time validation before the JavaBean is modified.

4.1.1.1. Examples

All the examples will be based on the following class definition, constraint declarations and address instance.

```
public class Address {
    @NotNull @Length(max=30)
    private String addressline1;

    @Length(max=30)
    private String addressline2;
```

```
    private String zipCode;

    private String city;

    public String getAddressline1() {
        return addressline1;
    }

    public void setAddressline1(String addressline1) {
        this.addressline1 = addressline1;
    }

    public String getAddressline2() {
        return addressline2;
    }

    public void setAddressline2(String addressline2) {
        this.addressline2 = addressline2;
    }

    public String getZipCode() {
        return zipCode;
    }

    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }

    @Length(max=30) @NotNull
    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }
}

Address address = new Address();
address.setAddressline1( null );
address.setAddressline2( null );
address.setCity("Llanfairpwllgwyngyllgogerychwyrndrobwyll-llantysiliogogogoch");
//town in North Wales
```

The following code will return two `ConstraintViolation` objects. One for `addressline1` violating `@NotNull` and one for `city` violating `@Length`.

```
validator.validate(address).size() == 2
```

The following code will return one `ConstraintViolation` since `city` violates `@Length` and only `city` is validated.

```
validator.validateProperty(address, "city").size() == 1
```

The following code will return no `ConstraintViolation` object because the value "Paris" for `city` would not raise any constraint failure.

```
validator.validateValue("city", "Paris").size() == 0
```

4.1.2. groups

Groups allow you to restrict the set of constraints applied during validation. Groups targeted are passed as parameters to the `validate`, `validateProperty` and `validateValue` methods. All constraints belonging to the targeted group are applied during the Section 3.5. If no group is passed, the `Default` group is assumed. Section 2.1.1.2 describes how to define groups on constraints.

4.1.2.1. Examples

```
/** Validates a minimal set of constraints */
public interface Minimal {}

public class Address {

    @NotEmpty(groups = Minimal.class)
    @Length(max=50)
    private String street1;

    @NotEmpty
    private String city;

    @NotEmpty(groups = {Minimal.class, Default.class})
    private String zipCode;
    ...
}
```

In the previous example, `@NotEmpty` (and it's composing constraints) on `street1` applies to the group `Minimal`, `@Length` on `street1` applies to the group `Default` and `@NotEmpty` (and it's composing constraints) on `zipCode` applies to the groups `Default` and `Minimal`.

```
validator.validate(address);
```

validates the group `Default` (implicitly) and applies `@Length` on `street1`, `@NotEmpty` (and its composing constraints) on `city`, `@NotEmpty` (and its composing constraints) on `zipCode`. Particularly, `@NotEmpty` (and its composing constraints) on `street1` are not applied.

```
validator.validate(address, Minimal.class);
```

applies `@NotEmpty` (and its composing constraints) on `street1` and `@NotEmpty` (and its composing constraints) on `zipCode` because they belong to the `Minimal` group.

```
validator.validate(address, Minimal.class, Default.class);
```

validates both `Default` and `Minimal` groups. The routine applies `@NotEmpty` (and its composing constraints) and `@Length` on `street1`, `@NotEmpty` (and its composing constraints) on `city`, `@NotEmpty` (and its composing constraints) on `zipCode`. Note that if `zipCode` is empty, only one `ConstraintViolation` object will represent the failure and the not empty validation will only be executed once.

Let's look at a more complex example involving group sequence.

```
public class Address {
    @NotEmpty(groups = Minimal.class)
    @Length(max=50, groups=FirstStep.class)
    private String street1;

    @NotEmpty(groups="SecondStep.class)
    private String city;
```

```
    @NotEmpty(groups = {Minimal.class, SecondStep.class})
    private String zipCode;
    ...

    public interface FirstStep {}

    public interface SecondStep {}

    @GroupSequence(sequence={Firststep.class, SecondStep.class})
    public interface Total {}
}
```

When running:

```
validator.validate(address, Minimal.class, Total.class);
```

the validation process will process `@NotEmpty` (and it's composing constraints) and `@Length` from `street1` and `@NotEmpty` (and it's composing constraints) from `zipCode`. If `@Length` from `street1` does not generate a failure, then `@NotEmpty` (and it's composing constraints) from `city` will be processed as part of `SecondStep`. Note that `@NotEmpty` (and it's composing constraints) from `zipCode` are not reprocessed as they have already been processed before.

When running:

```
validator.validate(address, Total.class, SecondStep.class);
```

`@NotEmpty` (and it's composing constraints) from `city` and `@NotEmpty` (and it's composing constraints) from `zipCode` will be processed even if `@Length` from `street1` fails: while `SecondStep` is in the `Total` group sequence and hence should not be triggered if `FirstStep` has a failure, it also has been requested outside the sequence (in this case explicitly).

4.2. ConstraintViolation

`ConstraintViolation` is the class describing a single constraint failure. A set of `ConstraintViolation` is returned for an object validation.

```
/**
 * Describe a constraint violation. This object describe the error context as
 * well as the message describing the violation.
 *
 * @author Emmanuel Bernard
 * @todo the rational behind rootBean and propertyPath is to keep the context
 *       available to the user
 */
public interface ConstraintViolation<T> {

    /**
     * @return The interpolated error message for this constraint violation.
     */
    String getInterpolatedMessage();

    /**
     * @return The non-interpolated error message for this constraint violation.
     */
    String getRawMessage();
```

```
    /**
     * @return The root bean being validated.
     */
    T getRootBean();

    /**
     * If a bean constraint, the bean instance the constraint is applied on
     * If a property constraint, the bean instance hosting the property the
     * constraint is applied on
     *
     * @return the leaf bean the constraint is applied on or null if Validator#validateValue
     * is used
     */
    Object getLeafBean();

    /**
     * @return the property path to the value from <code>rootBean</code>
     *         <code>null</code> if the value is the <code>rootBean<code> itself.
     */
    String getPropertyPath();

    /**
     * @return the value failing to pass the constraint.
     */
    Object getInvalidValue();

    /**
     * @return the list of groups that the triggered constraint applies on and which also are
     *         within the list of groups requested for validation.
     *
     * TODO: considering removal, if you think it's important, speak up
     */
    Set<Class<?>> getGroups();

    /**
     * Constraint metadata reported to fail.
     * The returned instance is immutable.
     *
     * @return constraint metadata
     */
    ConstraintDescriptor getConstraintDescriptor();
}
```

The `getInterpolatedMessage` method returns the expanded (localized) message for the failing constraint (see Section 4.3 for more information on message expansion). This can be used by clients to expose user friendly messages.

The `getRawMessage` method returns the non-interpolated error message (usually the `message` attribute on the constraint declaration). Frameworks can use this as an error code key.

The `getRootBean` method returns the root object being validated that led to the failing constraint (i.e. the object the client code passes to the `Validator.validate` method.

The `getInvalidValue` method returns the value (field, property or validated object) being passed to `isValid`.

The `getPropertyPath` is built according to the following rules:

- When an association is traversed:
    - if the association is not hosted by the root object (ie hosted on an associated object) a dot (`.`) is concaten-

ated to the `propertyPath`

- the name of the association property (field name or Java Bean property name) is concatenated to the `propertyPath`.
- if the association is a `List` or an array, the index value surrounded by square brackets (`[index]`) is concatenated to the `propertyPath` (for example `order.orderLines[1]`)
- if the association is a `Map`, for a given map entry, the result of `key.toString()` surrounded by square brackets and quotes (`["key.toString()"]`) is concatenated to the `propertyPath` (for example `item.evaluation["quality"]`)

- For property level constraint (field and getter):
  - if the property level constraint is not hosted by the root object (ie hosted on an associated object) a dot (`.`) is concatenated to the `propertyPath`
  - the name of the property (field name or Java Bean property name) is concatenated to the `propertyPath`
  - the `propertyPath` is considered complete
- For class level constraint:
  - nothing is concatenated to the `propertyPath`, it is considered complete. If the `propertyPath` is empty, "" is returned

Note

the collection notation is following the Unified Expression Language conventions.

Note

From `rootBean` and `propertyPath`, it is possible to rebuild the context of the failure

Assuming the following object definitions and when book is validated:

```
@SecurityChecking
public class Author {
    private String firstName;

    @NotEmpty(message="lastname must not be null")
    private String lastName;

    @Length(max=30)
    private String company;
    ...
}

@PresentInAmazon
public class Book {
    @NotEmpty(groups={FirstLevelCheck.class, Default.class})
    private String title;

    @Valid
```

```
    @NotNull
    private List<Author> authors;

    @Valid
    private Map<String, Review> reviewsPerSource;
    ...
}

public class Review {
    @Min(0) private int rating;
    ...
}
```

`propertyPath` evaluations are described in Table 4.1:

Table 4.1. propertyPath examples

Constraint	propertyPath
`@PresentInAmazon` on `Book`	"" (empty string)
`@NotEmpty` on `Book.title`	title
`@NotNull` on `Book.authors`	authors
`@SecurityChecking` on the fourth author, `Author`	authors[3]
`@Length` on the fourth author, `Author.lastname`	authors[3].lastname
`@NotEmpty` on the first author, `Author.company`	authors[0].company
`@Min` on the review associated to Consumer Report, `Review.rating`	reviewsPerSource["Consumer Report"].rating

`groups` returns the intersection of the groups the triggered constraint applies on and the groups requested for validation.

`getConstraintDescriptor` provides access to the failing constraint metadata (see Section 5.5).

4.2.1. Examples

These examples assume the following definition of `@NotEmpty`.

```
@Documented
@NotNull
@Size(min=1)
@ReportAsViolationFromCompositeConstraint
@Constraint(validatedBy = NotEmptyConstraintValidator.class)
@Target({ METHOD, FIELD })
@Retention(RUNTIME)
public @interface NotEmpty {
    ...
}
```

and the following class definitions

```
public class Author {
    private String firstName;

    @NotEmpty(message="lastname must not be null")
    private String lastName;

    @Length(max=30)
    private String company;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getCompany() {
        return company;
    }

    public void setCompany(String company) {
        this.company = company;
    }
}

public class Book {
    @NotEmpty(groups={FirstLevelCheck.class, Default.class})
    private String title;

    @Valid
    @NotNull
    private Author author;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public Author getAuthor() {
        return author;
    }

    public void setAuthor(Author author) {
        this.author = author;
    }
}

Author author = new Author();
author.setCompany("ACME");
Book book = new Book();
book.setTitle("");
book.setAuthor(author);

Set<ConstraintViolation> constraintViolations = validator.validate(book);
```

`ConstraintViolations` is a set of size 2. One of the entries represents the failure of `@NotEmpty` (or more precisely `@Size(min=1)` a composing constraint of `@NotEmpty`) on the `title` property.

The `ConstraintViolation` object for this failure passes the following assertions:

```
//assuming an english locale, the interpolated message is returned
assert "may not be null or empty".equals( constraintViolation.getInterpolatedMessage() );
assert book == constraintViolation.getRootBean();
assert book == constraintViolation.getLeafBean();
//the offending value
assert book.getTitle().equals( constraintViolation.getInvalidValue() );
//the offending property
assert "title".equals( constraintViolation.getPropertyPath() );
assert constraintViolation.getGroups().length == 1

List expectedGroups = new ArrayList(1);
expectedGroups.add(Default.class);
for ( Class<?> group : constraintViolation.getGroups() ) {
    assert expectedGroups.contains(group);
}
```

The second failure, `@NotEmpty` (or more precisely `@NotNull` a composing constraint of `NotEmpty`) on the author's `lastname`, will produce the `ConstraintViolation` object satisfying the following assertions:

```
assert "lastname must not be null".equals( constraintViolation.getInterpolatedMessage() );
assert book == constraintViolation.getRootBean();
assert author == constraintViolation.getLeafBean();
//the offending value
assert book.getAuthor().getLastName() == constraintViolation.getInvalidValue();
//the offending property
assert "author.lastName".equals( constraintViolation.getPropertyPath() );
assert constraintViolation.getGroups().length == 0
```

4.3. Message interpolation

4.3.1. Default message interpolation

A conforming implementation includes a default message interpolator. This message interpolator shall use the algorithm defined here to interpolate message descriptors into human-readable messages.

Each constraint defines a message descriptor via its `message` property. Every constraint definition shall define a default message descriptor for that constraint. Messages can be overridden at declaration time in constraints by setting the `message` property on the constraint.

