Tickets

User guide

Table of contents

I. Introduction

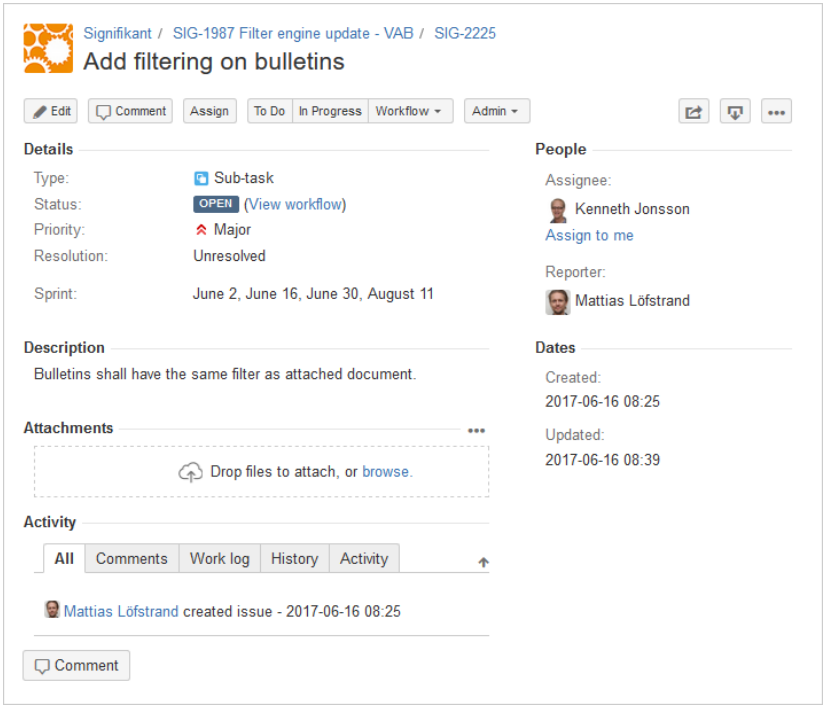
This document explains how to integrate and customize JIRA™ Atlassian tickets in Signifikant Web Viewer. It provides general information on how to configure the Web Viewer to enable the ticket functionality and how to create, customize, and use tickets.

This document is complemented by the ***Wiki***.

Dashboard

A dashboard is a centralized system allowing activity tracking. In all this document, the dashboard will be JIRA (a web application) since it is the system currently supported by the Web Viewer, but all the information explained will apply to whichever dashboard.

JIRA is initially used in software development and provides bug tracking, issue tracking and project management functions. It is generic enough to be used in all kinds of activities.



In a JIRA ticket, it is possible to create tasks, describe and comment them, assign people, set a level of priority, get mail alerts on changes, and many others.

Lots of ticket types can coexist and can be configured to have the fields that correspond to the needs.

For example, manufacturers and resellers can use them to order new parts, retrieve feedback from different workers, make warranty claims, etc…

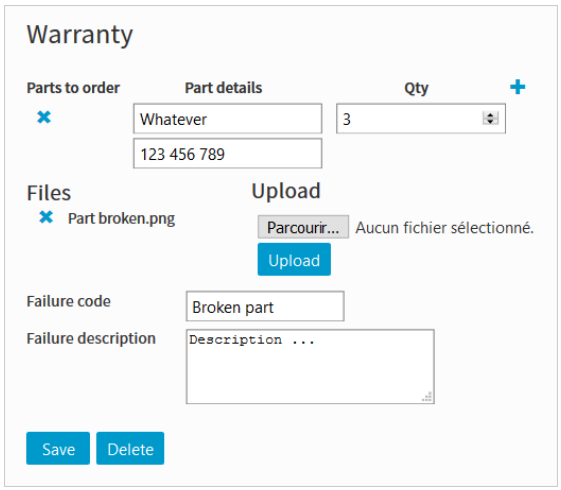
The issue can then be dealt by the appropriated area of the company.

Integration in Signifikant Web Viewer

The ticket functionality in the Web Viewer allows the end-user to create and update JIRA tickets directly in the Web Viewer according to templates written as XML files, which describe how the ticket should appear and behave in the Web Viewer.

