Constraint Aspects

Constraint Aspects/Parameters

This feature is available in MyFaces ExtVal in the 3rd release.
Before the official release you can use it via the snapshot version.

The problem

```
//not typesafe:
@Required(parameters = {@Param(key= "severity", value = "warn"), @Param(key= "display", value = "global")})
```

... this version allows you to add general information to a constraint. You can think about it as an extension point for constraints.

Constraint aspects allow to provide a typesafe alternative. So you can provide an ExtVal add-on without knowing the concrete constraint implementation.
Only a central logic (e.g. an add-on) has to know what information might be available as parameter at the constraint and how to use this information.
The final usage is quite easy (see the next example).

So you can have add-on-X which provides new features and constraints Y (they don't know each other). But you can use the features of add-on-X with constraints Y (if they have an attribute of the type `Class<? extends ValidationParameter>[]`)

How to use constraint parameters

Example usage of an ExtVal constraint + validation parameters:

The solution

```
@Required(parameters = {ViolationSeverity.Warn.class, DisplayGlobal.class})
```

That's it!!!

You can see - the final usage is typesafe, shorter and the constraint isn't aware of the specific values behind.
So if you have existing constraints you just have to add an attribute of the type `Class<? extends ValidationParameter>[]` and your constraint can automatically use any feature provided via a parameter implementation of ExtVal or an ExtVal add-on.

Stop reading here if you don't like to implement a custom parameter or if you don't like to query information of a parameter

Before you continue reading - constraint aspects don't replace "domain-attributes".
E.g. the values of min and max of `@Length` are attributes of the constraint which might be different at every usage.
Furthermore, they aren't interesting for other constraints like `@Equals`.
So it doesn't make sense to use a generic concept to replace such attributes.

Anyway, there are use-cases for shared features. ExtVal uses this mechanism e.g. for severities. Constraint aspects are interesting for features which allow a small set of possible values and which are interesting for different constraints of the whole application.
Without a generic solution you have to introduce one attribute per feature for every use-case at every constraint. And the implementation which uses these information have to be aware of all types of constraints (esp.: if you don't have a fix attribute name for it).

Constraint aspects allow you to extend the functionality of existing constraints.
Furthermore, they allow to introduce shared features. The implementations to process the information provided by the aspects/parameters aren't aware of the attribute name of the parameter nor of the concrete type of the constraint, parameter,... That means you provide a generic aspect attribute mechanism once and use it for different constraints and features.

More Examples

You can query the parameters of a constraint at any time. (You just need access to the constraint.)

Example for a custom constraint

```
public @interface MyConstraint {
  Class<? extends ValidationParameter>[] params() default ViolationSeverity.Error.class;
}
```

The important part is: `Class< ? extends ValidationParameter >`
You can choose any attribute-name! (An array isn't required)
A parameter implementation has to implement this marker interface. The values within the implementation are marked via `@ParameterValue`. Optionally you can provide @ParameterKey. If you don't specify an explicit key, the interface extending the ValidationParameter interface is the key.

Examples for possible parameter implementation styles

Parameter implementation - style 1

Direct implementation (just an interface - no concrete class) + grouped values

Style 1

```
public interface ViolationSeverity {
  interface Warn extends ValidationParameter {
    @ParameterKey
    public Class KEY = ViolationSeverity.class;

    @ParameterValue
    FacesMessage.Severity SEVERITY = FacesMessage.SEVERITY_WARN;
  }

  interface Error extends ValidationParameter {
    @ParameterKey
    public Class KEY = ViolationSeverity.class;

    @ParameterValue
    FacesMessage.Severity SEVERITY = FacesMessage.SEVERITY_ERROR;
  }
  // the other severities ...
}
```

Parameter implementation - style 2

Direct implementation (just an interface - no concrete class) + key that doesn't introduce a class dependency

Style 1

```
public interface AllowClientSideValidation extends ValidationParameter {
  @ParameterKey
  public String key = "client_side_validation_support";

  @ParameterValue
  boolean value = true;
}

public interface RestrictClientSideValidation extends ValidationParameter {
  @ParameterKey
  public String key = "client_side_validation_support";

  @ParameterValue
  boolean value = false;
}
```