The message descriptor is a string literal and may contain one or more message parameters. Message parameters are string literals enclosed in braces.

Example 4.1. Message using parameters

```
Value must be between {min} and {max}
```

4.3.1.1. Algorithm

The default message interpolator uses the following steps:

1. Message parameters are extracted from the message string and used as keys to search the `ResourceBundle` named `ValidationMessages` (often materialized as the property file `/ValidationMessages.properties` and its locale variations) using the defined locale (see below). If a property is found, the message parameter is replaced with the property value in the message string. Step 1 is applied recursively until no replacement is performed (ie. a message parameter value can itself contain a message parameter).
2. Message parameters are extracted from the message string and used as keys to search the Bean Validation provider's built-in `ResourceBundle` using the defined locale (see below). If a property is found, the message parameter is replaced with the property value in the message string. Contrary to step 1, step 2 is not processed recursively.
3. If step 2 triggers a replacement, then step 1 is applied again. Otherwise step 4 is performed.
4. Message parameters are extracted from the message string. Those matching the name of an attribute of the constraint declaration are replaced by the value of that attribute.

The defined locale is as followed:

- if the locale is passed to the interpolator method `interpolate(String, CosntraintDescriptor, Object, Locale)`, this `Locale` instance is used.
- otherwise, the default `Locale` as provided by `Locale.getDefault()` is used.

The proposed algorithm ensures that custom resource bundle always have priority over built-in resource bundle at all level of the recursive resolution. It also ensures that constraint declarations attributes values are not expanded further.

4.3.2. Custom message interpolation

A custom message interpolator may be provided (e.g., to interpolate contextual data, or to adjust the default `Locale` used). A message interpolator implements the `MessageInterpolator` interface.

```
/**
 * Interpolate a given constraint violation message.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface MessageInterpolator {
        /**
         * Interpolate the message from the constraint parameters and the actual validated object.
         * The locale is defaulted according to the <code>MessageInterpolator</code> implementation
         * See the implementation documentation for more detail.
         *
         * @param message The message to interpolate.
         * @param constraintDescriptor The constraint descriptor.
         * @param value The object being validated
         *
         * @return Interpolated error message.
```

```
         */
        String interpolate(String message,
                                          ConstraintDescriptor constraintDescriptor,
                                          Object value);

        /**
         * Interpolate the message from the constraint parameters and the actual validated object.
         * The Locale used is provided as a parameter
         *
         * @param message The message to interpolate.
         * @param constraintDescriptor The constraint descriptor.
         * @param value The object being validated
         * @param locale the locale targeted for the message
         *
         * @return Interpolated error message.
         */
        String interpolate(String message,
                                          ConstraintDescriptor constraintDescriptor,
                                          Object value,
                                          Locale locale);
}
```

message is the message descriptor as seen in `@ConstraintAnnotation.message` or provided to the `ConstraintContext` methods.

constraintDescriptor is the `ConstraintDescriptor` object representing the metadata of the failing constraint (see Constraint metadata request API).

value is the value being validated.

`MessageInterpolator.interpolate(String, ConstraintDescriptor, Object)` is invoked by for each constraint violation report generated. The default `Locale` is implementation specific.

`MessageInterpolator.interpolate(String, ConstraintDescriptor, Object, Locale)` can be invoked by a wrapping `MessageInterpolator` to enforce a specific `Locale` value by bypassing or overriding the default `Locale` strategy.

A message interpolator implementation shall be threadsafe.

The message interpolator is provided to the `ValidatorFactory` at construction time using `Configuration.messageInterpolator(MessageInterpolator)`. This message interpolator is shared by all validators generated by this `ValidatorFactory`.

It is possible to override the `MessageInterpolator` implementation for a given `Validator` instance by invoking `ValidatorFactory.usingContext().messageInterpolator(messageInterpolator).getValidator()`.

It is recommended that `MessageInterpolator` implementations delegate final interpolation to the Bean Validation default `MessageInterpolator` to ensure standard Bean Validation interpolation rules are followed, The default implementation is accessible through `Configuration.getDefaultMessageInterpolator()`.

4.3.3. Examples

These examples describe message interpolation based on the default message interpolator's built-in messages (see Appendix B), and the `ValidationMessages.properties` file shown in table Table 4.2. The current locale is assumed English.

```
//ValidationMessages.properties
myapp.creditcard.error=credit card number not valid
```

Table 4.2. message interpolation

Failing constraint declaration	interpolated message
@NotNull	must not be null
@Max(30)	must be less than or equal to 30
@Size(min=5, max=15, message="Key must have between {min} and {max} characters")	Key must have between 5 and 15 characters
@Digits(integer=9, fraction=2)	numeric value out of bounds (<9 digits>.<2 digits> expected)
@CreditCard(message={myapp.creditcard.error})	credit card number not valid

Here is an approach to specify the `Locale` value to choose on a given `Validator`. `Locale` aware `MessageInterpolator`. See Section 4.4 for more details on the APIs.

Example 4.2. Use MessageInterpolator to use a specific Locale value

```
/**
 * delegates to a MessageInterpolator implementation but enforce a given Locale
 */
public class LocaleSpecificMessageInterpolator implements MessageInterpolator {
    private final MessageInterpolator defaultInterpolator;
    private final Locale defaultLocale;

    public LocaleSpecificMessageInterpolator(MessageInterpolator interpolator, Locale locale) {
        this.defaultLocale = locale;
        this.defaultInterpolator = interpolator;
    }

    /**
     * enforece the locale passed to the interpolator
     */
    public String interpolate(String message,
                              ConstraintDescriptor constraintDescriptor,
                              Object value) {
        return defaultInterpolator.interpolate(message, constraintDescriptor,
                                               value, this.defaultLocale);
    }

    // no real use, implemented for completeness
    public String interpolate(String message,
                              ConstraintDescriptor constraintDescriptor,
                              Object value,
                              Locale locale) {
        return defaultInterpolator.interpolate(message, constraintDescriptor, value, locale);
    }
}


Locale locale = getMyCurrentLocale();
```

```
MessageInterpolator interpolator = new LocaleSpecificMessageInterpolator(
                                        validatorFactory.getMessageInterpolator(),
                                        locale);

Validator validator = validatorFactory.usingContext()
                                       .messageInterpolator(interpolator)
                                       .getValidator();
```

Most of the time, however, the relevant `Locale` will be provided by your application framework transparently. This framework will implement its own version of `MessageInterpolator` and pass it during the `ValidatorFactory` configuration. The application will not have to set the `Locale` itself. This example shows how a container framework would implement `MessageInterpolator` to provide a user specific default locale.

Example 4.3. Contextual container possible MessageInterpolator implementation

```
public class ContextualMessageInterpolator {
    private final MessageInterpolator delegate;

    public ContextualMessageInterpolator(MessageInterpolator delegate) {
        this.delegate = delegate;
    }

    public String interpolate(String message, ConstraintDescriptor constraintDescriptor,
                              Object value) {
        Locale locale = Container.getManager().getUserLocale();
        return this.delegate.interpolate(
                        message, constraintDescriptor, value, locale );
    }

    public String interpolate(String message, ConstraintDescriptor constraintDescriptor,
                              Object value, Locale locale) {
        return this.delegate.interpolate(message, constraintDescriptor, value, locale);
    }
}


//Build the ValidatorFactory
Configuration<?> configuration = Validation.byDefaultProvider().configure();
ValidatorFactory factory = configuration
    .messageInterpolator( new ContextualMessageInterpolator( configuration.getDefaultMessageInterpolator(
    .buildValidatorFactory();

//The container uses the factory to validate constraints using the specific MessageInterpolator
Validator validator = factory.getValidator();
```

4.4. Bootstrapping

The bootstrapping API aims at providing a `ValidatorFactory` object which is used to create `Validator` instances. The bootstrap process is decoupled from the provider implementation initialization: a bootstrap implementation must be able to bootstrap any Bean Validation provider implementation. The bootstrap sequence has been designed to achieve several goals:

- plug multiple implementations

- choose a specific implementation
- extensibility: an application using a specific provider implementation can use specific configurations
- share and reuse of metadata across validators
- leave as much freedom as possible to implementations
- provide integration mechanisms to Java EE 6 and other containers
- type safety

The main artifacts involved in the bootstrap process are:

- `Validation`: API entry point. Lets you optionally define the Bean Validation provider targeted as well as a provider resolution strategy. Validation generates `Configuration` objects and can bootstrap any provider implementation.
- `ValidationProvider`: contract between the bootstrap procedure and a Bean Validation provider implementation.
- `ValidationProviderResolver`: returns a list of all Bean Validation providers available in the execution context (generally the classpath).
- `Configuration`: collects the configuration details that will be used to build `ValidatorFactory`. A specific sub interface of `Configuration` must be provided by Bean Validation providers as a unique identifier. This sub interface typically hosts provider specific configurations.
- `ValidatorFactory`: result of the bootstrap process. Build `Validator` instances from a given Bean Validation provider.

Let's first see the API in action through some examples before diving into the concrete definitions.

4.4.1. Examples

The most simple approach is to initialize the default Bean Validation provider or the one defined in the XML configuration file. The `ValidatorFactory` is then ready to provide `Validator` instances.

Example 4.4. Simple Bean Validation bootstrap sequence

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();

//cache the factory somewhere
Validator validator = factory.getValidator();
```

The `ValidatorFactory` object is thread-safe. Building `Validator` instances is typically a cheap operation. Building a `ValidatorFactory` is typically more expensive. Make sure to check your Bean Validation implementation documentation for more accurate details.

The second example shows how a container can customize some Bean Validator resource handling to match its own behavior.

Example 4.5. Customize message resolution, traversable resolver and constraint Validator factory implementation

```
//some customization from a container
ValidatorFactory factory = Validation
       .byDefaultProvider().configure()
          .messageInterpolator( new ContainerMessageInterpolator() )
          .constraintValidatorFactory( new ontainerComponentConstraintValidatorFactory() )
          .traversableResolver( new JPAAwareTraversableResolver() )
          .buildValidatorFactory();

//cache the factory somewhere
Validator validator = factory.getValidator();
```

The third example shows how to bootstrap Bean Validation in an environment not following the traditional Java classloader strategies (such as tools or alternative service containers like OSGi). They can provider some alternative provider resolution strategy to discover Bean Validation providers.

Example 4.6. Customize the Bean Validation provider resolution mechanism

```
//osgi environment
ValidatorFactory factory = Validation
       .byDefaultProvider()
          .providerResolver( new OSGiServiceDiscoverer() )
          .configure()
             .buildValidatorFactory();

//cache the factory somewhere
Validator validator = factory.getValidator();
```

The next example shows how a client can choose a specific Bean Validation provider and configure provider specific properties programmatically in a type-safe way.

Example 4.7. Use a specific provider and add specific configuration

```
ValidatorFactory factory = Validation
       .byProvider( ACMEConfiguration.class )  //chose a specific provider
       .configure()
          .messageInterpolator( new ContainerMessageInterpolator() ) //default configuration option
          .addConstaint(Address.class, customConstraintDescriptor) //ACME specific method
          .buildValidatorFactory();

//same initialization decomposing calls
ACMEConfiguration acmeConfiguration = Validation
       .byProvider( ACMEConfiguration.class )
       .configure();

ValidatorFactory factory = acmeConfiguration
          .messageInterpolator( new ContainerMessageInterpolator() ) //default configuration option
          .addConstaint(Address.class, customConstraintDescriptor) //ACME specific method
```

```
          .buildValidatorFactory();


/**
 * ACME specific validator configuration and configuration options
 */
public interface ACMEConfiguration
                   extends Configuration<ACMEConfiguration> {
    /**
     * Programmatically add constraints. Specific to the ACME provider.
     */
    ACMEConfiguration addConstraint(Class<?> entity,
                                    ACMEConstraintDescriptor constraintDescriptor);
}
```

The last example shows how a `Validator` can use a specific `MessageInterpolator` implementation

Example 4.8. Use a specific MessageInterpolator instance for a given Validator

```
ValidatorFactory factory = ...;
MessageInterpolator customInterpolator = new LocaleSpecificMessageInterpolator(
    locale,
    factory.getMessageInterpolator()
);