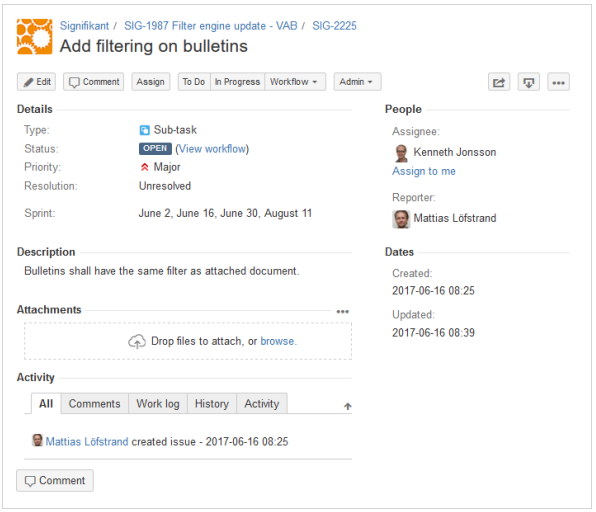
In the end, these XML files describe dynamic web forms whose contents are converted into appropriate dashboard tickets. These tickets can then be viewed and updated at will from the Web Viewer or from JIRA directly.

Signifikant tickets are easy to use and highly customizable. Many fields are available and can interact with each other when described conditions are met. The tickets validation can also be customized as well as their visual aspect, with CSS descriptions.



Signifikant ticket

JIRA ticket



II. User guide

1. Getting ready

Settings

To allow ticket creation, modification, and the viewer access (ability to read tickets already created) from the Web Viewer, proper configuration is needed in the root of ***profile.config*** as shown below:

```
<TicketSettings>
    <CreationEnabled>true</CreationEnabled>
    <ModificationEnabled>true</ModificationEnabled>
    <ViewerEnabled>true</ViewerEnabled>
</TicketSettings>
```

Add a ticket template file

Ticket templates are defined inside one or several XML files: each file can contain as many templates as wanted. Each ticket file can be imported in the Web Viewer by adding their respective path in the root ***profile.config*** as shown below:

```
<TicketPaths>
    <Path>C:\Users\Documents\ticket_1.txt</Path>
    <Path>C:\Users\Desktop\tickets_2_3.config</Path>
    <Path>C:\Users\SomeFolder\ticket_4.xml</Path>
</TicketPaths>
```

