Chapter 1

The Settings Framework (DRAFT)

When an application gets mature it often needs to provide variations such as the default selection color, the default font and the default font size... Often such variations represent user preferences of possible software customizations. Since the 1.1 release, Pharo contains and uses the Settings framework to manage its preferences. With Setting, an application can expose its configuration. Settings is not limited to manage Pharo preferences but we suggest to use for any application. The nice thing about Settings is that it is not intrusive, it supports a modular decomposition of software and can be added to an application after its inception. This is what we will see now.

1.1 Settings in a Nutshell

Setting supports an object-oriented approach to preferences definition and manipulation. What we want to express by this sentence is that each package or subsystem should define its own customization points (often represented as a variable). The code of a subsystem then freely accesses such customization value and use it to change it behavior to reflect the preference. Then using Setting, a subsystem describe its preferences so that the end user can manipulate them. However, at not point in time, the code of a subsystem will explicit refer to setting objects to adapt its behavior. The control flow of a subsystem does not involve Setting. This is the major point of difference between Setting and the preference system available in Pharo1.0.

A preference is a particular *value* which is usually accessible. Basically such a preference value is stored in a class variable or in an instance variable of a singleton and is directly managed through the use of simple accessors.

Pharo contains numerous preferences such as the user interface theme, the desktop background color or a boolean flag to allow or prohibit the use of sound are currently declared as preferences. We will show how we can define a preference in Section **??**.

Pharo users need to browse existing preferences and eventually change their value, this is the major role of the settings browser presented in section 1.2.

A setting is a *declaration* (description) of a preference value. To be viewed and updated through the setting browser, a preference value must be described by a setting. Such a setting is built by a particular method tagged with a pragma. The section 1.3 explains how to declare a setting.

1.2 The Setting Browser

The setting browser, shown in figure Figure 1.1, mainly allows the browsing of all currently declared settings and to change related preference values. To open it, just use the World menu (World ▷ System ▷ Settings) or evaluate the following expression:

```
SettingBrowser open
```

The settings are presented in several trees in the middle panel. Setting searching and filtering is available from the top toolbar whereas the bottom panels show currently selected setting description (left bottom panel) and current package set (right bottom panel).

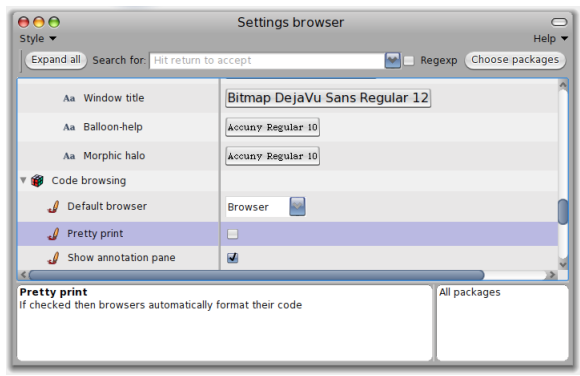

Figure 1.1: The settings browser

Browsing and changing preference values

Setting declarations are organized in trees which can be browsed in the middle panel. In order to get a description for a setting, just click on it: the setting is selected and the left bottom panel is updated with informations about the selected setting.

Changing a preference value is simply done through the browser: each line holds a widget on the right with which you can update the value. The kind of widget depends on the actual type of the preference value. Whereas a preference value can be of any kind, the setting browser is currently able to present a specific input widget for the following types: *Boolean*, *Color*, *FilePath*, *FileDirectory*, *Font*, *Number*, *Point* and *String*. A drop-list, a password field or a range input widget using a slider can also be used. Of course, the list of possible widgets is not closed as it is possible to make the setting browser support new kind of preference values or use different input widgets. This point is explained in section **??**.

If the actual type of a setting is either *String*, *FilePath*, *FileDirectory*, *Number* or *Point*, to change a value, the user has to enter some text in a editable drop-list widget. In such a case, the input must be confirmed by hitting the return key (or with cmd-s). If such a setting value is changed often, the drop-list widget is very handy because you can retrieve and use previously entered values in one click!

Other possible actions are all accessible from the contextual menu. Depending on the selected setting, they may be different. Three versions of it are shown in Figure 1.2.