Validator localizedValidator =
    factory.usingContext()
                   .messageInterpolator(customInterpolator)
                   .getValidator();
```

In the same flow, a custom `TraversableResolver` can be passed.

We will now explore the various interfaces, their constraints and usage. We will go from the `ValidatorFactory` to the `Validation` class walking up the bootstrap chain.

4.4.2. ValidatorFactory

`ValidatorFactory` objects build and provide initialized instances of `Validator` to Bean Validation clients. Each `Validator` instance is configured for a given context (message interpolator, traversable resolver). Clients should cache `ValidatorFactory` objects and reuse them for optimal performances. The API is designed to allow implementors to share constraint metadata in `ValidatorFactory`.

`ValidatorFactory` implementations must be thread-safe. `ValidatorFactory` implementations can cache `Validator` instances if needed.

Example 4.9. ValidatorFactory interface

```
/**
 * Factory returning initialized Validator instances.
 * Implementations are thread-safe
 * This object is typically cached and reused.
 *
 * @author Emmanuel Bernard
```

```
 */
public interface ValidatorFactory {
        /**
         * return an initialized Validator instance using the default factory instances
         * for message interpolator and traversable resolver.
         *
         * Validator instances can be pooled and shared by the implementation
         */
        Validator getValidator();

        /**
         * Define the validator context and return a
         * Validator compliant with this state
         *
         * @return a ValidatorContext
         */
        ValidatorContext usingContext();

        /**
         * Returns the MessageInterpolator instance configured at initialization time
         * for the ValidatorFactory
         * This is the instance used by #getValidator(Class)
         *
         * @return MessageInterpolator instance
         */
        MessageInterpolator getMessageInterpolator();
}
```

A `ValidatorFactory` is provided by a `Configuration`.

`ValidatorContext` returned by `usingContext` can be used to customize the state in which the `Validator` must be initialized. This is used to customize the `MessageInterpolator` or the `TraversableResolver`.

Example 4.10. ValidatorContext interface

```
/**
 * Return a Validator corresponding to the initialized state.
 *
 * @author Emmanuel Bernard
 */
public interface ValidatorContext {
        /**
         * Defines the message interpolator implementation used by the Validator.
         * If unset, the message interpolator of the ValidatorFactory is used.
         *
         * @return self following the chaining method pattern
         */
        ValidatorContext messageInterpolator(MessageInterpolator messageInterpolator);

        /**
         * Defines the traversable resolver implementation used by the Validator.
         * If unset, the traversable resolver of the ValidatorFactory is used.
         *
         * @return self following the chaining method pattern
         */
        ValidatorContext traversableResolver(TraversableResolver traversableResolver);

        /**
         * @return an initialized <code>Validator</code> instance respecting the defined state.
         * Validator instances can be pooled and shared by the implementation.
         */
```

```
        Validator getValidator();
}
```

The `MessageInterpolator` or the `TraversableResolver` passed to the `ValidatorContext` are used instead of the `ValidatorFactory`'s `MessageInterpolator` or `TraversableResolver` instances.

`getMessageInterpolator()` returns the `MessageInterpolator` instance configured during the initialization of the `ValidatorFactory`. It is particularly useful to build a `Validator` specific `MessageInterpolator` wrapping the one from the `ValidatorFactory`.

Example 4.11. Use of ValidatorFactory

```
ValidatorFactory factory = ...
Validator validatorUsingDefaults = factory.getValidator();
Validator validatorUsingCustomTraversable = factory
                      .usingContext()
                      .traversableResolver( new JPATraversableResolver() )
                      .getValidator();
```

See Example 4.2 for an example using `getMessageInterpolator()`.

4.4.3. Configuration

`Configuration` collects configuration informations, determines the correct provider implementation and delegates it the `ValidatorFactory` creation. This class lets you define:

- the message interpolator strategy instance
- the traversable resolver strategy instance
- the constraint Validator factory instance
- the configuration `InputStream`

A `Configuration` does provide a `MessageInterpolator` implementation following the default Bean Validation `MessageInterpolator` rules as defined in Section 4.3.1 by calling `getDefaultMessageInterpolator()`. Such an implementation is useful to let a custom `MessageInterpolator` delegates to the standard `MessageInterpolator` (see Section 4.3.2 and an example making use of `getDefaultMessageInterpolator()` in Example 4.3).

Clients call `Configuration.buildValidatorFactory()` to retrieve the initialized `ValidatorFactory` instance.

Example 4.12. Configuration interface

```
/**
 * Receives configuration information, selects the appropriate
 * Bean Validation provider and build the appropriate
 * ValidatorFactory.
 * <p/>
 * Usage:
```

```
 * <pre>
 * Configuration<?> configuration = //provided by one of the Validation bootstrap methods
 *     ValidatorFactory = configuration
 *         .messageInterpolator( new CustomMessageInterpolator() )
 *         .buildValidatorFactory();
 * </pre>
 * <p/>
 * The ValidationProviderResolver is specified at Configuration time
 * (see {@link javax.validation.spi.ValidationProvider}).
 * If none is explicitely requested, the default ValidationProviderResolver is used.
 * <p/>
 * The provider is selected in the following way:
 * - if a specific Configuration subclass is requested programmatically using
 * Validation.byProvider(Class), find the first provider matching it
 * - if a specific Configuration subclass is defined in META-INF/validation.xml,
 * find the first provider matching it
 * - otherwise, use the first provider returned by the ValidationProviderResolver
 * <p/>
 * Implementations are not meant to be thread-safe
 *
 * @author Emmanuel Bernard
 */
public interface Configuration<T extends Configuration<T>> {
        /**
         * Defines the message interpolator used. Has priority over the configuration
         * based message interpolator.
         *
         * @param interpolator message interpolator implementation.
         *
         * @return <code>this</code> following the chaining method pattern.
         */
        T messageInterpolator(MessageInterpolator interpolator);

        /**
         * Defines the traversable resolver used. Has priority over the configuration
         * based traversable resolver.
         *
         * @param resolver traversable resolver implementation.
         *
         * @return <code>this</code> following the chaining method pattern.
         */
        T traversableResolver(TraversableResolver resolver);

        /**
         * Defines the constraint validator factory. Has priority over the configuration
         * based constraint factory.
         *
         * @param constraintValidatorFactory constraint factory inmplementation.
         *
         * @return <code>this</code> following the chaining method pattern.
         */
        T constraintValidatorFactory(ConstraintValidatorFactory constraintValidatorFactory);

        /**
         * Configure the ValidatorFactory based on <code>stream</code>
         * If not specified, META-INF/validation.xml is used
         * <p/>
         * The stream should be closed by the client API after the
         * ValidatorFactory has been returned
         *
         * @param stream configuration stream.
         *
         * @return <code>this</code> following the chaining method pattern.
         */
        T customConfiguration(InputStream stream);
```

```
        /**
         * Return an implementation of the MessageInterpolator interface following the
         * default MessageInterpolator defined in the specification:
         *  - use the ValidationMessages resource bundle to load keys
         *  - use Locale.getDefault()
         *
         * @return default MessageInterpolator implementation compliant with the specification
         */
        MessageInterpolator getDefaultMessageInterpolator();

        /**
         * Build a ValidatorFactory implementation.
         *
         * @return ValidatorFactory
         */
        ValidatorFactory buildValidatorFactory();
}
```

A Bean Validation provider must define a sub interface of `Configuration` uniquely identifying the provider. The `isSuitable()` method of its `ValidationProvider` implementation must return true when this sub interface type is passed as a parameter, false otherwise. The `Configuration` sub interface typically hosts provider specific configuration methods.

To facilitate the use of provider specific configuration methods, `Configuration` uses generics: `Configuration<T extends Configuration<T>>` ; the generic return type `T` is returned by chaining methods. The provider specific sub interface must resolve the generic T as itself as shown in the following example.

Example 4.13. Example of provider specific Configuration sub interface

```
/**
 * Unique identifier of the ACME provider
 * also host some provider specific configuration methods
 */
public interface ACMEConfiguration
    extends Configuration<ACMEonfiguration> {

    /**
     * Enables contraints implementation dynamic reloading when using ACME
     * default to false
     */
    Configuration enableDynamicReloading(boolean);

}
```

When `Configuration.buildValidatorFactory()` is called, the initialized `ValidatorFactory` is returned. More specifically, the requested Bean Validation provider is determined and the result of `validationProvider.buildValidatorFactory(ConfigurationState)` is returned. `ConfigurationState` gives access to the configuration artifacts passed to `Configuration`. A typical implementation of `Configuration` also implements `ConfigurationState`, hence `this` can be passed to `buildValidatorFactory(ConfigurationState)`.

Example 4.14. ConfigurationState interface

```
/**
 * Contract between a <code>Configuration</code> and a
```

```
 * </code>ValidatorProvider</code> to create a <code>ValidatorFactory</code>.
 * The configuration artifacts provided to the
 * <code>Configuration</code> are passed along.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface ConfigurationState {
        /**
         * Message interpolator as defined by the client programmatically
         * or null if undefined.
         *
         * @return message provider instance or null if not defined
         */
        MessageInterpolator getMessageInterpolator();

        /**
         * Returns the configuration stream defined by the client programmatically
         * or null if undefined.
         *
         * @return the configuration input stream or null
         */
        InputStream getConfigurationStream();

        /**
         * Defines the constraint validator implementation factory as defined by
         * the client programmatically or null if undefined
         *
         * @return factory instance or null if not defined
         */
        ConstraintValidatorFactory getConstraintValidatorFactory();

        /**
         * Traversable resolver as defined by the client programmatically
         * or null if undefined.
         *
         * @return traversable provider instance or null if not defined
         */
        TraversableResolver getTraversableResolver();
}
```

The requested provider implementation is resolved according to the following rules in the following order:

- Use the provider implementation requested if `Configuration` has been created from `Validation.byProvider(Class)`.
- Use the provider implementation associated with the `Configuration` implementation described in the XML configuration (under `validation.provider`) if defined: the value of this element is the fully qualified class name of the `Configuration` sub interface uniquely identifying the provider.
- Use the first provider implementation returned by `validationProviderResolver.getValidationProviders()`.

The `ValidationProviderResolver` is specified when `Configuration` instances are created (see `ValidationProvider`). If no `ValidationProviderResolver` instance has been specified, the default `ValidationProviderResolver` is used.

`Configuration` instances are provided to the Bean Validation client through the `Validation` methods. `Configura-`

`tion` instances are created by `ValidationProvider`.

Warning

Should we add a ignore XML method? to bypass the XMl file configuration?

Here is an example of `Configuration` use.

Example 4.15. Use Configuration

```
Configuration configuration = ...
ValidatorFactory factory = configuration
              .messageInterpolator( new WBMessageInterpolator() )
              .traversableResolver( new JPAAwareTraversableResolver() )
              .buildValidatorFactory();
```

4.4.4. ValidationProvider and ValidationProviderResolver

`ValidationProvider` is the contract between the bootstrap process and a specific Bean Validation provider. `ValidationProviderResolver` implements the discovery mechanism for Bean Validation provider implementation. Any Bean Validation client can implement such a discovery mechanism but it is typically implemented by containers having specific classloader structures and restrictions.

4.4.4.1. ValidationProviderResolver

`ValidationProviderResolver` returns the list of Bean Validation providers available at runtime and more specifically a `ValidationProvider` instance for each provider available in the context. This service can be customized by implementing `ValidationProviderResolver`. Implementations must be thread-safe.

Example 4.16. ValidationProviderResolver

```
/**
 * Determine the list of Bean Validation providers available in the runtime environment
 * <p>
 * Bean Validation providers are identified by the presence of
 * META-INF/services/javax.validation.spi.ValidationProvider
 * files following the Service Provider pattern described
 * <a href="http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html#Service%20Provider">here</a>
 * <p/>
 * Each META-INF/services/javax.validation.spi.ValidationProvider file contains the list of
 * ValidationProvider implementations each of them representing a provider.
 *
 * Implementations must be thread-safe.
 *
 * @author Emmanuel Bernard
 */
public interface ValidationProviderResolver {
    /**
     * Returns a list of ValidationProviders available in the runtime environment.
     *
     * @return list of validation providers.
     */
    List<ValidationProvider> getValidationProviders();
}
```

By default, providers are resolved using the Service Provider pattern described in http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html#Service%20Provider. Each Bean Validation provider should supply a service provider configuration file by creating a text file `javax.validation.spi.ValidationProvider` and placing it in the `META-INF/services` directory of one of its jar files. The content of the file should contain the name of the provider implementation class of the `javax.validation.spi.ValidationProvider` interface.

Persistence provider jars may be installed or made available in the same ways as other service providers, e.g. as extensions or added to the application classpath according to the guidelines in the JAR file specification.

The default `ValidationProviderResolver` implementation will locate all the Bean Validation providers by their provider configuration files visible in the classpath. The default `ValidationProviderResolver` implementation is recommended and custom `ValidationProviderResolver` implementations should be rarely used. A typical use of a custom resolution is resolving providers in a classloader constrained container like OSGi or in a tool environment (IDE).

The default implementation of `ValidationProviderResolver` must be available at `javax.validation.bootstrap.DefaultValidationProviderResolver`. It must contain a public no-arg constructor and must not have any other non private attribute or method besides the method described by `ValidationProviderResolver`.

4.4.4.2. ValidationProvider

`ValidationProvider` represents the SPI (Service Provider Interface) defining the contract between the provider discovery and initialization mechanism, and the provider. A `ValidationProvider` does:

- Determine if a provider matches a given `Configuration` sub interface. One `Configuration` sub interface specifically represent one Bean Validation provider.
- Provide a provider specific `Configuration` implementation. This `Configuration` will specifically build `ValidatorFactory` instances of the provider it comes from.
- Build a `ValidatorFactory` object from the configuration provided by `ConfigurationState`.

Example 4.17. ValidationProvider

