Chapter 1

The Settings Framework

with the participation of:
Alain Plantec *(alain.plantec@univ-brest.fr)*

As an application matures it often needs to provide variations such as a default selection color, a default font or a default font size. Often such variations represent user preferences for possible software customizations. Since its 1.1 release, Pharo has contained and used the Settings framework to manage its preferences. With Settings, an application can expose its configuration. Settings is not limited to managing Pharo preferences and we suggest using it for any application. What is nice about Settings is that it is not intrusive, it supports modular decomposition of software and it can be added to an application even after that application's inception. The Settings framework is what we will look at now.

1.1 Settings architecture

Setting supports an object-oriented approach to preference definition and manipulation. What we want to express by this sentence is that:

1. each package or subsystem should define its own customization points (often represented as a variable or a class variable). The code of a subsystem then freely accesses such a customization value and uses it to change its behaviour to reflect the preference.

2. a subsystem describes its preferences so that the end-user can manipulate them. However, at not point in time, will the code of a subsystem explicitly refer to setting objects to adapt its behaviour.

The control flow of a subsystem does not involve Settings. This is the major point of difference between Settings and the preference system available in Pharo1.0.

Vocabulary

A *preference* is a particular *value* which is managed as a variable value. 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. We will show how we can define a preference in Section 1.3.

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 specific pragma. This specific pragma serves as a classification tag which is used to automatically identify the method as a setting (see Figure 1.1). Section 1.3 explains how to declare a setting.

Pharo users need to browse existing preferences and eventually change their value through a dedicated user interface. This is the major role of the *Settings Browser* presented in Section 1.2.

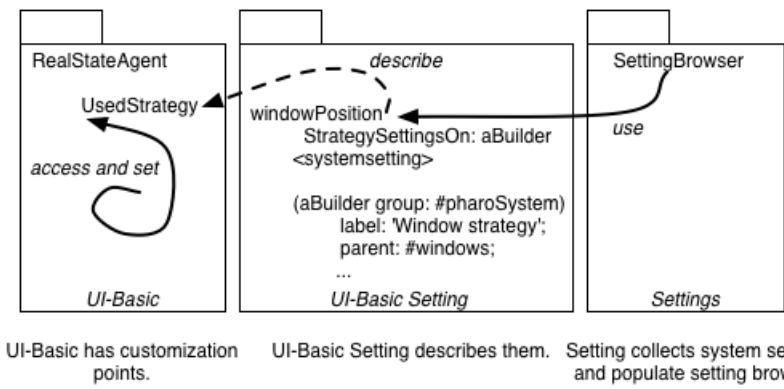

Figure 1.1: A package declares and uses customization points as variables. As an example, *UsedStrategy* is declared as a class variable of RealEstateAgent. Such customization points are described with Setting instances that are created by the automatic running of setting declaration methods. The *Settings Browser* collects the setting instances and presents them to the user.

Figure 1.1 shows important points of the architecture put in place by Settings: The *Settings* package can be unloaded and a package defining preferences does not depend on the *Settings* package. This architecture is supported by the following points:

Customization points. Each application customization points should be defined. In Figure 1.1, the class RealStateAgent of the package *UI-Basic* defines the class variable UsedStrategy which defines where the windows appear. The flow of the package *UI-Basic* is modular and self-contained: the class RealStateAgent does not depend on the settings framework. The class RealStateAgent has been designed to be parametrized.

Description of customization point. The Settings framework supports the description of the setting UsedStrategy. In Figure 1.1, the package *UI-Basic Setting* defines a method (it could be an extension to the class RealStateAgent or another class. The important point is that the method declaring the setting does not refer directly to Setting classes but describes the setting using a builder. This way the description could even be present in the *UI-Basic* package without introducing a reference.

Collecting setting for user presentation. The Settings package defines tools to manage settings such as a *Settings Browser* that the user opens to change her/his preferences. The *Settings Browser* collects settings and uses their description to change the value of preferences. The control flow of the program and the dependencies are always from the package Settings to the package that has preferences and not the inverse.

1.2 The Settings Browser

The *Settings Browser*, shown in Figure 1.2, mainly allows one to browse all currently declared settings and to change related preference values.

 *To open the Settings Browser, 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 tool-bar whereas the bottom panels show currently selected setting descriptions (left bottom panel) and current package set (right bottom panel).