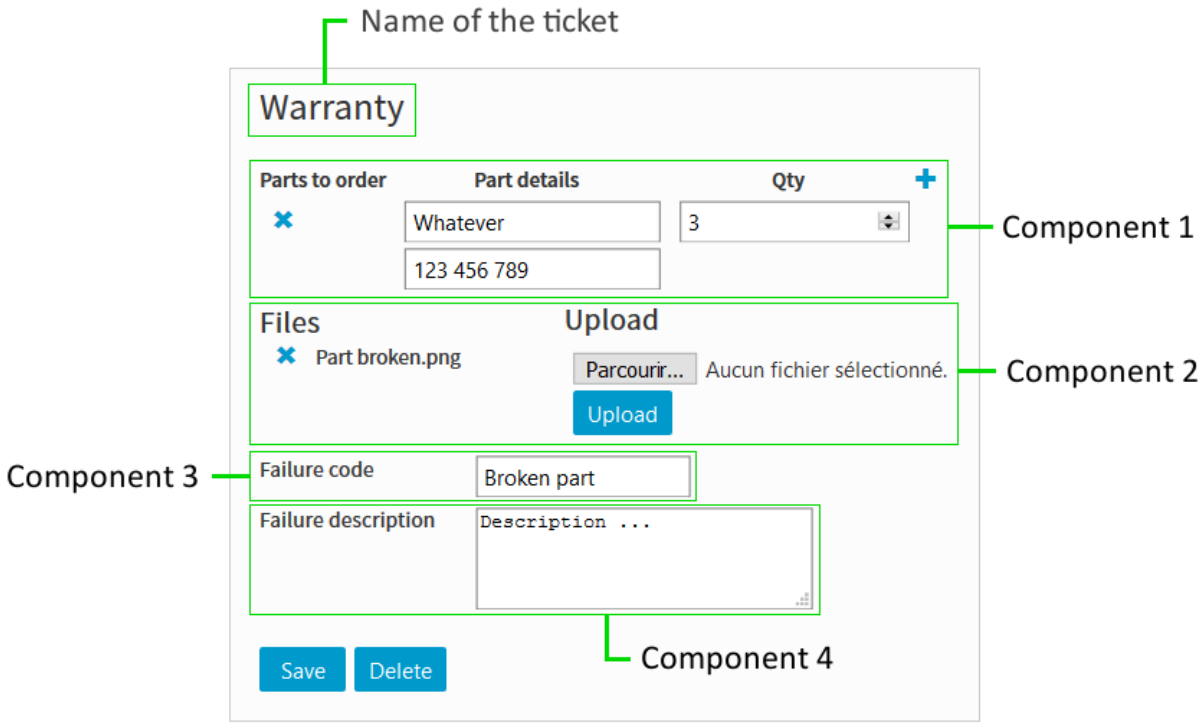
Content of a ticket template file

A ticket file must define an ***AllTickets*** tag in which a succession of ***Ticket*** tag occurs:

```
<AllTickets>

  <Ticket>
    <!-- /* Ticket 1 */ (I'm a comment) -->
  </Ticket>

  <Ticket>
    <!-- /* Ticket 2 */ -->
  </Ticket>

  <Ticket>
    <!-- /* Ticket 3 */ -->
  </Ticket>

</AllTickets>
```

2. Create a ticket template

A ticket is a form that consists of fields called *components*. Each ticket must have a unique name and can have as many components as wanted whose content can potentially be stored in the generated JIRA ticket.



Such a ticket template is defined as:

```
<Ticket>
  <Name>Warranty</Name>

  <Component>
    <!-- /* Component 1 */ -->
  </Component>
  <Component>
    <!-- /* Component 2 */ -->
  </Component>
  <Component>
    <!-- /* Component 3 */ -->
  </Component>
  <Component>
    <!-- /* Component 4 */ -->
  </Component>

</Ticket>
```

Components are described in the later section ***3. Add components***.

Dashboard integration

To indicate which dashboard system the ticket is bonded with, we add a tag called ***DashboardIntegration***:

```
<Ticket>
  <DashboardIntegration>Jira</DashboardIntegration>
</Ticket>
```

We recommend this syntax for such an important tag:

```
<Ticket DashboardIntegration="Jira">

</Ticket>
```

From now, only the dashboard JIRA is supported, but in the future others such as Windchill will have support too.

To configure a JIRA project in the Web Viewer, these settings must be added to the root of ***profile.config***:

```
<TicketProvider>
    <JiraTicketProvider>
        <User>user@enterprise.site</User>
        <Password>password</Password>
        <BaseUrl>https://my_enterprise.atlassian.net</BaseUrl>
        <ProjectId>jira_id</ProjectId>
    </JiraTicketProvider>
</TicketProvider>
```

By default, the ticket generated in JIRA will be a *Task* whose summary starts with *[MyTicketName]*. You can generate your own JIRA ticket by adding the name of the JIRA ticket with the tag ***Type***:

```
<Ticket DashboardIntegration="Jira">
  <Name>Warranty</Name>
  <Type>Claim</Type>
</Ticket>
```

3. Add components

Components are simple and various fields which can be added in a ticket. They can be either a textbox, a checkbox, a slider, and many others, even a component to send mail !

Each component inside a ticket must have a unique **one-word** name which acts as an id, and a type that describes what the component is:

```
<Component>
  <Name>MyDescription</Name>
  <Type>TextBox</Type>
</Component>
```

Such a component will display as:

All the types that can be used are listed in the ***Wiki***.

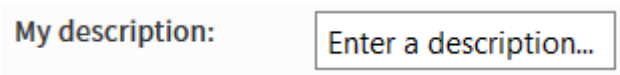
Additional options

Each component, depending on their type, has a specific number of additional options that can be used for customization. For example, the component type *TextBox* has the option ***Default*** that sets the initial text inside the textbox.

Also, most of the components are displayed with their name first, and since it is a one-word name, the option ***DisplayedName*** exists to be able to display a friendlier name:

```
<Component>
  <Name>MyDescription</Name>
  <Type>TextBox</Type>
  <DisplayedName>My description:</DisplayedName>
  <Default>Enter a description...</Default>
</Component>
```

Displays as:



Once again, all the options available for a component type are listed in the ***Wiki***.