```
/**
 * Contract between the validation bootstrap mechanism and the provider engine.
 * <p/>
 * Implementations must have a public no-arg constructor. The construction of a provider
 * should be as "lightweight" as possible.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface ValidationProvider {
        /**
         * This sub interface uniquely identify a provider.
         *
         * @param configurationClass targeted configuration class.
         *
         * @return <code>true</code> if <code>configurationClass</code> is the Bean Validation Provider
         *         sub-interface for Configuration
         */
```

```
        boolean isSuitable(Class<? extends Configuration<?>> configurationClass);

        /**
         * Returns a Configuration instance implementing the
         * <code>configurationClass</code> interface.
         * The Configuration instance uses the current provider (<code>this</code>)
         * to build the ValidatorFactory instance.
         * <p/>
         * This method can only be called on providers returning true on
         * <code>#isSuitable(configurationClass)</code>
         *
         * @param configurationClass the Configuration class type
         * @param state bootstrap state
         *
         * @return specific Configuration implementation
         */
        <T extends Configuration<T>> T createSpecializedConfiguration(
                        BootstrapState state,
                        Class<T> configurationClass);

        /**
         * Returns a Configuration instance. This instance is not bound to
         * use the current provider. The choice of provider follows the algorithm described
         * in {@link javax.validation.Configuration}
         * <p/>
         * The ValidationProviderResolver used is provided by <code>state</code>.
         * If null, the default ValidationProviderResolver is used.
         *
         * @param state bootstrap state
         *
         * @return Configuration implementation
         */
        Configuration<?> createGenericConfiguration(BootstrapState state);

        /**
         * Build a ValidatorFactory using the current provider implementation. The
         * ValidatorFactory is assembled and follow the configuration passed
         * using ConfigurationState.
         * <p>
         * The returned ValidatorFactory is properly initialized and ready for use.
         * </p>
         *
         * @param configurationState the configuration descriptor
         *
         * @return the instanciated ValidatorFactory
         */
        ValidatorFactory buildValidatorFactory(ConfigurationState configurationState);
}
```

Example 4.18. BootstrapState interface

```
/**
 * Defines the state used to bootstrap the Configuration
 *
 * @author Emmanuel Bernard
 */
public interface BootstrapState {
        /**
         * returns the user defined ValidationProviderResolver strategy instance or <code>null</code>
         * if undefined.
         *
         * @return ValidationProviderResolver instance or null
```

```
         */
        ValidationProviderResolver getValidationProviderResolver();
}
```

A client can request a specific Bean Validation provider by using `Validation.byProvider(Class<T exptends Configuration<T>>)` or by defining the provider in the XML configuration file. The key uniquely identifying a Bean Validation provider is a provider specific sub interface of `Configuration`. The sub interface does not have to add any new method but is the natural holder of provider specific methods.

Example 4.19. Example of provider specific Configuration sub interface

```
/**
 * Unique identifier of the ACME provider
 * also host some provider specific configuration methods
 */
public interface ACMEConfiguration
    extends Configuration<ACMEConfiguration> {

    /**
     * Enables contraints implementation dynamic reloading when using ACME
     * default to false
     */
    ACMEConfiguration enableDynamicReloading(boolean);

}
```

Note

`Configuration` references itself in the generic definition. Methods of `Configuration` will return the `ACME-Configuration` making the API easy to use even for vendor specific extensions.

The provider discovery mechanism uses the following algorithm:

- Retrieve available providers using `ValidationProviderResolver.getValidationProviders()`.
- The first `ValidationProvider` matching the requested provider is returned. Providers are evaluated in the order they are provided by `ValidationProviderResolver`. A provider is considered matching if `ValidationProvider.isSuitable(Class<T extends Configuration<T>>)` returns true when the requested provider specific `Configuration` sub interface is passed as a parameter.

When the default Bean Validation provider is requested, the first `ValidationProvider` returned by the `ValidationProviderResolver` strategy is returned.

Every Bean Validation provider must provide a `ValidationProvider` implementation containing a public no-arg constructor and add the corresponding `META-INF/services/javax.validation.spi.ValidationProvider` file descriptor in one of its jars.

4.4.5. Validation

The `Validation` class is the entry point used to bootstrap Bean Validation providers. The first entry point, `build-`

`DefaultValidatorFactory()`, returns a `ValidatorFactory`. The first provider returned by the default `ValidationProviderResolver` is used to build the `ValidatorFactory`. `Validation.buildDefaultValidatorFactory()` is equivalent to `Validation.byDefaultProvider().configure().buildValidatorFactory()`.

Warning

Should the resolver strategy be configurable by XML

Example 4.20. Validation methods available

```
/**
 * This class is the entry point for the Bean Validation framework. There are three ways
 * to bootstrap the framework:
 * <ul>
 * <li>
 * The easiest approach is to use the default Bean Validation provider.
 * <pre>
 * ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
 * </pre>
 * In this case {@link  javax.validation.bootstrap.DefaultValidationProviderResolver}
 * will be used to locate available providers.
 *
 * The chosen provider is defined as followed:
 * <ul>
 * <li>if the XML configuration defines a provider, this provider is used</li>
 * <li>if the XML configuration does not define a provider or if no XML configuration
 * is present the first provider returned by the ValidationProviderResolver
 * isntance is used.</li>
 * </ul>
 * </li>
 * <li>
 * The second bootstrap approach allows to choose a custom
 * <code>ValidationProviderResolver</code>. The chosen
 * <code>ValidationProvider</code> is then determined in the same way
 * as in the default bootstrapping case (see above).
 * <pre>
 * Configuration&lt?&gt; configuration = Validation
 *    .byDefaultProvider()
 *    .providerResolver( new MyResolverStrategy() )
 *    .configure();
 * ValidatorFactory factory = configuration.buildValidatorFactory();
 * </pre>
 * </li>
 *
 * <p/>
 * <li>
 * The third approach allows you to specify explicitly and in
 * a type safe fashion the expected provider by
 * using its specific <code>Configuration</code> sub-interface.
 *
 * Optionally you can choose a custom <code>ValidationProviderResolver</code>.
 * <pre>
 * ACMEConfiguration configuration = Validation
 *    .byProvider(ACMEConfiguration.class)
 *    .providerResolver( new MyResolverStrategy() )  // optionally set the provider resolver
 *    .configure();
 * ValidatorFactory factory = configuration.buildValidatorFactory();
 * </pre>
 * </li>
 * </ul>
 * Note:<br/>
 * <ul>
 * <li>
```

```
 * The ValidatorFactory object built by the bootstrap process should be cached
 * and shared amongst Validator consumers.
 * </li>
 * <li>
 * This class is thread-safe.
 * </li>
 * </ul>
 *
 * @author Emmanuel Bernard
 * @author Hardy Feretnschik
 * @see DefaultValidationProviderResolver
 */
public class Validation {

        /**
         * Build and return a ValidatorFactory instanced based on the
         * default Bean Validation provider and following the
         * XML configuration.
         * <p/>
         * The provider list is resolved using the
         * {@link  javax.validation.bootstrap.DefaultValidationProviderResolver}.
         * <p/> The code is semantically equivalent to
         * <code>Validation.byDefaultProvider().configure().buildValidatorFactory()</code>
         *
         * @return <code>ValidatorFactory</code> instance.
         */
        public static ValidatorFactory buildDefaultValidatorFactory() {
                [...]
        }

        /**
         * Build a <code>Configuration</code>. The provider list is resolved
         * using the strategy provided to the bootstrap state.
         * <pre>
         * Configuration&lt?&gt; configuration = Validation
         *    .byDefaultProvider()
         *    .providerResolver( new MyResolverStrategy() )
         *    .configure();
         * ValidatorFactory factory = configuration.buildValidatorFactory();
         * </pre>
         * The actual provider choice is given by the XML configuration. If the XML
         * configuration does not exsist the first available provider will be returned.
         *
         * @return instance building a generic <code>Configuration</code>
         * compliant with the bootstrap state provided.
         */
        public static GenericBootstrap byDefaultProvider() {
                [...]
        }

        /**
         * Build a <code>Configuration</code> for a particular provider implementation.
         * Optionally override the provider resolution strategy used to determine the provider.
         * <p/>
         * Used by applications targeting a specific provider programmatically.
         * <p/>
         * <pre>
         * ACMEConfiguration configuration =
         *     Validation.byProvider(ACMEConfiguration.class)
         *             .providerResolver( new MyResolverStrategy() )
         *             .configure();
         * </pre>,
         * where <code>ACMEConfiguration</code> is the
         * <code>Configuration</code> sub interface uniquely identifying the
         * ACME Bean Validation provider.
         *
```

```
         * @param configurationType the <code>Configuration</code> sub interface
         * uniquely defining the targeted provider.
         *
         * @return instance building a provider specific <code>Configuration</code>
         * sub interface implementation.
         *
         */
        public static <T extends Configuration<T>>
        ProviderSpecificBootstrap<T> byProvider(Class<T> configurationType) {
                [...]
        }

    [...]
}
```

The second entry point lets the client provide a custom `ValidationProviderResolution` instance. This instance is passed to `GenericBootstrap`. `GenericBootstrap` builds a generic `Configuration` using the first `ValidationProvider` returned by `ValidationProviderResolution` and calling `ValidationProvider.createGenericConfiguration(BootstrapState state)`. `BootstrapState` holds the `ValidationProviderResolution` instance passed to `GenericBootstrap` and will be used by the `Configuration` instance when resolving the provider to use.

Example 4.21. GenericBootstrap interface

```
/**
 * Defines the state used to bootstrap Bean Validation and
 * creates a provider agnostic Configuration.
 *
 * @author Emmanuel Bernard
 */
public interface GenericBootstrap {
        /**
         * Defines the provider resolution strategy.
         * This resolver returns the list of providers evaluated
         * to build the Configuration
         * <p/>
         * If no resolver is defined, the default ValidationProviderResolver
         * implementation is used.
         *
         * @return <code>this</code> following the chaining method pattern
         */
        GenericBootstrap providerResolver(ValidationProviderResolver resolver);

        /**
         * Returns a generic Configuration implementation.
         * At this stage the provider used to build the ValidatorFactory is not defined.
         * <p/>
         * The Configuration implementation is provided by the first provider returned
         * by the ValidationProviderResolver strategy.
         *
         * @return a Configuration implementation compliant with the bootstrap state
         */
        Configuration<?> configure();
}
```

The last entry point lets the client define the specific Bean Validation provider requested as well as a custom `ValidationProviderResolver` implementation if needed. The entry point method, `Validation.byProvider(Class<T>`

`configurationClass)`, takes the provider specific `Configuration` sub interface type and returns a `ProviderSpecificBootstrap` object that guarantees to return an instance of the specific `Configuration` sub interface. Thanks to the use of generics, the client API does not have to cast to the `Configuration` sub interface.

A `ProviderSpecificBootstrap` object can optionally receive a `ValidationProviderResolver` instance.

Example 4.22. ProviderSpecificBootstrap interface