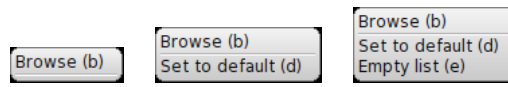

Figure 1.2: The contextual popup menu

- **Browse (b)**: opens a system browser on the method which declares the setting. It is also accessible via the keyboard shortcut *cmd-b* or if you double-click on a setting. It is very handy if you want to change the setting implementation or simply see how it is implemented to understand the framework by investigating some examples (how to declare a setting is explained in Section 1.3).
- **Set to default (d)**: set the selected setting value to the default one. It is very handy if, as an example, you have played with a setting to observe its effect and finally decide to come back to its default. It is also possible to set to default all settings is one single action, this is explained in Section **??**.

- **Empty list (e)**: If the input widget is an editable drop-list, this menu item allows one to forget previously entered values by emptying the recorded list.

Searching and filtering settings

Pharo contains a lot of settings and finding one of them can be tedious. You can filter the settings list by entering something in the search text field of the top bar of the SettingsBrowser. Then, only the settings which name or description contains the text you've entered will be shown. The text can be a regular expression if the "Regexp" checkbox is checked.

Another way to filter the list of settings is to choose them by package. Just click on the "Choose package" button, then a dialog is opened with the list of packages in which some settings are declared. If you choose one or several of them, then, only settings which are declared in the selected packages will be shown. Notice that the bottom right text pane is updated with the name of the selected packages.

Depending on where and when you are using Pharo, you may have to change preferences repeatedly. As an example, when you are doing a demonstration, you may want to have bigger fonts, at work you may need to set a proxy whereas at home none is needed. Having to change a set of preferences depending on where you are and what you are doing can be very tedious and boring. With the setting browser it is possible to save the current set of preference values in a named style that can be reloaded later. Setting style management is presented in Section **??**.

1.3 Declaring a setting

All global preferences of Pharo can be viewed or changed using the SettingBrowser. A preference is typically a class variable or an instance variable of a singleton. If one want to be able to change its value from the *SettingsBrowser*, then a setting must be declared for it. A setting is declared by a particular class method which have to be implemented as follows: it takes a builder as argument and it is tagged by the <*systemsettings*> simple pragma. The argument aBuilder serves as a facade for setting declarations building and the pragma allows the SettingBrowser to dynamically discover current setting declarations.

The important point is that a setting declaration must be package specific. It means that each package is responsible for the declaring of its own settings. For a particular package, specific settings are declared by one or several of its classes. The direct benefit is that when the package is loaded, then its settings are automatically loaded and that when a package is unloaded, then

its settings are automatically unloaded. In addition a Setting declaration should not refer to any Setting class but to the builder argument. This makes sure that your application is not dependent from Setting and that you will be able to remove Setting if you want to define extremely small footprint applications.

A simple setting

Let's take the example of the caseSensitiveFinds preference. It is a boolean preference which is used for text searching. If it is true, then a text finding is case sensitive. This preference is stored in the CaseSensitiveFinds class variable in ParagraphEditor. Its value can be queried and changed by, respectively, ParagraphEditor class>>caseSensitiveFinds and ParagraphEditor class>>caseSensitiveFinds: given below:

```
ParagraphEditor class>>caseSensitiveFinds
  ↑ CaseSensitiveFinds ifNil: [CaseSensitiveFinds := false]

ParagraphEditor class>>caseSensitiveFinds: aBoolean
  CaseSensitiveFinds := aBoolean
```

To define a setting for this preference (the CaseSensitiveFinds class variable) and be able to see it and change it from the settings browser, the method below should be implemented. The result is shown in the screenshot of the Figure 1.3.

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
  <systemsettings>
  (aBuilder setting: #caseSensitiveFinds)
    target: ParagraphEditor;
    label: 'Case sensitive search' translated;
    description: 'If true, then the "find" command in text will always make its searches in
     a case−sensitive fashion' translated;
    parent: #codeEditing.
```

Now, let's study this setting declaration with more details.

The header

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
...
```

This class method is declared in CodeHolderSystemSettings. This class is dedicated to settings and contains nothing but settings declarations. Defining such a class is not mandatory, in fact any class can have settings declarations. We define it that way to make sure that the setting declaration is

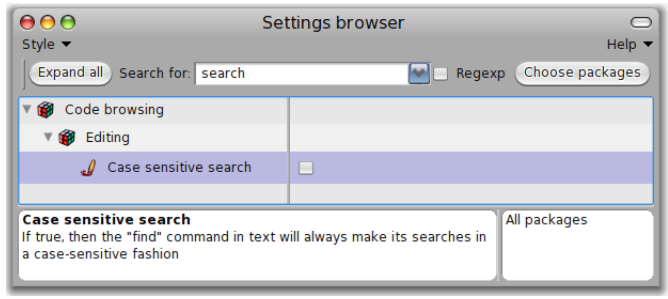

Figure 1.3: The *caseSensitiveFinds* setting

simply packaged in a different package than the one of the preference definition.