Dashboard option

To describe which component content must be in the JIRA ticket and where it should be, we use the option ***Dashboard***:

```
<Component Dashboard="Body">
  <Name>MyDescription</Name>
  <Type>TextBox</Type>
</Component>
```

Without this option, the component will not be exported. Such components are most likely checkboxes or buttons whose purpose is to interact with other components.

An infinity of components can have the dashboard option ***Body*** (and will be exported in the main body of the JIRA ticket). Other dashboard options such as ***Summary*** can only be taken by one component.

Custom fields can also be added in JIRA. To describe which component is bonded to them, we use the dashboard option ***Custom:MyField*** (where *MyField* is the JIRA field name):

```
<Component Dashboard="Custom:MyField">
  <Name>MyDescription</Name>
  <Type>TextBox</Type>
</Component>
```

Component behavior

Behaviors are used to make components change under described conditions. They are used by adding ***Behavior*** tags in a component, composed of sub-tags ***Predicate*** and ***Action***:

```
<Component>
    <Name>MyDescription</Name>
    <Type>TextBox</Type>
</Component>

<Component>
    <Name>MyCheckBox</Name>
    <Type>CheckBox</Type>
    <Behavior>
        <Predicate>MyCheckBox.IsChecked()</Predicate>
        <Action>MyDescription.SetText("hello")</Action>
    </Behavior>
</Component>
```

In this example, the text *hello* will be displayed in the textbox when the checkbox is checked.

Component functions are used using the syntax ***ComponentName.Function(Arg1, Arg2, …)***.

Every component has its own predicate and action functions depending on its type. It is meant to be as versatile as possible, see the section **4. In depth: predicates and actions** to see how predicates and actions can be used. As many behaviors as wanted can be added.

Component validation

To check whether a ticket is valid before sending it in JIRA, components can have ***Validation*** tags with ***Predicates*** and an associated ***Message***:

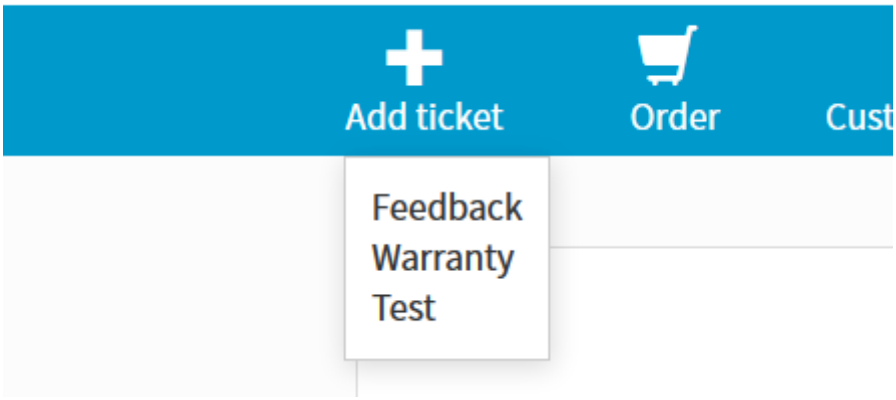
```
<Component>
    <Name>MyDescription</Name>
    <Type>TextBox</Type>

    <Validation>
        <Predicate>MyDescription.GetLenth() > 0</Predicate>
        <Message>Please enter a description</Message>
    </Validation>
</Component>
```

In this example, the textbox must contain some text for the ticket to be validated.

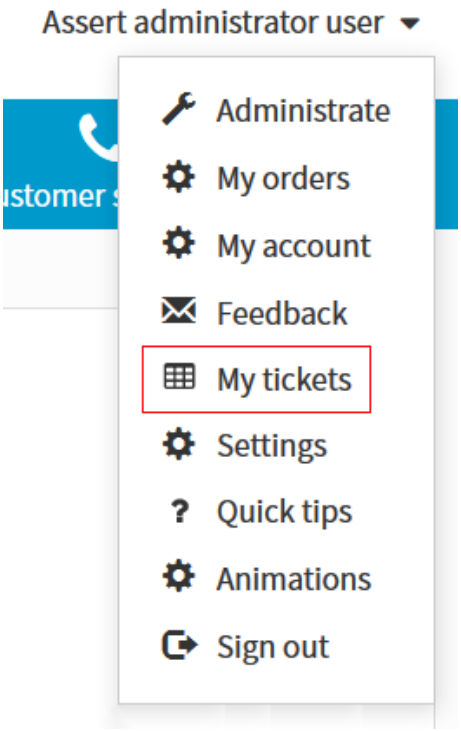
As many validations as wanted can be added.