```
/**
 * Defines the state used to bootstrap Bean Validation and
 * creates a provider specific Configuration. The specific Configuration
 * sub interface uniquely identifying a provider.
 * <p/>
 * The requested provider is the first provider suitable for T (as defined in
 * {@link javax.validation.spi.ValidationProvider#isSuitable(Class)}). The
 * list of providers evaluated is returned by {@link ValidationProviderResolver}.
 * If no ValidationProviderResolver is defined, the
 * default ValidationProviderResolver strategy is used.
 *
 * @author Emmanuel Bernard
 */
public interface ProviderSpecificBootstrap<T extends Configuration<T>> {

        /**
         * Optionally define the provider resolver implementation used.
         * If not defined, use the default ValidationProviderResolver
         *
         * @param resolver ValidationProviderResolver implementation used
         *
         * @return self
         */
        public ProviderSpecificBootstrap<T> providerResolver(ValidationProviderResolver resolver);

        /**
         * Determine the provider implementation suitable for configurationType and delegate
         * the creation of this specific Configuration subclass to the provider.
         *
         * @return a Configuration sub interface implementation
         */
        public T configure();
}
```

`ProviderSpecificBootstrap.configure()` must return the result of `ValidationProvider.createSpecializedConfiguration(BootstrapState state, Class<T extends Configuration<T>> configurationType)`. The state parameter holds the `ValidationProviderResolver` passed to `ProviderSpecificBootstrap`. The configuration type passed as a parameter is the configuration type passed to `Validation.byProvider(Class)`. The validation provider is selected from the configuration type according to the algorithm described in (Section 4.4.4.2).

The `Validation` implementation provided by the Bean Validation provider must not contain any non private attribute or method aside from the three public static bootstrap methods:

- `public static ValidatorFactory buildDefaultValidatorFactory()`
- `public static GenericBootstrap byDefaultProvider()`

- `public static <T extends Configuration<T>> ProviderSpecificBootstrap<T> byProvider(Class<T> configurationType)`

The bootstrap API is designed to allow complete portability amongst Bean Validation provider implementations. The bootstrap implementation must ensure it can bootstrap third party providers.

4.4.6. Usage

The Bean Validation bootstrap API can be used directly by the application, through the use of a container or by framework in need for validation. In all cases, the following rules apply:

- `ValidatorFactory` is a thread-safe object that should be built once per deployment unit
- `Validator` is thread-safe too and should be considered a lightweight object. `ValidatorFactory` would typically implement appropriate `Validator` instance caching strategies if needed.

Containers such as Java EE, Web Bean, dependency injection frameworks, component frameworks are encouraged to propose access to `ValidatorFactory` and `Validator` objects in a way that respects the following rules. For example, injection of `Validator` should be possible.

5

Constraint metadata request APIs

The Bean Validation specification provides a way to query the constraint repository. This API is expected to be used for tooling support as well as integration with other frameworks, libraries and JSRs. The Bean Validation specification aims to provide both a validation engine and a metadata repository for object constraints. Frameworks (EE or SE) in need for constraint definition, validation and metadata will be able to rely on the Bean Validation specification for these services avoiding any unnecessary duplication work from an application and infrastructure point of view.

5.1. Validator

The main API to access all metadata related to a given object is `Validator` (see Section 4.4 for more information on how to retrieve a `Validator` instance).

A `Validator` instance hosts the method to access to the metadata repository for a given class. It is recommended to leave the caching of `Validator` instances to the `ValidatorFactory`. `Validator` implementations are thread-safe.

```
/**
 * Validate a given object type
 * Implementations of this interface must be thread-safe
 *
 * @author Emmanuel Bernard
 */
public interface Validator {

    [...] //See 4.1

    /**
     * Return the descriptor object describing bean constraints
     * The returned object (and associated objects including ConstraintDescriptors)
     * are immutable.
     *
     * @param clazz class type evaluated
     */
    BeanDescriptor getConstraintsForClass(Class<?> clazz);
}
```

`getConstraintsForClass` returns a `BeanDescriptor` object describing the bean level constraints (see Section 3.1.1) and providing access to the property level constraints metadata.

5.2. ElementDescriptor

`ElementDescriptor` is the root interface describing elements hosting constraints. It is used to describe the list of constraints for a given element (whether it be a field, a method or a class).

```
/**
 * Describes a validated element (class, field or property).
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface ElementDescriptor {

        /**
         * return true if at least one constraint declaration is present on the element.
         */
        boolean hasConstraints();

        /**
         * @return Statically defined returned type.
         *
         * @todo should it be Type or even completly removed
         */
        Class<?> getType();

        /**
         * @return All the constraint descriptors for this element.
         */
        Set<ConstraintDescriptor> getConstraintDescriptors();
}
```

`getType` returns either the object type for a class, or the returned type for a property (TODO problem of generics resolution).

`getConstraintDescriptors` returns the `ConstraintDescriptors` (see Section 5.5), each `ConstraintDescriptor` describing one of the constraints declared on the given element.

`hasConstraints` returns true if the element (class, field or property) holds at lease one constraint declaration.

5.3. BeanDescriptor

The `BeanDescriptor` interface describes a constrained Java Bean. This interface is returned by `Validator.getConstraintsForClass(Class<?>)`.

```
/**
 * Describe a constrained Java Bean and the constraints associated to it.
 *
 * @author Emmanuel Bernard
 */
public interface BeanDescriptor extends ElementDescriptor {
        /**
         * Returns true if the bean involves validation:
         *  - a constraint is hosted on the bean itself
         *  - a constraint is hosted on one of the bean properties
         *  - or a bean property is marked for cascade (@Valid)
         *
         * @return true if the bean nvolves validation
         *
         */
        boolean isBeanConstrained();

        /**
         * Return the property level constraints for a given propertyName
         * or null if either the property does not exist or has no constraint
         * The returned object (and associated objects including ConstraintDescriptors)
```

```
     * are immutable.
         *
         * @param propertyName property evaludated
         */
        PropertyDescriptor getConstraintsForProperty(String propertyName);

        /**
         * return the property names having at least a constraint defined or marked
         * as cascaded (@Valid)
         */
        Set<String> getConstrainedProperties();
}
```

`isBeanConstrained` returns true if the given class (and superclasses and interfaces) host at least one validation declaration (either constraint or `@Valid` annotation). If the method returns false, the Bean Validation engine can safely ignore the bean as it will not be impacted by validation.

`getConstraintsForProperty` returns a `PropertyDescriptor` object describing the property level constraints (See Section 3.1.2). The property is uniquely identified by its name as per the JavaBeans convention: field level and getter level constraints of the given name are all returned.

`getConstrainedProperties` returns the names of the bean properties having at least one constraint or being cascaded (`@Valid` annotation).

5.4. PropertyDescriptor

The `PropertyDescriptor` interface describes a constrained property of a Java Bean. This interface is returned by `BeanDescriptor.getConstraintsForProperty(String)`. Constraints declared on the attribute and the getter of the same name according to the Java Bean rules are returned by this descriptor.

```
/**
 * Describes a Java Bean property hosting validation constraints.
 *
 * Constraints placed on the attribute and the getter for a given property
 * are all referenced by this object.
 *
 * @author Emmanuel Bernard
 */
public interface PropertyDescriptor extends ElementDescriptor {
    /**
     * Is the property marked by the <code>@Valid</code> annotation.
     * @return true if the annotation is present
     */
    boolean isCascaded();

    /**
     * Name of the property acording to the Java Bean specification.
     * @return property name
     */
    String getPropertyName();
}
```

The `isCascaded` method returns `true` if the property is marked with `@Valid`.

`getPropertyName` returns the property name as described in Section 4.2.

5.5. ConstraintDescriptor

A `ConstraintDescriptor` object describes a given constraint declaration (i.e. a constraint annotation).

```
/**
 * Describes a single constraint and its composing constraints.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface ConstraintDescriptor {
        /**
         * Returns the annotation describing the constraint declaration.
         * If a composing constraint, parameter values are reflecting
         * the overridden parameters from the main constraint
         *
         * @return The annotation for this constraint.
         */
        Annotation getAnnotation();

        /**
         * @return The groups the constraint is applied on.
         */
        Set<Class<?>> getGroups();

        /**
         * @return the constraint validation implementation class
         */
        Class<? extends ConstraintValidator<?,?>>[]
            getConstraintValidatorClasses();

        /**
         * Returns a map containing the annotation parameter names as keys and the
         * annotation parameter values as value.
         * If this constraint is used as part of a composed constraint, parameter
         * values are reflecting the overridden parameters from the main constraint.
         *
         * @return Returns a map containing the annotation paramter names as keys
         *         and the annotation parameter values as value.
         */
        Map<String, Object> getParameters();

        /**
         * Return a set of composing <code>ConstraintDescriptor</code>s where each
         * descriptor describes a composing constraint. <code>ConstraintDescriptor</code>
         * instances of composing constraints reflect overridden parameter values in
         * {@link #getParameters()}  and {@link #getAnnotation()}.
         *
         * @return a set of <code>ConstraintDescriptor<code> objects or an empty set
         *         in case there are no composing constraints.
         */
        Set<ConstraintDescriptor> getComposingConstraints();

        /**
         * @return true if the constraint is annotated with @ReportAsViolationFromCompositeConstraint
         */
        boolean isReportAsViolationFromCompositeConstraint();
}
```

`getAnnotation` returns the annotation instance (or an annotation instance representing the given constraint declaration). If `ConstraintDescriptor` represents a composing annotation (see Section 2.3), the returned annotation must reflect parameter overriding. In other words, the annotation parameter values are the overridden values.

`getParameters` returns a map containing the annotation parameter names as a key, and the annotation parameter values as a value (this API is anticipated to be simpler to use by tools than reflection over the annotation instance). If `ConstraintDescriptor` represents a composing annotation (see Section 2.3), the returned `Map` must reflect parameter overriding.

`getGroups` returns the groups the constraint is supposed to be applied upon.

`getConstraintValidatorClasses` returns the `ConstraintValidator` classes associated with he constraint.

5.6. Example

Assuming the following `@NotEmpty` definition

```
@Documented
@NotNull
@Size(min=1)
@ReportAsViolationFromCompositeConstraint
@Constraint(validatedBy = NotEmptyConstraintValidator.class)
@Target({ METHOD, FIELD })
@Retention(RUNTIME)
public @interface NotEmpty {
    ...
}
```

and the following class definitions

```
public class Author {
    private String firstName;

    @NotEmpty(message="lastname must not be null")
    private String lastName;

    @Length(max=30)
    private String company;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String getCompany() {
        return company;
    }

    public void setCompany(String company) {
        this.company = company;
    }
}

public class Book {
    private String title;
    private String description;
```

```
    @Valid
    @NotNull
    private Author author;

    @NotEmpty(groups={FirstLevelCheck.class, Default.class})
    @Length(max=30)
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public Author getAuthor() {
        return author;
    }

    public void setAuthor(Author author) {
        this.author = author;
    }

    public String getDescription() {
        return description;
    }

    public void setAuthor(String description) {
        this.description = description;
    }
}
```

The following assertions are true.