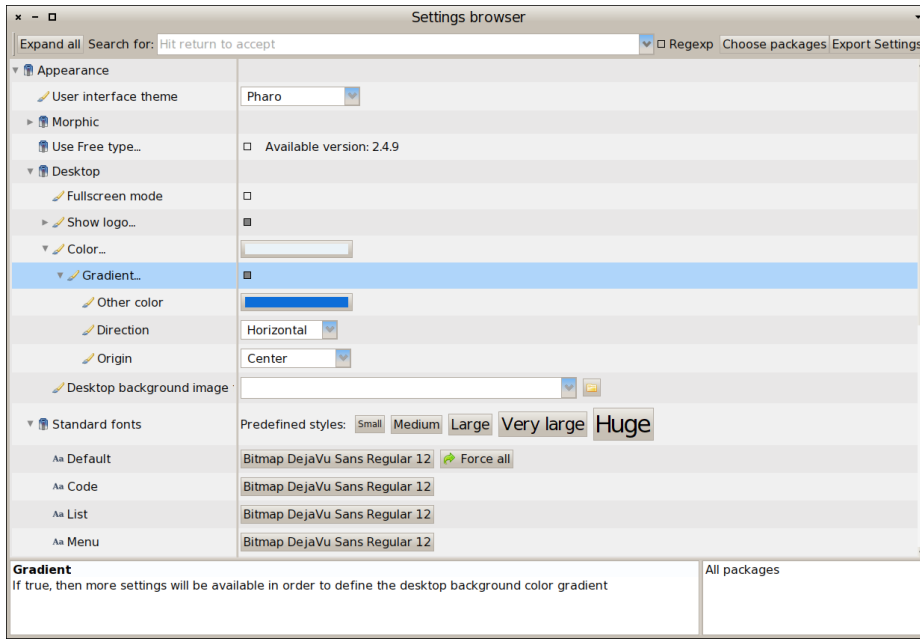

Figure 1.2: The *Settings Browser*.

Browsing and changing preference values

Setting declarations are organized in trees which can be browsed in the middle panel. To get a description for a setting, just click on it: the setting is selected and the left bottom panel is updated with information 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*, *FileName*, *DirectoryName*, *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 kinds of preference values or use different input widgets. This point is explained in Section 1.8.

If the actual type of a setting is either *String*, *FileName*, *DirectoryName*,*Number* or *Point*, to change a value, the user has to enter some text in an 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 comes in handy because you can retrieve and use previously entered values in one click! Moreover, in case of a *FileName* or a *DirectoryName*, a button is added to open a file name or a directory name chooser dialog.

Other possible actions are all accessible from the contextual menu. Depending on the selected setting, they may be different. The two possible versions are shown in Figure 1.3.

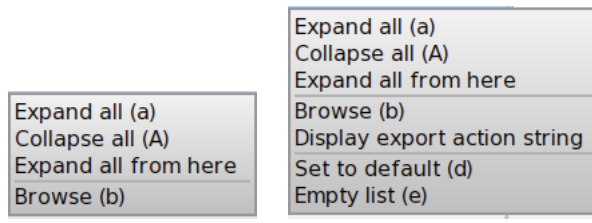

Figure 1.3: The contextual popup menu

- **Expand all (a)**: expand all the setting tree nodes recursively. It is also accessible via the keyboard shortcut *cmd-a*.
- **Collapse all (a)**: collapse all the setting tree nodes recursively. It is also accessible via the keyboard shortcut *cmd-A*.
- **Expand all from here**: Expand the currently selected setting tree node recursively.
- **Browse (b)**: open a system browser on the method that 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).
- **Display export action string**: a setting can be exported as a start-up action, this menu option allow to display how the start-up action is coded (Start-up action management is explained in Section 1.7).
- **Set to default (d)**: set the selected setting value to the default one. It is useful if, as an example, you have played with a setting to observe its effect and finally decide to come back to its default.
- **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 whose name or description contains the text you have 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, only settings which are declared in the selected packages are 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 repeately. 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 *Settings 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 *Settings Browser*. A preference is typically a class variable or an instance variable of a singleton. If one wants to be able to change a value from the *SettingsBrowser*, then a setting must be declared for it. A setting is declared by a particular *class* method that should be implemented as follows: it takes a builder as argument and it is tagged with the <*systemsettings*> pragma.