4. Usage in the Web Viewer

Once tickets are enabled in the Web Viewer and some templates are added to ***profile.config***, tickets can be added using the tool bar:



Created tickets can then be accessed in the top-right dropdown list:



The user will then be redirected to a list of its created ticket. Clicking on an element of the list will allow to edit the ticket.

5. In depth: predicates and actions

Using component name inside the same component

Using the name of a component where the predicates and actions are, is optional. For example, this means these two behaviors are equivalent:

```
<Component>
    <Name>MyDescription</Name>
    <Type>TextBox</Type>
    <Behavior>
        <Predicate>MyCheckBox.IsChecked()</Predicate>
        <Action>MyDescription.SetText("hello")</Action>
    </Behavior>
</Component>

<Component>
    <Name>MyDescription</Name>
    <Type>TextBox</Type>
    <Behavior>
        <Predicate>MyCheckBox.IsChecked()</Predicate>
        <Action>SetText("hello")</Action>
    </Behavior>
</Component>
```

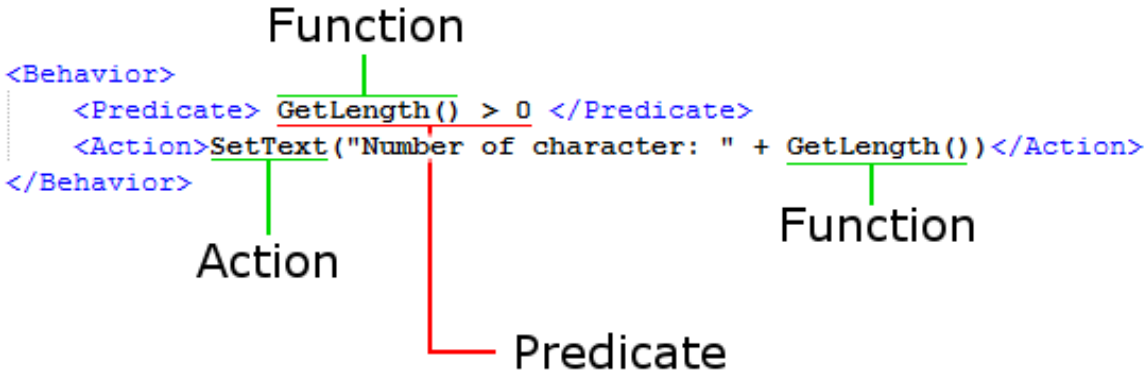
It is also notable that behaviors and validations don't necessarily need to be in their corresponding component. Having all of them inside one component is possible, even though it is not recommended.

Differences between predicates, actions and functions

A predicate is an expression that can be evaluated as a final result which can be either true or false**.** A *Predicate* tag must contain an expression that converts into true or false.

An action is a function of a component that modify the component state and doesn't return a value. Action can only be used inside an *Action* tag.

A function returns a value of a component that can be used inside a predicate or as arguments in another function or action.



Combining predicates

Predicates can be combined using ***And***, ***Or***, ***Equal*** and ***Not*** (in behaviors or validations):

```
<Behavior>
    <Or>
        <And>
            <Predicate> 1 </Predicate>
            <Predicate> 2 </Predicate>
        </And>
        <Predicate> 3 </Predicate>
        <Predicate> 4 </Predicate>
    </Or>
    <Action> ... </Action>
</Behavior>
```

Here, the action will be executed if *p4* is true, or if *p3* is true, or if *p1* and *p2* are true, which can also be written *(p4 or p3 or (p1 and p2))*.

By default, when none of the tags *And*, *Or* and *Equal* are defined, the *And* tag is implicitly used. Consequently, those two predicates are equivalent:

```
<Behavior>
    <And>
        <Predicate> 1 </Predicate>
        <Predicate> 2 </Predicate>
    </And>
    <Action> ... </Action>
</Behavior>

<Behavior>
    <Predicate> 1 </Predicate>
    <Predicate> 2 </Predicate>
    <Action> ... </Action>
</Behavior>
```

The *Equal* tag is true when all its predicates have the same value: they must all be true or all be false.