```
BeanDescriptor bookDescriptor = validator.getConstraintsForClass(Book.class);

assert ! bookDescriptor.hasConstraints();

assert bookDescriptor.isBeanConstrained();

assert bookDescriptor.getConstraintDescriptors().size() == 0 //no bean-level constraint

//more specifically "author" and "title"
assert bookDescriptor.getConstrainedProperties().size() == 2;

//not a property
assert bookDescriptor.getConstraintsForProperty("doesNotExist") == null;

//property with no constraint
assert bookDescriptor.getConstraintsForProperty("description") == null;

ElementDescriptor propertyDescriptor = bookDescriptor.getConstraintsForProperty("title");
assert propertyDescriptor.getConstraintDescriptors().size() == 2
assert "title".equals( propertyDescriptor.getPropertyName() );

//assuming the implementation returns the @NotEmpty constraint first
ConstraintDescriptor constraintDescriptor = propertyDescriptor.getConstraintDescriptors()
                                                              .iterator().next();
assert constraintDescriptor.getAnnotation().getAnnotationType().equals( NotEmpty.class );
assert constraintDescriptor.getGroups().size() == 2; //FirstLevelCheck and Default
assert constraintDescriptor.getComposingConstraints().size() == 2;
assert constraintDescriptor.isReportAsViolationFromCompositeConstraint() == true

//@NotEmpty cannot be null
boolean notNullPresence = false;
for ( ConstraintDescriptor composingDescriptor : constraintDescriptor.getComposingConstraints() ) {
```

```
    if ( composingDescriptor.getAnnotation().getAnnotationType().equals( NotNull.class ) ) {
        notNullPresence = true;
    }
}
assert notNullPresence;

//assuming the implementation returns the Length constraint second
constraintDescriptor = propertyDescriptor.getConstraintDescriptors().iterator().next().next();
assert constraintDescriptor.getAnnotation().getAnnotationType().equals( Length.class );
assert constraintDescriptor.getParameters().get("max") == 30;
assert constraintDescriptor.getGroups().length == 0;

propertyDescriptor = bookDescriptor.getConstraintsForProperty("author");
assert propertyDescriptor.getConstraintDescriptors().size() == 1
assert propertyDescriptor.isCascaded()
```

6

Built-in Constraint definitions

The specification defines a small set of built-in constraints. Their usage is encouraged both in regular constraint declarations and as composing constraints. Using this set of constraints will enhance portability of your constraints across constraint-consuming frameworks relying on the metadata API (such as client side validation frameworks or database schema generation frameworks).

All built-in constraints are in the `javax.validation.constraints` package. Here is the list of constraints and their declaration.

Example 6.1. @Null constraint

```
/**
 * The annotated element must be null.
 * Accepts any type.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
public @interface Null {
    String message() default "{validator.null}";

    Class<?>[] groups() default { };
}
```

Example 6.2. @NotNull constraint

```
/**
 * The annotated element must not be <code>null</code>.
 * Accepts any type.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
public @interface NotNull {
    String message() default "{validator.notNull}";

    Class<?>[] groups() default { };
}
```

Example 6.3. @AssertTrue constraint

```
/**
 * The annotated element must be true.
 * Supported types are <code>boolean</code> and <code>Boolean</code>
 * <p/>
 * <code>null</code> elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
public @interface AssertTrue {
    String message() default "{validator.assertTrue}";

    Class<?>[] groups() default { };
}
```

Example 6.4. @AssertFalse constraint

```
/**
 * The annotated element must be false.
 * Supported types are <code>boolean</code> and <code>Boolean</code>
 * <p/>
 * <code>null</code> elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
public @interface AssertFalse {
    String message() default "{validator.assertFalse}";

    Class<?>[] groups() default { };
}
```

Example 6.5. @Min constraint

```
/**
 * The annotated element must be a number whose value must be greater or
 * equal than the specificed minimum
 * <p/>
 * Supported types are:
 * <ul>
 * <li><code>BigDecimal</code></li>
 * <li><code>BigInteger</code></li>
 * <li><code>Number</code></li>
 * <li><code>String</code></li>
 * <li><code>byte</code>, <code>short</code>, <code>int</code>, <code>long</code>,
 * <code>float</code>, <code>double</code></li>
 * </ul>
 * <p/>
 * <code>null</code> elements are considered valid
 *
 * @author Emmanuel Bernard
 * @todo support byte ?!
 * @todo Is string supported or not?
 */
```

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
public @interface Min {
    String message() default "{validator.min}";

    Class<?>[] groups() default { };

    /**
     * @return Value the element must be higher or equal to
     */
    long value();
}
```

Example 6.6. @Max constraint

```
/**
 * The annotated element must be a number whose value must be lower or
 * equal than the specificed maximum.
 * <p/>
 * Supported types are:
 * <ul>
 * <li><code>BigDecimal</code></li>
 * <li><code>BigInteger</code></li>
 * <li><code>Number</code></li>
 * <li><code>String</code></li>
 * <li><code>byte</code>, <code>short</code>, <code>int</code>, <code>long</code>,
 * <code>float</code>, <code>double</code></li>
 * </ul>
 * <p/>
 * <code>null</code> elements are considered valid
 *
 * @author Emmanuel Bernard
 * @todo support byte ?!
 * @todo Is string supported or not?
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
public @interface Max {
    String message() default "{validator.max}";

    Class<?>[] groups() default { };

    /**
     * @return Value the element must be lower or equal to
     */
    long value();
}
```

Example 6.7. @Size constraint

```
/**
 * The annotated element size must be between the specified boundaries (included).
 *
 * Supported types are:
 * <ul>
 * <li><code>String</code> (string length is evaludated)</li>
```

```
 * <li><code>Collection</code> (collection size is evaluated)</li>
 * <li><code>Map</code> (map size is evaluated)</li>
 * <li>Array (array length is evaludated)</li>
 *
 * <code>null</code> elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
public @interface Size {
    String message() default "{validator.min}";
    Class<?>[] groups() default {};

    /**
     * @return size the element must be higher or equal to
     */
    int min() default Integer.MIN_VALUE;

    /**
     * @return size the element must be lower or equal to
     */
    int max() default Integer.MAX_VALUE;
}
```

Example 6.8. @Digits constraint

```
/**
 * The annotated element must be a number within accepted range
 * Supported types are:
 * <ul>
 * <li><code>BigDecimal</code></li>
 * <li><code>BigInteger</code></li>
 * <li><code>Number</code></li>
 * <li><code>String</code></li>
 * <li><code>short</code>, <code>int</code>, <code>long</code>, <code>float</code>,
 * <code>double</code></li>
 * </ul>
 * <p/>
 * <code>null</code> elements are considered valid
 *
 * @author Emmanuel Bernard
 * @todo support byte ?!
 * @todo Is string supported or not?
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
public @interface Digits {
    String message() default "{validator.digits}";

    Class<?>[] groups() default { };

    /**
     * @return maximum number of integral digits accepted for this number
     */
    int integer();

    /**
     * @return maximum number of fractional digits accepted for this numbe
     */
```

```
    int fraction();
}
```

Example 6.9. @Past constraint

```
/**
 * The annotated element must be a date in the past.
 * Now is defined as the current time according to the virtual machine
 * The calendar used if the compared type is of type <code>Calendar</code>
 * is the calendar based on the current timezone and the current locale.
 * <p/>
 * TODO what are the implications
 * <p/>
 * Supported types are:
 * <ul>
 * <li><code>java.util.Date</code></li>
 * <li><code>java.util.Calendar</code></li>
 * </ul>
 * - TODO new date/time JSR types?
 * <p/>
 * <code>null</code> elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
public @interface Past {
    String message() default "{validator.past}";

    Class<?>[] groups() default { };
}
```

Example 6.10. @Future constraint

```
/**
 * The annotated element must be a date in the future.
 * Now is defined as the current time according to the virtual machine
 * The calendar used if the compared type is of type <code>Calendar</code>
 * is the calendar based on the current timezone and the current locale.
 * <p/>
 * TODO what are the implications
 * <p/>
 * Supported types are:
 * <ul>
 * <li><code>java.util.Date</code></li>
 * <li><code>java.util.Calendar</code></li>
 * </ul>
 * - TODO new date/time JSR types?
 * <p/>
 * <code>null</code> elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
public @interface Future {
```

```
    String message() default "{validator.future}";

    Class<?>[] groups() default { };
}
```

Built-in annotations are not annotated with `@ConstraintValidator` to avoid any dependency between the specification API and a specific implementation. Each Bean Validation provider must recognize built-in constraint annotations as valid constraint definitions and provide a compliant constraint implementation for each.

Warning

Add @Like, @AlphaNumerical? other subset of Regexp? Regexp seems too generic and not descriptive enough.

7

XML deployment descriptor

Two kind of XML descriptors are used by Bean Validation. The first one involves the Bean Validation configuration and is provided as `META-INF/validation.xml`. The second involves representing constraints declarations on classes, it will closely match the annotations declaration approach.

XML descriptors will be completed for the proposed final draft.

A

Terminology

This appendix aims at giving an overview on the different specific terms used through this specification. There are not to be considered formal definitions. Formal definitions are to be inferred from the core specification.

Table A.1. terminology

Term	Definition
Constraint	A restriction on a bean instance, the value of a field or the value of a JavaBean property
Constraint declaration	Assignment of a constraint on a target (bean, field, property) for a specific class. Typically by declaring an annotation on the target but can also be done through a deployment descriptor in XML
Validation routine	Implementation of the validation algorithm associated to a given constraint Also means, sequence of operations executed by the Bean Validation provider to validate a given object
Constraint definition	Defines a type of constraint, it's attributes and the actual constraint implementation. Usually done through annotations, this definition can also be done through XML
group	Constraints can belong to one or more group or context. Useful to apply a subset of the constraints for a given use case. By default, the `Default` group is used.
group sequence	Define a group ordering in the validation process. If a given ordered group contains one or more failure, the following ones in the sequence must be ignored.
Constraint validation	constraint logic algorithm used to determine whether a given value passes a constraint or not.
Constraint validation implementation	Class implementing the constraint logic and used to determine whether a given value pass a constraint or not.
Validation API	Main API used to validate a given type of bean
Bean validation provider	Implementation of the specification

Term	Definition
Message interpolator	Algorithm used to build the end user message associated to a constraint failure. Typically useful for i18n
Constraint metadata API	API exposing the constraints applied to a given bean type. Also considered one of the integration points with other JSR or frameworks.
javax.validation.ConstraintValidator	interface implemented by a constraint validation implementation
Composing constraint	Constraint declared on another constraint definition. When the main constraint is validated, the composing constraints are validated too.
javax.validation.Validator	Main interface for the validation API
javax.validation.ConstraintViolation	interface describing a given constraint failure on a given bean

B

Standard ResourceBundle messages

The properties listed below are resolved by the default message interpolator.

```
validator.null=must be null
validator.notNull=must not be mull
validator.assertTrue=assertion failed
validator.assertFalse=assertion failed
validator.min=must be greater than or equal to {value}
validator.max=must be less than or equal to {value}
validator.size=size must be between {min} and {max}
validator.digits=numeric value out of bounds (<{integer} digits>.<{fraction} digits> expected)
validator.past=must be a past date
validator.future=must be a future date
```

C

Proposal for method-level validation

This proposal is not yet present in the specification but is considered for inclusion due to numerous feedbacks from the early draft.

A popular demand was to provide a method and parameter level validation mechanism reusing the constraint descriptions of the specification. This set of APIs is meant to be used by interceptor frameworks such as:

- application frameworks like Web Beans
- component frameworks like Enterprise Java Beans
- aspect based frameworks

These frameworks can call the validation APIs to validate either the parameter list or the returned value of a method when such method is called. More precisely, validation occurs around a method invocation.

This extension of the Bean Validation API allows to reuse the core engine as well as the constraint definition and declaration for such method level validations.

The following APIs are added to `Validator`.

```
public interface Validator {
    // [...]


    /**
     * Validate each parameter value based on the constraints described on
     * the parameters of <code>method</code>.
     *
     * @param clazz class hosting the method
     * @param method the method whose parameters are currectly validated
     * @param parameterValues the parameter values passed to the method for invocation
     * @param groups groups targeted for validation
     *
     * @return set of constraint violations
     *
     * @throws IllegalArgumentException if the method does not belong to <code>T</code>
     *         or if the Object[] does not match the method signature
     */
    <T> Set<ConstraintViolation<T>> validateParameters(Class<T> clazz, Method method,
                                                       Object[] parameterValues,
                                                       Class<?>... groups);

    /**
     * Validate the parameter value based on the constraints described on
     * the parameterIndex-th parameter of <code>method</code>.
     *
     * @param clazz class hosting the method
     * @param method the method whose parameters are currectly validated
     * @param parameterValue the parameter value passed to the parameterIndex-t parameter of method
```

```
     * @param parameterIndex parameter index of the parameter validated in method
     * @param groups groups targeted for validation
     *
     * @return set of constraint violations
     *
     * @throws IllegalArgumentException if the method does not belong to <code>T</code>
     *         or if prameterIndex is out of bound
     */
    <T> Set<ConstraintViolation> validateParameter(Class<T> clazz, Method method,
                                                   Object parameterValue,
                                                   int parameterIndex, Class<?>... groups);

    /**
     * Validate each parameter value based on the constraints described on
     * <code>method</code>.
     *
     * @param clazz class hosting the method
     * @param method the method whose result is validated
     * @param returnedValue the value returned by the method invocation
     * @param groups groups targeted for validation
     *
     * @return set of constraint violations
     *
     * @throws IllegalArgumentException if the method does not belong to <code>T</code>
     */
    <T> Set<ConstraintViolation> validateReturnedValue(Class<T> clazz, Method method,
                                                       Object returnedValue, Class<?>... groups);

    /**
     * Validate each parameter value based on the constraints described on
     * the parameters of <code>constructor</code>.
     *
     * @param clazz class hosting the constructor
     * @param constructor the constructor whose parameters are currectly validated
     * @param parameterValues the parameter values passed to the constructor for invocation
     * @param groups groups targeted for validation
     *
     * @return set of constraint violations
     *
     * @throws IllegalArgumentException if the constructor does not belong to <code>T</code>
     *         or if the Object[] does not match the constructor signature
     */
    <T> Set<ConstraintViolation> validateParameters(Class<T> clazz, Constructor constructor,
                                                    Object[] parameterValues, Class<?>... groups);

    /**
     * Validate the parameter value based on the constraints described on
     * the parameterIndex-th parameter of <code>constructor</code>.
     *
     * @param clazz class hosting the constructor
     * @param constructor the method whose parameters are currectly validated
     * @param parameterValue the parameter value passed to the
     *                       parameterIndex-th parameter of constructor
     * @param parameterIndex parameter index of the parameter validated in constructor
     * @param groups groups targeted for validation
     *
     * @return set of constraint violations
     *
     * @throws IllegalArgumentException if the constructor does not belong to <code>T</code>
     *         or if prameterIndex is out of bound
     */
    <T> Set<ConstraintViolation> validateParameter(Class<T> clazz, Constructor constructor,
                                                   Object parameterValue, int parameterIndex,
                                                   Class<?>... groups);
```

The constraints declarations evaluated are the constraints hosted on the parameters of the method or constructor. If `@Valid` is placed on a parameter, constraints declared on the object itself are considered.