The argument, aBuilder, serves as an API or facade for building setting declarations. The pragma allows the *Settings Browser* to dynamically discover current setting declarations.

The important point is that a setting declaration should 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 or a companion package. There is no global setting defining class or package (as was the case in Pharo1.0). The direct benefit is that when the package is loaded, then its settings are automatically loaded. 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 assures that your application is not dependent

on Settings and that you will be able to remove Setting if you want to define extremely small footprint applications.

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 text finding is case sensitive. This preference is stored in the CaseSensitiveFinds class variable of the class TextEditor. Its value can be queried and changed by, respectively, TextEditor class>>caseSensitiveFinds and TextEditor class>>caseSensitiveFinds: given below:

```
TextEditor class>>caseSensitiveFinds
    ^ CaseSensitiveFinds ifNil: [CaseSensitiveFinds := false]

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

To define a setting for this preference (*i.e.*, for the CaseSensitiveFinds class variable) and be able to see it and change it from the *Settings Browser*, the method below is implemented. The result is shown in the screenshot of the Figure 1.4.

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

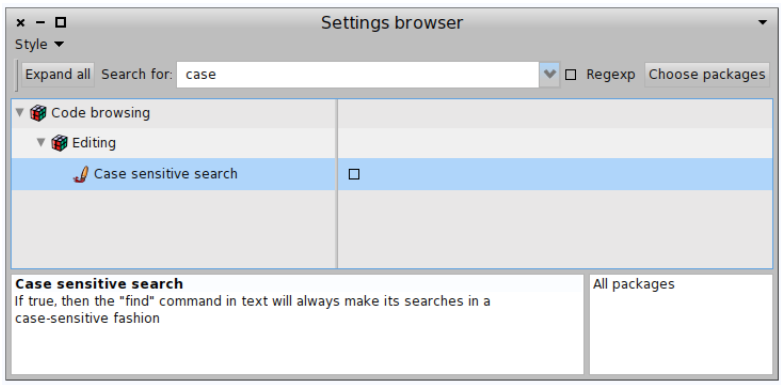

Figure 1.4: The *caseSensitiveFinds* setting

Now, let's study this setting declaration in details.

The header

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

This class method is declared in the class CodeHolderSystemSettings. This class is dedicated to settings and contains nothing but setting declarations. Defining such a class is not mandatory; in fact, any class can define setting declarations. We define it that way to make sure that the setting declaration is packaged in a different package than the one of the preference definition – for layering purposes.

This method takes a builder as argument. This object serves as a facade API for setting 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. In addition, if you compile a setting declaration method while a *Settings Browser* is opened then it is automatically updated with the new setting.

The setting configuration

A setting is declared by sending the message setting: to the builder with an identifier passed as argument. Here is an example where the identifier is #caseSensitiveFinds:

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

Sending the message setting: to a builder creates a *setting node builder* which itself is a wrapper for a setting node. By default, the symbol passed as

argument is considered as the selector used by the *Settings Browser* to get the preference value. The selector for changing the preference value is by default built by adding a colon to the getter selector (*i.e.*, it is 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 is sufficient if caseSensitiveFinds and caseSensitiveFinds: accessors are implemented in CodeHolderSystemSettings.

In fact, very often, these default initializations will not fit your need. Of course you can adapt the setting node configuration to take into account your specific situation. For example, the corresponding getter and setter accessors for the caseSensitiveFinds setting are implemented in the class TextEditor. Then, we should explicitly set that the target is TextEditor. This is done by sending the message target: to the setting node with the target class TextEditor passed as argument as shown by the updated definition:

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

This very short version is fully functional and enough to be compiled and taken into account by the *Settings Browser* as shown by Figure 1.5.

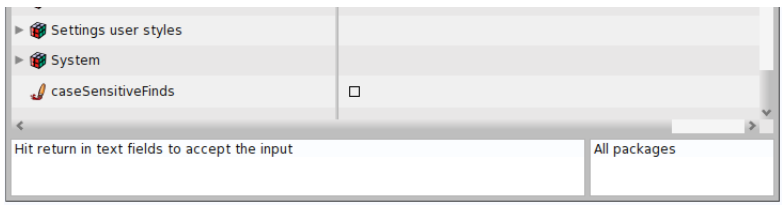

Figure 1.5: A first simple version of the caseSensitiveFinds setting.

Unfortunately, the presentation is not really user-friendly because:

- the label shown in the settings browser is the identifier (the symbol used to build accessors to access it),
- there is no description or explanation available for this setting, and
- the new setting is simply added at the root of the setting tree.

To address such shortcomings, you can better configure your setting node with a label and a description respectively with the label: and description: messages which take a string as argument.

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

Don't forget to send translated to the label and the description strings, it will greatly facilitate the translation into other languages.

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

More about the target

The target of a setting is the receiver for getting and changing the preference value. Most of the time it is a class. Typically, a preference value is stored in a class variable. Thus, class side methods are used as accessors for accessing the setting.

But the receiver can also be a singleton object. This is currently the case for many preferences. As an example, the Free Type fonts preferences, they are all stored in the instance variables of a FreeTypeSettings singleton. Thus, here, the receiver is the FreeTypeSettings instance that you can get by evaluating the following expression:

```
FreeTypeSettings current
```

One can use this expression to configure the target of a corresponding setting. As an example the #glyphContrast preference could be declared as follow:

```
(aBuilder setting: #glyphContrast)
    target: FreeTypeSettings current;
    label: 'Glyph contrast' translated;
    ...
```

This is simple, but unfortunately, declaring such a singleton target like this is not a good idea. This declaration is not compatible with the *Setting style* functionalities (see Section **??**). In such a case, one would have to separately indicate the target class and the message selector to send to the target class to get the singleton. Thus, as shown in the example below, you should use the targetSelector: message:

```
(aBuilder setting: #glyphContrast)
    target: FreeTypeSettings;
```

```
  targetSelector: #current;
  label: 'Glyph contrast' translated;
  ...
```

More about default values

The way the *Settings Browser* builds a setting input widget depends on the actual value type of a preference. Having nil as a value for a preference is a problem for the *Settings Browser* because it can't figure out which input widget to use. So basically, to be properly shown with the good input widget, a preference must always be set with a non nil value. You can set a default value to a preference by initializing it as usual, with a #initialize method or with a lazy initialization programed in the accessor method of the preference.

Regarding the *Settings Browser*, the best way is the lazy initialization (see the example of the #caseSensitiveFinds preference given in Section 1.3). Indeed, as explained in Section 1.2, from the *Settings Browser* contextual menu, you can reset a preference value to its default or globally reset all preference values. In fact, it is done by setting the preference value to reset to nil. As a consequence, the preference is automatically set to its default value as soon as it is get by using its dedicated accessor.

It is not always possible to change the way an accessor is implemented. A reason for that could be that the preference accessor is maintained within another package which you are not allowed to change. As shown in the example below, as a workaround, you can indicate a default value from the declaration of the setting by sending the message default: to the setting node:

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

1.4 Organizing your settings

Within the *Settings Browser*, settings are organized 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: TextEditor;
        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.
```

The #codeEditing node is also declared somewhere in the system. For example, it could be defined as a group as we will see now.

Declaring a group

A group is a simple node without any value and which is only used for children grouping. The node identified by #codeEditing is created by sending the group: message to the builder with its identifier passed as argument. Notice also that, as shown in Figure 1.4, the #codeEditing node is not at root because it has declared itself as a child of the #codeBrowsing node.

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

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. 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, all new settings are implicitly declared as children of the root group (the receiver of the with: message).

Figure 1.6: Declaring a subtree in one method: the *Configurable formatter* setting example.

As an example, take a look at Figure 1.6, 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 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

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 the gradient background. Instead, when 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 1.7:

- 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 to set a gradient background are shown.

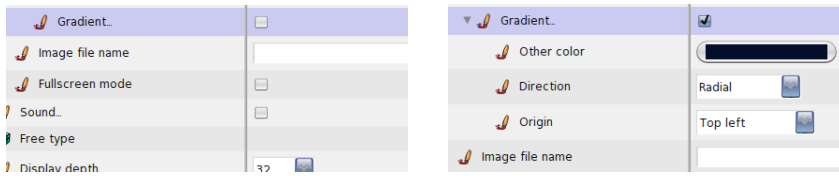

Figure 1.7: Example of optional subtree. Right – no gradient is selected. Left – gradient is selected so additional preferences are available.