The *Not* tag is a special tag that invert a predicate result:

```
<Behavior>
    <Not>
        <Predicate> 1 </Predicate>
        <Predicate> 2 </Predicate>
    </Not>
    <Action> ... </Action>
</Behavior>
```

Here, the action will be executed if *(p1 and p2)* is false, meaning that it won't be executed when *p1* and *p2* are both true (note that an implicit tag *And* is surrounding *p1* and *p2* in this example).

C-style predicates

Various predicate combinations can also be defined inside one predicate, using a C-style language:

```
<Behavior>
    <Predicate> p1 && (p2 == p3) && !(p4 || !p5) </Predicate>
    <Action> ... </Action>
</Behavior>
```

&&	is equivalent to	*And*	(logical operator)
||	is equivalent to	*Or*	(logical operator)
!	is equivalent to	*Not*	(logical operator)
==	is equivalent to	*Equal*	(comparison operator)
!=	is equivalent to	*Not Equal*	(comparison operator)

These operators also have a priority evaluation order:

! > == , != > && > || where > means "*has more priority than*".

Operators with the same priority will be evaluated from left to right.

You can also use parentheses to group predicates as shown above. Consequently, the above predicate is equivalent to:

```
<Behavior>
    <Predicate> 1 </Predicate>
    <Equal>
        <Predicate> 2 </Predicate>
        <Predicate> 3 </Predicate>
    </Equal>
    <Not>
        <Or>
            <Predicate> 4 </Predicate>
            <Not>
                <Predicate> 5 </Predicate>
            </Not>
        </Or>
    </Not>
    <Action> ... </Action>
</Behavior>
```

Math operators, comparison operators

While predicates are essentially values that can be true or false, other operators exist to do calculous and comparison between numbers:

```
<Behavior>
    <Predicate> p1 && MyDescription.GetLength() + 4 > 7 </Predicate>
    <Action> ... </Action>
</Behavior>
```

Here, the predicate will be true if *p1* is true and if the textbox has more than 3 characters in its inner text.

Calculous can be used to calculate values to pass to functions in predicates and actions:

```
<Action>SetText("hello" + (GetLength() + 5))</Action>
```

In this example, if the text in the textbox is *"test"*, the text displayed after the evaluation of the action will be *"hello9"* (note that an implicit conversion occurred to append *9* at the end of the string of characters "*hello"*, more info about operators, types and their conversion in the ***Wiki***).

Finally, all the operators available are:

&&	and	is equivalent to	*And*	(logical)
||	or	is equivalent to	*Or*	(logical)
!	not	is equivalent to	*Not*	(logical)
==	equal	is equivalent to	*Equal*	(comparison)
!=	not equal	is equivalent to	*Not Equal*	(comparison)
>	more than			(comparison)
>=	more or equal			(comparison)
<	less than			(comparison)
<=	less or equal			(comparison)
*	multiplication			(math)
/	division			(math)
-	minus			(math)
+	addition			(math)

These operators also have a priority evaluation order:

! > *, / > +, - > >, >=, <, <=, ==, != > && > ||

where > means "*has more priority than*".

Operators with the same priority will be evaluated from left to right.

Common mistake with comparison operators

Here is a mistake commonly done while using comparison operators. This predicate is not doing what is intended:

```
<Predicate> 0 < GetLength() < 10 </Predicate>
```

Wrong

Instead, this predicate should be used:

```
<Predicate> 0 < GetLength() && GetLength() < 10 </Predicate>
```

Several actions, dual actions

Several actions can be defined in a behavior. All the actions will be executed when the whole predicate is true:

```
<Behavior>
    <And>
        <Predicate> p1 </Predicate>
        <Predicate> p2 </Predicate>
    </And>
    <Action> a1 </Action>
    <Action> a2 </Action>
</Behavior>
```

Actions can also be placed inside an ***Else*** tag. These actions will be executed when the whole predicate is false. A common example of their usage is this one:

```
<Behavior>
    <Predicate>GetLength() > 0</Predicate>
    <Action>MyCheckBox.Check()</Action>
    <Else>
        <Action>MyCheckBox.Uncheck()</Action>
    </Else>
</Behavior>
```

In this example, the checkbox will be checked if the textbox has some text inside it.