This method takes a builder as argument. This object serves as a facade for settings buildings: the contents of the method essentially consists in sending messages to the builder to declare and organize a sub-tree of settings.

The pragma

A setting declaration is tagged with the <systemsettings> pragma.

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
   <systemsettings>
...
```

In fact, when the settings browser is opened, it first collects all settings declarations by searching all methods with the <systemsettings> pragma. More, if you compile a setting declaration method while a settings browser is opened then it is automatically updated with the new setting.

The setting building

A setting is simply declared by sending the message setting: to the builder with an identifier passed as argument:

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
   <systemsettings>
   (aBuilder setting: #caseSensitiveFinds)
...
```

By default, the identifier passed as argument is considered as the selector which can be used by the settings browser to get the preference value and

the selector for changing the preference value is by default built by adding a colon to the getter selector (*i.e.*, caseSensitiveFinds: here).

These selectors are sent to a target which is by default the class in which the method is implemented (*i.e.*, CodeHolderSystemSettings). Thus, this one line setting declaration would be sufficient if caseSensitiveFinds and caseSensitiveFinds: accessors were implemented in CodeHolderSystemSettings. Here, instead, these accessors are implemented in ParagraphEditor. Then, it must be explicitly set that the target is ParagraphEditor as done below:

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
  <systemsettings>
  (aBuilder setting: #caseSensitiveFinds)
    target: ParagraphEditor
```

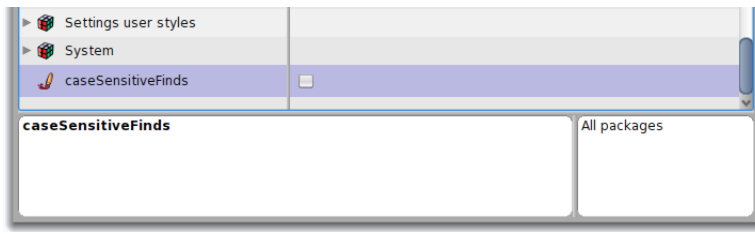

Figure 1.4: the very simple version of the *caseSensitiveFinds* setting

This very short version is enough to be compiled and taken into account by the settings browser as shown by Figure 1.4. Unfortunately, the presentation is not user-friendly because:

- the label shown in the settings browser is its identifier,
- there is no description or explanation available for this setting and,
- the new setting is simply added at the root of the setting tree.

You can indicate a label and a description with respectively the #label: and #description: messages which take a string as argument. Don't forget to send #translated to the label and the description strings, it will greatly facilitate the translation in other languages.

Concerning the classification and the settings tree organisation, there are several ways to improve it and this point is fully detailed in the next section.

Organising your settings

Within the settings browser, settings are organised in trees where related settings are shown as children of the same parent.

Declaring a parent

The simplest way to declare your setting as a child of another setting is to use the #parent: message with the identifier of the parent setting passed as argument. In the example below, the parent node is an existing node declared with the #codeEditing identifier.

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
    <systemsettings>
    (aBuilder setting: #caseSensitiveFinds)
        target: ParagraphEditor;
        label: 'Case sensitive search' translated;
        description: 'If true, then the "find" command in text will always make its searches in
         a case−sensitive fashion' translated;
        parent: #codeEditing.
```

Then, it means that the #codeEditing node is also declared somewhere in the system. As an example, it could be declared as follow:

```
CodeHolderSystemSettings class>>codeEditingSettingsOn: aBuilder
    <systemsettings>
    (aBuilder group: #codeEditing)
        label: 'Editing' translated;
        parent: #codeBrowsing.
```

Notice that the #codeEditing node is created by sending the #group: message to the builder with its identifier passed as argument. A group is a simple node without any value and which is only used for children grouping. Notice also that, as shown in the Figure 1.3, the #codeEditing node is not at root because it is declared itself as a child of the #codeBrowsing node.

Declaring a sub-tree

Being able to declare its own settings as a child of a pre-existing node is very useful when a package wants to enrich existing standard settings. But it can also be very tedious for settings which are very application specific.

Thus, directly declaring a sub-tree of settings in one method is also possible. Then, typically, a root group is declared for the application settings and the children settings themselves are also declared within the same method. This is simply done through the sending of the with: message to the root group. The with: message takes a block as argument. In this block, every new settings are implicitely declared as children of the root group (the receiver of the with: message).