To handle such optional settings is simple: optional settings should be declared 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:

```
(aBuilder setting: #useDesktopGradientFill)
    label: 'Gradient';
    description: 'If true, then more settings will be available 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.

Ordering your settings

By default, sibling settings are sorted alphabetically by their label. You may want to change this default behavior. Changing the settings ordering can be done two ways: by simply forbidding the default ordering or by explicitly specifying an order.

As in the following example of the #appearance group, you can indicate that no ordering should be performed by sending the noOrdering message to the parent node.

```
appearanceSettingsOn: aBuilder
    <systemsettings>
    (aBuilder group: #appearance)
        label: 'Appearance' translated;
        description: 'All settings concerned with the look"n feel of your system' translated;
        noOrdering;
        with: [... ]
```

You can indicate the order of a setting node among its siblings by sending the message order: to it with a number passed as argument. The number can be an Integer or a Float. Nodes with an order number are always placed before others and are sorted according to their respective order number. If an order is given to an item, then no ordering is applied for other siblings.

As an example, take a look at how the #standardFonts group is declared:

```
(aBuilder group: #standardFonts)
    label: 'Standard fonts' translated;
    target: StandardFonts;
    parent: #appearance;
    with: [
        (aBuilder launcher: #updateFromSystem)
            order: 1;
            targetSelector: #current;
            script: #updateFromSystem;
            label: 'Update fonts from system' translated.
        (aBuilder setting: #defaultFont)
            label: 'Default' translated.
        (aBuilder setting: #codeFont)
            label: 'Code' translated.
        (aBuilder setting: #listFont)
    ...
```

In this example, the launcher #updateFromSystem is declared to be the first node, then other siblings with identifiers #defaultFont, #codeFont, and #listFont are placed according to the declaration order.

1.5 Providing more precise value domain

By default, the possible value set of a preference is not restricted and is given by the actual type of the preference. For example, for a color preference, the widget allows you to choose whatever color. For a number, the widget allows the user to enter any 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 50,000 is

allowed. In these cases, it is much more comfortable if the widget can only accept particular values. To address this issue, 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 1.8. Its value represents the margin size in pixels that is left around a window when it is expanded.

Figure 1.8: Example of range setting.

Its 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 to constrain the setting widget. As shown by the example below, comparing it to a simple setting, the only two differences are that:

- the new setting node is created with the range: message instead of the setting: message and
- the valid range is given by sending the range: message to the setting node, 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 left 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 1.9. The allowed values are 'Reverse Stagger', 'Cascade', or 'Standard'.

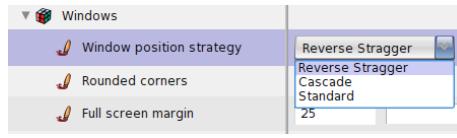

Figure 1.9: Example of a list setting.

The example below shows a simplified declaration for the *window position strategy* setting.

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

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

- the new setting node 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 node, 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 Stagger' or #Cascade or #Standard.

Unfortunately, these values are not very handy. A programmer may expect another value. 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 realize that the result is not a Symbol among #'Reverse Stagger', #Cascade, or #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 as you can see in the declaration below:

```
windowPositionStrategySettingsOn: aBuilder
    <systemsettings>
    (aBuilder pickOne: #usedStrategy)
```

```
...
domainValues: {'Reverse Stagger' translated −> #staggerFor: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 not 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 #staggerFor:initialExtent:world:, #cascadeFor:initialExtent:world: and #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 as key and of the class which implements the theme as value.

1.6 Launching a script

Imagine that you want to launch an external configuration tool or that you want to allow one to configure the system or a particular package with the help of a script. In such a case you can declare a *launcher*. A launcher is shown with a label as a regular setting except that no value is to be entered for it. Instead, a button labelled *Launch* is integrated in the *Settings Browser* and clicking on it launch an associated script.

As an example, to use True Type Fonts, the system must be updated by collecting all the available fonts in the host system. This can be done by evaluating the following expression:

```
FreeTypeFontProvider current updateFromSystem
```

It is possible to run this script from the *Settings Browser*. The corresponding launcher is shown in Figure 1.10. The integration of such a launcher is quite

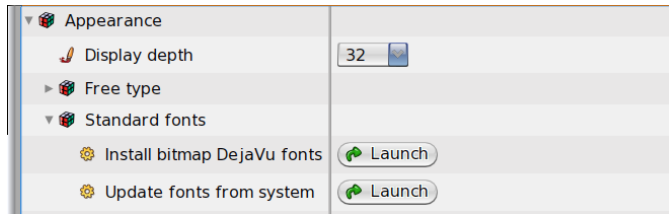

Figure 1.10: Example of launchers.