Some actions of a component are defined as *dual actions*, meaning they come as a pair: one action does the opposite result of the other one. The actions *Check()* and *Uncheck()* are a good example of dual actions.

It is then possible to use the ***DualAction*** tag, meaning one action is executed when the whole predicate is true and the associated dual action is executed when the whole predicate is false. Dual actions are essentially syntactic sugar to make actions more concise.

This behavior is exactly the same as the one shown above:

```
<Behavior>
    <Predicate>GetLength() > 0</Predicate>
    <DualAction>MyCheckBox.Check()</DualAction>
</Behavior>
```

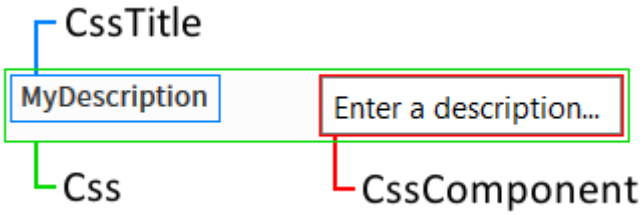
Dual actions can be placed in an *Else* tag, inverting the actions regarding the predicates, and can also be present several times and coexist with regular actions.

6. CSS customization

Customization of tickets and components appearance is done with CSS style sheet. Tickets and each of their components, depending on their type, have several options to apply CSS classes. All possibilities are described in the ***Wiki***.

Example of CSS definition

To make it clearer, here is how the textbox component can be customized:




```
<Component>
    <Name>MyDescription</Name>
    <Type>TextBox</Type>
    <Css>my-class1</Css>
    <CssTitle>my-class2</CssTitle>
    <CssComponent>my-class3</CssComponent>
</Component>
```

CSS file

All the classes are fetch from the shared style sheet ***Site.css***.

It is however possible to add another CSS file to separate the CSS logic: one ticket file and one CSS file per ticket is a good way to go. To do so, the option ***CssPath*** can be added to the ticket with the path of the file to load.

When another file is added like this, the classes are fetch from this file first, and if they can't be found they are fetch from *Site.css*.

```
<Ticket>
  <Name>Warranty</Name>
  <CssPath>C:\Users\Documents\ticket_style.css</CssPath>
</Ticket>
```

Default CSS classes

Default CSS classes are applied to components and tickets. When a CSS class is given (like in the example above), the default class is overridden and the given one is used.

To disable all the default CSS classes, we can set the option ***DefaultCSS*** to false:

```
<Component>
    <Name>MyDescription</Name>
    <Type>TextBox</Type>
    <DefaultCss>false</DefaultCss>
</Component>
```

Generated CSS classes

To avoid having the CSS definition in the ticket file, it is possible to use generated classes. Those are CSS classes whose name is composed of the ticket and the component name:

```
<Component>
    <Name>MyDescription</Name>
    <Type>TextBox</Type>
    <DefaultCss>false</DefaultCss>
</Component>
```

```
.Warranty-MyDescription {
    /* Description of Css */
}

.Warranty-MyDescription-title {
    /* Description of CssTitle */
}

.Warranty-MyDescription-component {
    /* Description of CssComponent */
}
```

All useable generated classes are listed in the ***Wiki***. They are created like so:

.***TicketName-ComponentName****-CssOption*

Note that the default CSS option must be set to false to use these generated classes. It is possible to combine generated classes and regular CSS options in the ticket file: if such an option is defined, it will be used instead of the generated class:

```
<Component>
    <Name>MyDescription</Name>
    <Type>TextBox</Type>
    <DefaultCss>false</DefaultCss>
    <Css>my-class1</Css>
</Component>
```

Overriden

```
.Warranty-MyDescription {
    /* Description of Css */
}

.Warranty-MyDescription-title {
    /* Description of CssTitle */
}

.Warranty-MyDescription-component {
    /* Description of CssComponent */
}
```

It is possible to use the default CSS classes by using ***default*** in the chosen CSS option:

```
<Component>
    <Name>MyDescription</Name>
    <Type>TextBox</Type>
    <DefaultCss>false</DefaultCss>
    <Css>default</Css>
</Component>
```

All the real class names associated with *default* are listed in the ***Wiki***, so that it is possible to inherit from them in CSS files.