`validateReturnedValue` evaluates the constraints hosted on the method itself. If `@Valid` is placed on the method, the constraints declared on the object itself are considered.

```
public @NotNull String saveItem(@Valid @NotNull Item item, @Max(23) BigDecimal price)
```

In the previous example,

- `item` is validated against `@NotNull` and all the constraints it hosts
- `price` is validated against `@Max(23)`
- the result of `saveItem` is validated against `@NotNull`

Note that the Bean Validation specification does not trigger the validation call. An external framework is responsible for calling one of the `validateParameters`, `validateParameter` and `validateReturnedValue` methods at the appropriate time.

For completeness, Bean Validation exposes metadata for constraints hosted on parameters and methods.

```
public interface ParameterDescriptor extends ElementDescriptor {
    boolean isCascaded();
    int getIndex();
}

public interface MethodDescriptor extends ElementDescriptor {
    List<ParameterDescriptor> getParameterDescriptors(); //index aligned
    boolean isCascaded();
}

public interface ConstructorDescriptor extends ElementDescriptor {
    List<ParameterDescriptor> getParameterDescriptors(); //index aligned
}

public interface BeanDescriptor {
    MethodDescriptor getConstraintsForMethod(Method);
    MethodDescriptor getConstraintsForConstructor(Constructor);
    Set<String> getConstrainedProperties();
    Set<Method> getConstrainedMethods();
    Set<Constructor> getConstrainedConstructors();
}
```

D

Proposal for Java Persistence 2.0 integration

This proposal is been discussed in the Java Persistence 2.0 expert group for evaluation. It does not represent the final views of the JPA 2.0 expert group and is here to give a global picture of how Bean Validation can be brought to the EE ecosystem.

```
Java Persistence / Bean Validation integration proposal

Integration between Java Persistence and Bean Validation (BV) happens at two levels:
- as a metadata provider for DDL generation and generally speaking the behavior of
the persistence provider. This approach will be treated in a different email.
- as a validation engine called on entity changes


** Using the Bean Validation (BV) engine inside a Java Persistence provider

Java Persistence delegates the validation of entities to Bean Validation on the following
entity events:
- pre persist (an entity must be valid to be inserted in the DB)
- pre update (an entity change must be valid before being propagated to the database)
- pre remove (eg. a Customer still owning some Payments cannot be removed)

These events are called at the same time as their respective callback methods after
all callbacks have been called.

For each event type, a list of groups is targeted for validation. By default,
pre persist and pre update events validate the group Default (default BV group)
and pre remove does not validate any group. Each list can be overridden
by a JPA property:
- javax.persistence.validation.group.pre-persist for pre persist events
- javax.persistence.validation.group.pre-update for pre update events
- javax.persistence.validation.group.pre-remove for pre remove events

Question: should JPA introduce the following groups
/** Defaut validation group used when an entity is persisted */
public interface Persist extends Default {}

/** Defaut validation group used when an entity is updated */
public interface Update extends Default {}

/** Defaut validation group used when an entity is removed */
public interface Remove {}


Each property holds the list of targeted groups (fully qualified class name)
separated by a comma.

When any of the pre-cited event is raised on an entity instance 'a' of type A,
the persistence provider must validate 'a' by invoking a validator with the
targeted groups. If the list of targeted groups is empty,
no validation is performed. If the set of constraint violation is not empty,
a javax.validation.ValidationException containing a reference to this set of violations is raised.


** Special need from Java Persistence
```

```
Validator instances used to validate entities on pre-* events must use a TraversableResolver
implementing the following logic:
- Lazy properties or associations must not be traversed
- Single or multi valued associations pointing to entities must not be traversed

A typical implementation could rely on
Persistence.isLoaded(Object objectHostingTheProperty, String property)
to determine the load state of the entity properties.
In the situation where only a singe (ie the default) persistence provider is available
in the classpath, the container can implement a potentially optimized TraversableResolver
that does not rely on Persistence.isLoaded(Object objectHostingTheProperty, String property).

The TraversableResolver instance is passed to Bean Validation through
ValidatorFactory.usingContext().traversableResolver(jpaSpecificTR).getValidator()

These rules guarantee that:
- no lazy property nor association will be loaded by side effect
- no entity will be validated more than once even if it is reachable through @Valid several
times by entities changing state. This keeps JPA cascading and BV orthogonal.


** Interaction between Persistence and Validation Providers
BV metadata APIs and runtime are available from ValidatorFactory instances. It
is recommended to cache ValidatorFactory instances and retrieve Validator on demand.
ValidatorFactory provides metadata informations to the JPA provider, it will be built at
or before EntityManagerFactory initialization.

Providing the ValidatorFactory
-- in EE
A ValidatorFactory is made available at the platform level by the Java EE container. This
ValidatorFactory instance is passed to the persistence provider via the configuration
Map of PersistenceProvider.createContainerEntityManagerFactory(PersistenceUnitInfo, Map).
The property name is
javax.persistence.validation.factory (Persistence.VALIDATOR_FACTORY)
Note that the container must pass an instance of VF.


-- in SE
Out of container, the Persistence provider access to the ValidatorFactory in two ways:
 - it is provided by the application via the configuration Map of
    Persistence.createEntityManagerFaactory(Map). The property name is
    javax.persistence.validation.factory  (Persistence.VALIDATOR_FACTORY)
 - if not provided by the user, and if Bean Validation is present in the classpath,
    the ValidatorFactory is instantiated using the default bootstrap
    approach ie Validation.buildDefaultValidatorFactory() which uses the configuration
     defined in META-INF/validation.xml


By default:
 - if a Bean Validation provider is present in the environment, the JPA provider
enables the Bean Validation integration features.
 - if no BeanValidation provider is present in the environment, the JPA provider
disable the Bean Validation integration features.
This default behavior is known as the "auto" mode.

QUESTION: should auto encompass ddl the undefined ddl modification behaviors
to prevent future backward compatibility headache?

When the mode "callback" is activated, the JPA provider
enables the callback based Bean Validation integration features or raises an exception if
no Bean Validation provider is present in the environment.

QUESTION: callback or event?

When the mode "none" is activated, the JPA provider
does not enable  the Bean Validation integration features.
```

```
The validation  mode which defaults to "auto" can be defined via a property
javax.persistence.validation.mode  (Persistence.VALIDATION_MODE)

Such a flag can be defined via the configuration Map at EMF creation time
or in META-INF/persistence.xml

//add constants
public Persistence {
    /** property key receiving a ValidatorFactory instance as value */
    public static final String VALIDATOR_FACTORY = "javax.persistence.validation.factory"

    /** property key receiving the validation mode as a String*/
    public static final String VALIDATOR_FACTORY = "javax.persistence.validation.mode"

   [...]
}
```

```
********* OLD do not read **************
Java Persistence / Bean Validation integration proposal

Integration between Java Persistence and Bean Validation (BV) happens at two levels:
 - as a metadata provider for DDL generation and generally speaking the behavior of
the persistence provider
 - as a validation engine called on entity changes

This proposal will describe both integration levels.

** Metadata provider and DDL generation
* main proposal

NOTE: this aspect of the proposal will be integrated to JPA after the runtime
aspect if time permits.

Java Persistence consumes Bean Validation (BV) metadata to enhance persistence property
metadata.

A Persistence provider must use the BV metadata of a given list of groups.
The default group evaluated is Default (default BV group). Groups evaluated
can be overridden by a JPA property javax.persistence.validation.group.metadata.
This contains the comma separated groups (fully qualified class name).

For each entity, apply the following algorithm.
For each persistent property in a given entity:
 - extract the list of BV constraints (including the composing constraints)
 - determine the subset of applicable constraints
    (ie constraints understood by the persistence provider)
 - apply these constraints on the persistent property metadata
 - if the property type is an embeddable object or a collection
of embeddable objects, apply the algorithm on the embeddable object properties

The list of constraints that must be understood by persistence providers are
as followed:
 - @NotNull should be considered equivalent to @Column(nullable=false) /
     @JoinColumn(nullable=false)
 - @Size.max should be considered equivalent to @Column.length
     for String properties
 - @Digits (which contains integer and fraction) should be considered
     equivalent to @Column.precision = integer+fraction,
     @Column.scale = fraction for decimal columns

The BV annotation metadata should have priority over JPA metadata
(JPA has no sensible "unset" values on their annotations).

Question: should we add @Unique that would map to @Column(unique=true)?
```

```
@Unique cannot be tested at the Java level reliably but could generate
a database unique constraint generation. @Unique is not part
of the BV spec today.

Persistence Provider should optionally recognize and try to apply the
following constraints as well:
 - @Min / @Max on numeric columns (TODO String too?)
 - @Future / @Past on temporal columns
 - @Size for collections and array (not sure it is feasible).

Persistence Providers can also apply non standard constraints to their metadata model.
For example, provider ACME might recognize and understand @com.acme.validation.Email
and apply it to the database model.

While most high level constraints will not be recognize, the BV built-in constraints
will be the common language spoken by Persistence Providers. Any high level constraint
can be composed of more modular constraints (constraint composition).

* additional proposition: consider JPA annotations as constraint definitions
Some people have asked for BV to recognize Java Persistence annotations as legit
constraint definitions. Besides making BV more complex, I am not sure of the
actual benefit of such an approach. I am even doubtful of the value as a legacy
migration too. It also raises multiple problems including how to define proper
error messages and groups. This proposal seems to be a good candidate for BV
provider extension if a provider is interested.
  => proposal received no interest from the EG so far

* additional proposal
In case of a constraint violation report detected and generated by the database
(not null, etc), the Java persistence provider catches this report and translates
it into a BV error report. From the perspective of the application, constraint
errors are viewed through a unified layer. BV must provide some API to create a
constraint violation error (constraintDescriptor.createConstraintViolation(...)).

While this proposal has a lot of value-add, I wonder how difficult it can be to
implement this in persistence providers.


** Using the Bean Validation (BV) engine inside a Java Persistence provider
NOTE: This part of the proposition has been worked most extensively by the exert group.

Java Persistence delegates the validation of entities to BV on the following entity events:
- pre persist (an entity must be valid to be inserted in the DB)
- pre update (an entity change must be valid before being propagated to the database)
- pre remove (eg. a Customer still owning some Payments cannot be removed)
- question has been asked about post load. What would be the value add?

These events are called at the same time as their respective callback methods but
after all user registered callbacks have been called.

For each event type, a list of groups is targeted for validation. By default,
pre persist and pre update events vaildate the group Default (default BV group)
and pre remove does not validate any group. Each list can be overridden
by a JPA property:
- javax.persistence.validation.group.pre-persist for pre persist events
- javax.persistence.validation.group.pre-update for pre update events
- javax.persistence.validation.group.pre-remove for pre remove events

Each property holds the list of targeted groups (fully qualified class name)
separated by a comma.

When any of the pre-cited event is raised on an entity instance 'a' of type A,
the persistence provider must validate 'a' by invoking a validator with the
targeted groups. If the list of targeted groups is empty,
no validation is performed. If the set of constraint violation is not empty,
a ValidationException containing a reference to this set of violations is raised.
```

```
Question: should we raise the exception at the first invalid object and stop
(easy way)? Or should all objects in the persistence context expected
to raise a pre-* event be passed to the pre-* events and their constraint
violations merged into a single ValidationException
 => as defined today, the first invalid object will stop and raise an exception

Question: should ValidationException be defined in JPA or BV?
 => BV seems a more natural place

** Lazy loaded state boundaries

Validator instances used to validate entities on pre-* events must use a TraversableResolver
implementing the following logic:
 - Lazy properties or associations must not be traversed
 - Single or multi valued associations pointing to entities must not be traversed

A work is in progress to define a static method provider-agnostic to determine the
fetching state of a property
eg. boolean Persistence.isLoaded(Object objectHostingTheProperty, String property)

This TraversableResolver implementation is provided and injected by the persistence
provider. This implementation is not exposed.

The TraversableResolver instance is passed to Bean Validation through
ValidatorFactory.usingContext().traversableResolver(jpaSpecificTR).getValidator();
The Validator instance retrieved is used by the persistence provider to validate the
entities on pre-* events.

The TraversableRsolver rules guarantee that:
 - no lazy property nor association will be loaded by side effect
 - no entity will be validated more than once even if it is reachable through @Valid several
times due to multiple entities changing state.
A consequence is that JPA cascading and BV cascading are orthogonal for entity associations.

QUESTION: should embeddable objects be validated even if not marked as @Valid?
 => Inclined to say no.


Outside JPA, and if BV validates object graphs loaded by JPA,
BV needs to determine the lazy loaded state boundaries.
BV needs to know whether or not an entity / property is loaded or lazy.
Java Persistence should expose a way to determine this.
Because BV must work on detached objects and on objects made of potentially
multiple EMF, this contract should be persistence provider agnostic.

Persistence.isLoaded(Object objectHostingTheProperty, String property) currently
discussed should fulfill such contract.

When the container (say EE) knows that only a single (ie the default) persistence
provider is available in the classpath, it can implement a potentially optimized
TraversableResolver that does not rely on
Persistence.isLoaded(Object objectHostingTheProperty, String property). This decision
is left to the EE implementor.


** Interactions between Persistence and Validation Providers

BV metadata APIs and runtime are available via Validator instances provided by
a ValidatorFactory. BV recommends to cache ValidatorFactory instances while
retrieving Validators on demand.
Because the ValidatorFactory provides metadata informations to the JPA provider,
it will be built at or before an EntityManagerFactory initialization.

Providing the ValidatorFactory
-- in EE
```

```
A ValidatorFactory is made available at the platform level by the Java EE container. This
ValidationFactory instance is passed to the persistence provider via the PersistenceUnitInfo.
(this point is still under discussion in the expert group)

-- in SE
Out of container, the Persistence provider accesses the ValidatorFactory in two ways:
 - it is provided by the application via the configuration Map of
     Persistence.createEntityManagerFactory(Map). The property name is
     javax.persistence.validation_factory (constant in Persistence?)
 - if not provided by the user, the VF is instantiated using the default bootstrap
     approach. ie Validation.getBuilder().build() which uses the configuration
     defined in META-INF/validation.xml

Note that in the SE case, if the application uses Validation directly, it should
implement a TraversableResolver that does not cross lazy state boundaries
(the implementation is trivial if we have Persistence.isLoaded).

The BV / JPA integration features are enabled by default (provided a Bean Validation
provider is available in the classpath). It is possible to disable them using
javax.persistence.validation.mode metadata|event|all (should metadata be renamed DDL?)
Such a flag can be defined in META-INF/persistence.xml (JPA level).
  => the default value (ie activated by default or not) is under discussion by the EG
```