simple. You simply have to declare a setting for it! For example, look at how the launcher for the TT fonts is declared:

```
GraphicFontSettings class>> standardFontsSettingsOn:
   <systemsettings>
   (aBuilder group: #standardFonts)
      ...
      (aBuilder launcher: #updateFromSystem) ...
            target: FreeTypeFontProvider;
            targetSelector: #current;
            script: #updateFromSystem;
            label: 'Update fonts from system' translated.
```

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

- the new setting node is created by sending the launcher: message to the builder and
- the message script: is sent to the setting node with the selector of the script passed as argument.

1.7 Start-up actions management

Even if many preferences have been removed from Pharo because they were obsolete, there are a still a large number of them. And even if the *Settings Browser* is easy to use, it may be tedious to set up your own preferences even for a subset, each time you start working with a new image. A solution is to implement a script to set all your preferred choices. The best way is to create a specific class for that purpose. You can then include it in a package that you can reload each time you want to setup a fresh image. We call this kind of class a *Setting style*.

To manage *Setting styles*, the *Settings Browser* can be helpful in two ways. First, it can help you discover how to change a preference value, and second, it can create and update a particular style for you.

Scripting settings

Because preference variables are all accessible with accessor methods, it is naturally possible to initialize a set of preferences in a simple script. For the sake of simplicity, let's implement it in a Setting style.

As an example, a script can be implemented to change the background color and to set all fonts to a bigger one than the default. Let's create a Setting style class for that. We can call it MyPreferredStyle. The script is defined by a method of MyPreferredStyle. We call this method loadStyle because this selector is the standard hook for settings related script evaluating.

```
MyPreferredStyle>>loadStyle
   | f n |
   "Desktop color"
   PolymorphSystemSettings desktopColor: Color white.
   "Bigger font"
   n := StandardFonts defaultFont. "get the current default font"
   f := LogicalFontfamilyName: n familyName pointSize: 12. "font for my preferred size"
   StandardFonts setAllStandardFontsTo: f "reset all fonts"
```

PolymorphSystemSettings is the class in which all settings related to *PolyMorph* are declared. StandardFonts is the class that is used to manage Pharo default fonts.

Now, the question is if the desktop color setting is declared in PolymorphSystemSettings and that the DefaultFonts class allows fonts management? Where are all these settings declared and managed in general?

The answer is quite simple: just use the *Settings Browser*! As explained in Section 1.2, *cmd-b* or double clicking on an item open a browser on the declaration of the current setting node. You can also use the contextual menu for that. Browsing the declaration will give you the target class (where the preference variable is stored) and the selector for the preference value.

Now we would like MyPreferredStyle>>#loadStyle to be automatically evaluated when MyPreferredStyle is itself loaded in the system. For that purpose, the only thing to do is to implement an initialize method for the MyPreferredStyle class:

```
MyPreferredStyle class>>initialize
   self new loadStyle
```

Integrating a style in the *Settings Browser*

Any script can be integrated in the *Settings Browser* so that it could be loaded, browsed or even removed from it. For that purpose you only have to declare a name for it and to make sure that the *Settings Browser* will discover it. Just

implement a method named styleName on the class side of your style class. Concerning the example of previous section, it should be implemented as follows:

```
MyPreferredStyle class>>styleName
    "The style name used by the SettingBrowser"
    <settingstyle>
    ^ 'My preferred style'
```

MyPreferredStyle class>>styleName takes no argument and must return the name of your style as a *String*. The <*settingstyle*> pragma is used to let the *Settings Browser* know that MyPreferredStyle is a setting style class.

Once this method is compiled, open the Setting Browser and popup the *Style* top menu. As shown by Figure 1.11, you should see a dialog with a list of style names comprising your own one.

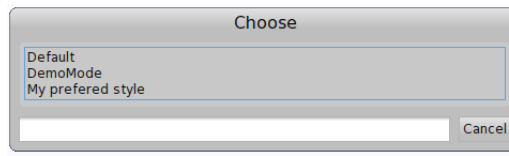

Figure 1.11: The dialog for loading style with your own style

1.8 Extending the Settings Browser

As explained in the section 1.2, the *Settings Browser* is by default able to manage simple preference types. These default possibilities are generally enough. But there are some situations where it can be very helpful to be able to handle more complex preference values.