Parameter implementation - style 3

Class with implicit key + mixed value types + multiple values with the same type

Style 3

```
public interface Priority extends ValidationParameter {
  @ParameterValue
  Integer getValue();

  @ParameterValue(id = ShortDescription.class)
  String getShortDescription();

  @ParameterValue(id = LongDescription.class)
  String getLongDescription();

  interface ShortDescription{}
  interface LongDescription{}
}

public class HighPriority implements Priority {
  public Integer getValue() {
    return 1;
  }

  public String getShortDescription() {
    return "do it asap";
  }

  public String getLongDescription() {
    return "do it immediately";
  }
}

public class LowPriority implements Priority {
  public Integer getValue() {
    return 3;
  }

  public String getShortDescription() {
    return "not that important";
  }

  public String getLongDescription() {
    return "the topic is not that important";
  }
}
```

Parameter implementation - style 4

Class with implicit key + the implementation provides the final value (the method isn't given by the interface) (PropertyValidationInterceptor extends ValidationParameter)

Style 4

```
public class CustomValidationInterceptor implements PropertyValidationInterceptor
{
    private static PropertyValidationInterceptor propertyValidationInterceptor;

    //called at every parameter query
    @ParameterValue
    public PropertyValidationInterceptor getInstance()
    {
        if(propertyValidationInterceptor == null)
        {
            propertyValidationInterceptor = new CustomValidationInterceptor();
        }
        return propertyValidationInterceptor;
    }

    public boolean beforeValidation(
            FacesContext facesContext, UIComponent uiComponent, Object convertedObject, Map<String, Object>
properties)
    {
        //your implementation
        return true;
    }

    public void afterValidation(
            FacesContext facesContext, UIComponent uiComponent, Object convertedObject, Map<String, Object>
properties)
    {
        //your implementation
    }
}
```

Parameter implementation - style 5

Class with implicit key + implicit value -> a simple marker

Style 5

```
public interface ShowGlobalViolationMessageOnly extends ValidationParameter
{
}
```

Parameter implementation - style 6 and 7

Details are available in ParameterTestCase#testParameterStyleSix

An use-case is available at os890.

Style 6 and 7

```
public class TestPerson
{
    //...

    @Required(parameters = {
            LoginValidator.class,
            AdditionalValidator.class})
    private String name;

    public class LoginValidator extends TestValidatorProvider
    {
        @ParameterValue
        public TestValidationStrategyProvider getValue()
        {
            return this;
        }

        @Override
        public ValidationStrategy getValidationStrategy()
        {
            return new ValidationStrategy() {

                public void validate(FacesContext facesContext, UIComponent uiComponent, MetaDataEntry
metaDataEntry, Object convertedObject)
                {
                        //...
                }
            };
        }
    }
}

public class AdditionalValidator extends TestValidatorProvider
{
    //...
}
```

Using the constraint and these parameter implementations

Using constraint aspects

```
@MyConstraint(params = {
    ViolationSeverity.Warn.class,
    AllowClientSideValidation.class,
    HighPriority.class,
    CustomValidationInterceptor.class,
    ShowGlobalViolationMessageOnly.class,
    LoginValidator.class,
    AdditionalValidator.class})
```

Query the information

Extract constraint aspects

```
ValidationParameterExtractor extractor = ExtValUtils.getValidationParameterExtractor();

//extract all available parameters - returns a list
extractor().extract(myConstraint)

//query the severity - returns a list
extractor().extract(myConstraint, ViolationSeverity.class, FacesMessage.Severity.class)

//query all information of the priority (independent of the type) - returns a list
extractor().extract(myConstraint, Priority.class)

//query information of the priority - the result is filtered by type - returns a list
extractor().extract(myConstraint, Priority.class, Integer.class)

//query all descriptions of the priority - returns a list
extractor().extract(myConstraint, Priority.class, String.class)

//query a specific description - returns a single result
extractor().extract(myConstraint, Priority.class, String.class, Priority.ShortDescription.class)
```

As you see the extractor is a generic impl. which allows an easier usage of the constraint aspects.
You can see the signatures of the methods at: ValidationParameterExtractor.java

This concept isn't bound to ExtVal. ExtVal just provides one possible implementation which allows different parameter styles which are quite flexible in view of key/value aggregation and much more...