As an example, take a look at figure Figure 1.5, it shows the settings for the refactoring browser configurable formatter. This sub-tree of settings is fully declared in the method RBConfigurableFormatter class>>settingsOn: given

Figure 1.5: Declaring a subtree in one method: the *Configurable formatter* setting example

below. You can see that it declares the new root group #configurableFormatter with two children, #formatCommentWithStatements and #indentString:

```
RBConfigurableFormatter class>>settingsOn: aBuilder
   <systemsettings>
   (aBuilder group: #configurableFormatter)
      target: self;
      parent: #refactoring;
      label: 'Configurable Formatter' translated;
      description: 'Settings related to the formatter' translated;
      with: [
         (aBuilder setting: #formatCommentWithStatements)
            label: 'Format comment with statements' translated.
         (aBuilder setting: #indentString)
            label: 'Indent string' translated]
```

Optional sub-tree

Sometime, depending on the value of a particular preference, one might want to hide some settings because it doesn't make sense to show them. As an example, if the background color of the desktop is plain then it doesn't make sense to show settings which are related to gradient background. Instead, if the user wants a gradient background, then a second color, the gradient direction and the gradient origin settings should be presented. Look at the Figure 1.6:

- on the left, the *Gradient* widget is unchecked meaning that its actual value is false; in this case, it has no children,
- on the right, the *Gradient* widget is checked, then the setting value is set to true and as a consequence, the settings useful in order to set a gradient background are shown.

In order to handle such optional settings the only thing is to declare them as children of a boolean parent setting. In this case, children settings are shown only if the parent value is true. Concerning the desktop gradient example, the setting is declared in PolymorphSystemSettings as given below:

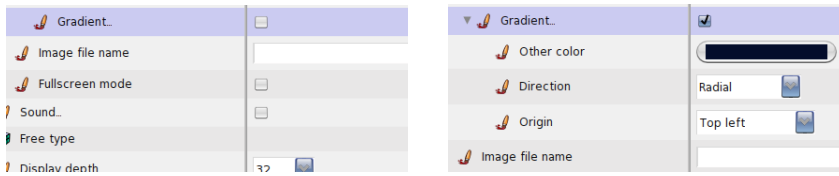

Figure 1.6: Example of optional subtree

```
(aBuilder setting: #useDesktopGradientFill)
    label: 'Gradient';
    description: 'If true, then more settings will be available in order to define the
     desktop background color gradient';
    with: [
        (aBuilder setting: #desktopGradientFillColor)
            label: 'Other color';
            description: 'This is the second color of your gradient (the first one is given by
     the "Color" setting' translated.
        (aBuilder pickOne: #desktopGradientDirection)
            label: 'Direction';
            domainValues: {#Horizontal. #Vertical. #Radial}.
        (aBuilder pickOne: #desktopGradientOrigin)
            label: 'Origin';
            domainValues: {
                'Top left' translated -> #topLeft. ...
```

The parent setting value is given by evaluating PolymorphSystemSettings class>>#useDesktopGradientFill. If it returns true, then the children #desktopGradientFillColor, #desktopGradientDirection and #desktopGradientOrigin are shown.

Restricting the value domain

By default, the possible value set is not restricted and is given by the actual type of the preference. As examples, for a color preference, the widget allows you to choose whatever color, for a number, the widget allows the user to enter whatever number. But, in some cases, only a particular set of values is desired. As an example, for the standard browser or for the user interface theme settings, the choice must be made among a finite set of classes, for the free type cache size, only a range from 0 to 50000 is allowed. In these cases, it is much more confortable if the widget can allow only particular values. So far, the domain value set can be constrained either with a range or with a list of values.

Declaring a range setting

As an example, let's consider the full screen margin preference shown in the Figure 1.7. Its value constitutes the margin size in pixels that is let around a window when it is expanded.

Figure 1.7: Example of range setting

It's value is an integer but it makes no sense to set -100 or 5000 to it. Instead, a minimum of -5 and a maximum of 100 constitute a good range of values. One can use this range in order to constraint the setting widget. As shown by the example below, comparing to a simple setting, the only two differences are that:

- the new setting is created with the #range: message instead of the #setting: message and
- the valid range is given by sending the #range: message to the newly declared setting, an Interval is given as argument.

```
screenMarginSettingOn: aBuilder
    <systemsettings>
    (aBuilder range: #fullScreenMargin)
        target: SystemWindow;
        parent: #windows;
        label: 'Full screen margin' translated;
        description: 'Specify the amount of space that is let around a windows when it''s
         opened fullscreen' translated;
        range: (−5 to: 100).
```

Selecting among a list

When a preference value is constrained to be one of a particular list of values, it is possible to declare it so that a drop list is used by the settings browser. This drop list is initialized with the predefined valid values. As an example, consider the *window position strategy* example. The corresponding widget is shown in action within the settings browser by Figure 1.8. The allowed values are 'Reverse Stragger', 'Cascade' and 'Standard'. The example below shows a simplified declaration for the *window position strategy* setting.