As an example, let us focus on the text selection preferences. We have the primary selection and three other optional kinds of text selection, the secondary selection, the 'find and replace' selection and the selection bar. For all selections, a background color can be set. For the primary, the secondary and the find and replace selection, a text color can also be chosen.

Declaring selection settings individually

So far, according to the default possibilities, a setting can be declared for each of the text selection characteristics so that each corresponding preference can be changed individually from the *Settings Browser*. Settings declared

for a particular selection kind can be grouped together as children of a setting group. As an immediate improvement, for an optional text selection, a boolean setting can be used instead of a simple group.

As an example, let's take the secondary selection. This text selection kind is optional and one can set a background and a text color for it. Corresponding preferences are declared as instance variables of ThemeSettings. Their values can be read and changed from the current theme by getting its associated ThemeSettings instance. Thus, the two color settings can be declared as children of the #useSecondarySelection boolean setting as given below:

```
(aBuilder setting: #useSecondarySelection)
   target: UITheme;
   targetSelector: #currentSettings;
   label: 'Use the secondary selection' translated;
   with: [
      (aBuilder setting: #secondarySelectionColor)
         label: 'Secondary selection color' translated.
      (aBuilder setting: #secondarySelectionTextColor)
         label: 'Secondary selection text color' translated].
```

The Figure 1.12 shows these setting declarations in the *Settings Browser*. The look and feel is clean but in fact two observations can be made:

1. it takes three lines for each selection kind. This is a little bit uncomfortable because the view for one selection takes a lot of vertical space,
2. the underlying model is not explicitly designed. The settings for one selection kind are grouped together in the *Settings Browser*, but corresponding preference values are declared as separated instance variables of ThemeSettings. In the next section we see how to improve this first solution with a better design.

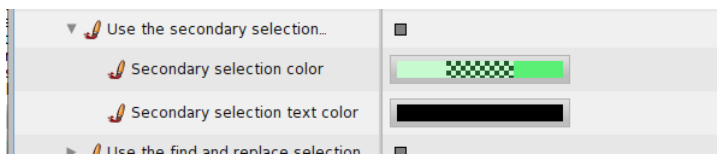

Figure 1.12: The secondary selection settings declared with basic setting values

An improved selection preference design

A better solution would be to design the concept of text selection preference. Then, we have only one value to manage for each selection preference in-

stead of three. A text selection preference is basically made of two colors, one for the background and the second for the text. Except the primary selection, each selection is optional. Then, we could design a text selection preference as follow:

```
Object subclass: #TextSelectionPreference
  instanceVariableNames: 'backgroundColor textColor mandatory used'
  classVariableNames: 'FindReplaceSelection PrimarySelection SecondarySelection
    SelectionBar'
  poolDictionaries: ''
  category: 'Settings-Tools'
```

TextSelectionPreference is made of four instance variables. Two of them are the colors. If the mandatory instance variable is set to false, then the used boolean instance variable can be changed. Instead, if the mandatory is set to true, then the used instance variable is set to true and is not changeable.

TextSelectionPreference has also four class variables, one for each kind of possible text selection preference. The getters and setters have also to be implemented i to be able to manage these preferences from the *Settings Browser*. As an example, for PrimarySelection:

```
TextSelectionPreference class>>primarySelection
  ^ PrimarySelection
    ifNil: [PrimarySelection := self new
              textColor: Color black;
              backgroundColor: (Color blue alpha: 0.5);
              mandatory: true;
              yourself]
```

You can notice that the mandatory attribute is initialized to true.

Another example with the selection bar preference:

```
TextSelectionPreference class>>selectionBar
  ^ SelectionBar
    ifNil: [SelectionBar := self new
            backgroundColor: Color lightBlue veryMuchLighter;
            mandatory: false;
            yourself]
```

Here, you can notice that the preference is declared as optional and with no text color.

For these preferences to be changeable from the *Settings Browser*, we have to declare two methods. The first one is for the setting declaration and the second is to implement the view.