E

Proposal for Java Server Faces 2.0 integration

This proposal has been built with the help of the Java Server Faces 2.0 expert group for evaluation. This proposal does not bind the Java Server faces 2.0 expert group and is here to give a global picture of how Bean Validation can be brought to the EE ecosystem.

```
Motivation
--------------

To provide bean based validation support in the view layer. This
supports the DRY principle.  The community have trialed this approach
for a couple of years with Hibernate Validator and Seam or MyFaces
extval. As Bean Validation, JSR 303, will be standardized for EE6, we
propose to introduce changes to better support for delegating validation
to an external validation framework and explicit support for Bean
Validation to JSF2.

Proposal
------------
A) GENERAL CHANGES

1) Add a context-param, javax.faces.VALIDATE_EMPTY_FIELDS, by default auto.
If true, all submitted fields will be validated. This is necessary to delegate
validation of whether a field can be null/empty to the model validator.
If false, empty values will not be passed to the validators.
If auto, the default will be true only if Bean Validation
is in the environment, false otherwise (keeps backward compatibility)

BACKWARDS COMPATIBILITY NOTE:

JSF 2 validators must not fail on non-null empty values; for a JSF 1.2
validator to work with JSF2 and VALIDATE_EMPTY_FIELDS, it must be be
safe to use with non-null, empty values. If a non-safe validator is
used, validate-empty-fields should be set to false.

2) Add these elements:

<application>
<default-validators>
<validator-id />

and

public abstract void addDefaultValidatorId(String validatorId);
public abstract Iterable<String> getDefaultValidatorIds();

to javax.faces.application.Application.

The JSF implementation should add the validator to all
EditableValueHolders after any locally defined validators.

If Bean Validation is present in the environment, and if no default
validator is explicitly specified (either via face-confg.xml or via
programmatic API), the validator with id javax.faces.Bean is
considered the default validator.
NOTE: if an application or a container override the default validator
```

```
list (face-config.xml or programmatic API) and wish to keep integration
with Bean Validation, it must manually add javax.faces.Bean to the list.

B) SPECIFIC SUPPORT FOR JSR-303

3) Add javax.faces.validator.BeanValidator with id
javax.faces.Bean. The property validationGroups on BeanValidator is
used to allow the view designer to specify a comma separated list of
groups which should be validated. A group is represented by the fully
qualified class name of its interface. If the validationGroups attribute is
omitted, the Default (javax.validation.groups.Default) group will be used.
If the model validator is set as the default validator, this tag can be
used to specify validation groups for this input.

The BeanValidator requires a version of EL which supports
ValueExpression.getReference(elContext).getProperty(). The validate() method should inspect
the parent EditableValueHolder and discover the ValueExpression specified
for the "value" attribute, and from it the EL expression's resolved
base and property using:

ValueReference ref = valueExpression.getReference(elContext);
String property = ref.getProperty();
Object base = ref.getBase();

The BeanValidator acquires
javax.validator.ValidatorFactory as defined in 5;
it may keep an application scoped cache of it.
The Validator instance should be acquired as defined in 4) and
Validator.validateValue(property, value,
validationGroups); called to find any validation errors.

Bean Validation returns multiple validation constraint failures from a
single validator, whilst JSF is limited to returning a single validation
failure; lifting this constrain is outside the scope of JSF2, but may be
addressed in the future.  The model validator may take any constraint
failure and use it to throw a ValidatorException. The FacesMessage
created for the ValidatorException should have SEVERITY_ERROR, a message
summary taken from the validation error message taken from the Bean
Validation validation error and the detail message should be null.

4) To ensure proper localization of the messages, JSF should provide a
custom BV MessageInterpolator resolving the Locale according to JSF defaults
and delegating to the default MessageInterpolator as defined in
ValidationFactory.getMessageInterpolator()

A possible implementation is described as followed

public class JsfMessageInterpolator {
    private final MessageInterpolator delegate;

    public JsfMessageInterpolator(MessageInterpolator delegate) {
        this.delegate = delegate;
    }

    public String interpolate(String message, ConstraintDescriptor constraintDescriptor,
                              Object value) {
        Locale locale = FacesContext.getCurrentInstance().getUIViewRoot().getLocale();
        return this.delegate.interpolate(
                         message, constraintDescriptor, value, locale );
    }

    public String interpolate(String message, ConstraintDescriptor constraintDescriptor,
                              Object value, Locale locale) {
        return this.delegate.interpolate(message, constraintDescriptor, value, locale);
    }
}
```

```
Assuming a ValidatorFactory, JSF receives a Validator instance by providing the custom message
interpolator to the validator state.

//could be cached
MessageInterpolator jsfMessageInterpolator = new JsfMessageInterpolator(
                  validatorFactory.getMessageInterpolator() );

//...

Validator validator = validatorFactory
          .usingContext()
          .messageInterpolator(jsfMessageInterpolator)
          .getValidator();

5) ValidatorFactory are retrieved using the following algorithm:
 - if the servlet context contains a ValidatorFactory instance under the attribute named
javax.faces.validator.beanValidator.ValidatorFactory, this instance is used by
JSF to acquire Validator instances
 - if the servlet context does not contain such entry, JSF looks for a Bean Validation
provider in the classpath. If present, the standard Bean Validation bootstrap strategy
is used. If not present, Bean Validation integration is disabled.

The standard Bean Validation bootstrap procedure is shown here
ValidatorFactory validatorFactory = Validation.buildDefaultValidatorFactory();

If the BeanValidator is used and no ValidatorFactory can be retrieved,
an exception is raised.

VALIDATOR_FACTORY_KEY is a static final String hosted on BeanValidator whose value
is javax.faces.validator.beanValidator.ValidatorFactory


6) Add a <f:validateBean /> tag. When nested in an EditableValueHolder
this should add the BeanValidator

7) JSR-303 allows the user to validate a graph of objects, JSF will not
support graph validation.

8) An subtree of components, for example a form, can be validated by
directly nesting a <f:validateBean /> element. Validating an entire
form is especially useful if you are not using the BeanValidator as the
default validator, or you want to set a validationGroup for the entire
form. The <f:validateBean /> should not render a UIComponent. Any
validationGroups specified at this level should be used when validating
each component in the subtree. Any validationGroups defined on
<f:validateBean /> nested in an EditableValueHolder take precedence
over the validationGroups defined for the subtree.

9) If JSF is running in EE then bean validation will be present in the
environment and activated in JSF (see rules 1) and 2) ).
The EE container will also make ValidatorFactory available as an
attribute in the ServletContext at
"javax.faces.validator.beanValidator.ValidatorFactory"

10) It is possible to disable the default validator for a field/a subtree
by using the disabled attribute: <h:inputText ...> <f:validateBean
disabled="true" /> </h:inputText> This is useful when you are using the
ModelValidator as the default validator, or validating an entire form

C) AJAX VALIDATION

11) JSF2's Ajax support transparently supports Ajax validation for any JSF
validator, including the BeanValidator. For example you could enable
ajax validation for the onBlur event of an input field:
```

```
<h:inputText value="#{item.name}">
<f:ajaxRequest execute="none" render="none" />
</h:inputText>
```

F

Proposal for Java EE integration

This proposal has not been discussed with the EE expert group but proposes a way to smoothly integrates Bean Validation to the platform. This proposition is the result of the work done on JSF and JPA integrations.

Java Persistence and Java Server Faces have deep integration proposals with Bean Validation. Java Connector Architecture is a potential client. `Validator` and `ValidatorFactory` are very good candidates for injectable Java EE resources (`ValidatorFactory` is responsible for providing `Validator` instances which execute the validation logic). To tight everything together in a unified way for the Java EE application developer, some integration at the Java EE level would be beneficial.

A `ValidatorFactory` is built by the EE container and exposed to applications as well as services like JPA, JSF and JCA(?) (this imply a `ValidatorFactory` must be built and ready before these services). While not required, it is recommended to cache and share the same `ValidatorFactory` for a given deployment unit. *Should we mandate it?*.

The container passes the `ValidatorFactory` instance to the JPA provider via the configuration Map of `PersistenceProvider.createContainerEntityManagerFactory(PersistenceUnitInfo, Map)`. The property name is `javax.persistence.validation.factory` (`Persistence.VALIDATOR_FACTORY`).

The container places the `ValidatorFactory` instance into the `ServletContext` in the following attribute name entry `javax.faces.validator.beanValidator.ValidatorFactory`. JSF 2 implementations are required to look in this entry for any `ValidatorFactory` and use it.

In addition, `Validator` and `ValidatorFactory` should be considered as Java EE resources: injection aware services should allow injection of `Validator` instances as well as `ValidatorFactory` instances. `@Resource` is used for this. If `Validator` instances are injected, the default validation context is used. In other words, `Validator` are provided by `validatorFactory.getValidator()`. The factory is available for the lifetime of the deployment.