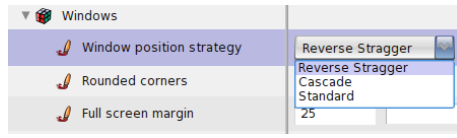

Figure 1.8: Example of a list setting

```
windowPositionStrategySettingsOn: aBuilder
    <systemsettings>
    (aBuilder pickOne: #usedStrategy)
        label: 'Window position strategy' translated;
        target: RealEstateAgent;
        domainValues: #(#'Reverse Stragger' #Cascade #Standard)
```

comparing to a simple setting, the only two differences are that:

- the new setting is created with the #pickOne: message instead of the #setting: message and
- the list of authorized values is given by sending the #domainValues: message to the newly declared setting, a Collection is given as argument (the default value being the first one).

Concerning this window strategy example, the value set to the preference would be either #'Reverse Stragger' or #Cascade or #Standard.

Unfortunately, these values are not very handy. A programmer may wish another value as, for example, some kind of *strategy object* or a Symbol which could directly serve as a selector. In fact, this second solution has been chosen by the RealEstateAgent class maintainers. If you inspect the value returned by RealEstateAgent usedStrategy you will realise that the result is not a Symbol among #(#'Reverse Stragger' #Cascade #Standard) but another symbol. Then, if you look at the way the window position strategy setting is really implemented you will see that the declaration differs from the basic solution given previously: the *domainValues:* argument is not a simple array of Symbols but an array of Associations a you can see in the declaration below:

```
windowPositionStrategySettingsOn: aBuilder
    <systemsettings>
    (aBuilder pickOne: #usedStrategy)
    ...
    domainValues: {'Reverse Stragger' translated −> #straggerFor:initialExtent:world:. '
        Cascade' translated −> #cascadeFor:initialExtent:world:. 'Standard' translated −>
        #standardFor:initialExtent:world:};
```

From the Settings Browser point of view, the content of the list is exactly the same and the user can't notice any difference because, if an array of Associations is given as argument to domainValues:, then the keys of the Associations are used for the user interface.

Concerning the value of the preference itself, if you inspect RealEstateAgent usedStrategy, you should notice that the result is a value among #(#straggerFor:initialExtent:world: #cascadeFor:initialExtent:world: #standardFor:initialExtent:world:). In fact, the values of the Associations are used to compute all possible real values for the setting.

The list of possible values can be of any kind. As another example, let's take a look at the way the user interface theme setting is declared in the PolymorphSystemSettings class:

```
(aBuilder pickOne: #uiThemeClass)
   label: 'User interface theme' translated;
   target: self;
   domainValues: (UITheme allThemeClasses collect: [:c | c themeName -> c])
```

In this example, #domainValues: takes an array of associations which is computed each time a Settings Browser is opened. Each association is made of the name of the theme for the key and of the class which implements the theme for the value.

1.4 Design of the Settings Framework

The design of the Settings framework is based on the three following points: (1) a preference is not defined in a global class but local to the package that uses it, (2) settings can be declared independently from the application they refer to, (3) any setting declaration can be loaded even if the Settings framework is not loaded.

Let's explain now these two points since they have an impact on the modular structure of Pharo.

Local value with a local flow. The Settings framework supports the idea that a preference value is local to a package. A package should define either a singleton or a class variable defined somewhere on a class. The methods of the class are able to refer to the variable. The package should provide some way to get and set the value of the preference.

Understanding the difference to previous design as implemented in Squeak3.9 or Pharo1.0 is important. In previous versions, the class Preferences was the place where the preferences as well as methods to change their values were defined. This implies that code using preferences was ref-

erencing the class Preferences during its execution. The flow of control was clearly not local the class using the preferences but always executing some methods on the Preferences class. This design led to a system with a lot of hidden dependencies.

No explicit dependency on the Setting Framework. Finally when declaring a setting, the code does not refer explicitly to any Setting class. This has the good property that you can load code containing setting declaration even if the Settings framework is not loaded. This way we make sure that we get a modular system. In case the settings framework is not loaded, the method containing the setting declaration is just not used by the system.