The setting declaration is implemented as follow:

```
TextSelectionPreference class>>selectionPreferenceOn: aBuilder
```

```
<systemsettings>
(aBuilder group: #selectionColors)
    label: 'Text selection colors' translated;
    parent: #appearance;
    target: self;
    with: [(aBuilder setting: #primarySelection) order: 1;
            label: 'Primary'.
        (aBuilder setting: #secondarySelection)
            label: 'Secondary'.
        (aBuilder setting: #findReplaceSelection)
            label: 'Find/replace'.
        (aBuilder setting: #selectionBar)
            label: 'Selection bar']
```

As you can see, there is absolutely nothing new in this declaration. The only thing that changes is that the value of the preferences are of a user defined class. In fact, in case of user defined or application specific preference class, the only particular thing to do is to implement one supplementary method for the view. This method must be named settingInputWidgetForNode: and must be implemented as a class method.

The method settingInputWidgetForNode: responsibility is to build the input widget for the *Settings Browser*. This method takes a SettingDeclaration as argument. SettingDeclaration is basically a model and its instances are managed by the *Settings Browser*.

Each SettingDeclaration instance serves as a preference value holder. Indeed, each setting that you can view in the *Settings Browser* is internally represented by a SettingDeclaration instance.

For each of our text selection preferences, we want to be able to change their colors and if the selection is optional, have the possibility to enable or disable their. Regarding the colors, depending on the selection preference value, only the background color is always shown. Indeed, if the text color of the preference value is nil, this means that having a text color does not make sense and then the corresponding color chooser is not built.

The settingInputWidgetForNode: method can be implemented as below:

```
TextSelectionPreference class>>settingInputWidgetForNode: aSettingDeclaration
    | preferenceValue backColorUI usedUI uiElements |
    preferenceValue := aSettingDeclaration preferenceValue.
    usedUI := self usedCheckboxForPreference: preferenceValue.
    backColorUI := self backgroundColorChooserForPreference: preferenceValue.
    uiElements := {usedUI. backColorUI},
        (preferenceValue textColor
            ifNotNil: [ { self textColorChooserForPreference: preferenceValue } ]
            ifNil: [{}]).
    ^ (self theme newRowIn: self world for: uiElements)
        cellInset: 20;
```

```
    yourself
```

This method simply adds some basic elements in a row and returns the row. First, you can notice that the actual preference value, an instance of TextSelectionPreference, is obtained from the SettingDeclaration instance by sending #preferenceValue to it. Then, the user interface elements can be built based on the actual TextSelectionPreference instance.

The first element is a *checkbox* or an empty space returned by the #usedCheckboxForPreference: invocation. This method is implemented as follow:

```
TextSelectionPreference class>>usedCheckboxForPreference: aSelectionPreference
  ^ aSelectionPreference optional
    ifTrue: [self theme
          newCheckboxIn: self world
          for: aSelectionPreference
          getSelected: #used
          setSelected: #used:
          getEnabled: #optional
          label: ''
          help: 'Enable or disable the selection']
    ifFalse: [Morph new height: 1;
           width: 30;
           color: Color transparent]
```

The next elements are two color choosers. As an example, the background color chooser is built as follows:

```
TextSelectionPreference class>>backgroundColorChooserForPreference:
      aSelectionPreference
  ^ self theme
    newColorChooserIn: self world
    for: aSelectionPreference
    getColor: #backgroundColor
    setColor: #backgroundColor:
    getEnabled: #used
    help: 'Background color' translated
```

Now, in the *Settings Browser*, the user interface looks as shown in Figure 1.13, with only one line for each selection kind instead of three as in our previous version.

1.9 Chapter summary

We presented Settings, a new framework to manage preferences in a modular way. The key point of Settings is that it supports a modular flow of control: a package is responsible to define customization points and can use

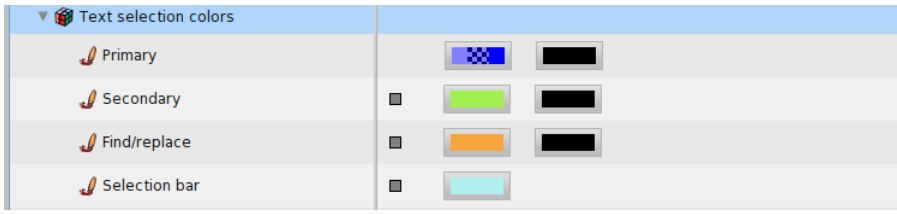

Figure 1.13: The text selection settings implemented with a specific preference class

them locally, then using Settings it is possible to describe such customization points. Finally, the Settings Browser collects such setting descriptions and presents them to the user. The flow is then from the *Settings Browser* to the customized packages.