Submitted to IWST 2016

Pragmas: Literal Messages as Powerful Method Annotations

S. Ducasse E. Miranda A. Plantec

Abstract

Often tools need to be extended at runtime depending on the availability of certain features. Simple registration mechanisms can handle such a situation: It often boils down to represent an action and describe such action with some meta-data. However, ad-hoc registration mechanisms have some drawbacks: they often not uniform and do not fit well with code navigability. In addition, meta data is not automatically synchronized with the data or behavior it describes. In this article we present the notion of Pragmas, method annotations, as it was introduced in VisualWorks and now it is an important extensibility mechanism of Pharo. We present some examples of pragmas within Pharo.

1. Introduction

Often tools need to be extended at runtime depending on the availability of certain features. This is typically the case for menubar offering access to currently loaded tools. Before pragmas were introduced in VisualWorks [6], the launcher's menubar was static and had lots of disabled entries for launching tools that were sold separately such as DLLAndC-Connect. It was a clear signed that a registration mechanism was missing at method level.

Simple registration mechanisms can handle such a situation: It often boils down to represent an action and describe such action with some metadata [4]. However, ad-hoc registration mechanisms have some drawbacks:

- They often not uniform. The user has to adapt to each of them.
- They do not fit well with code navigability and their existence and use may be difficult to discover.
- One important aspect with registration mechanisms is how to keep meta-data and method in sync. With ad-hoc registration mechanisms, meta data is not automatically synchronized with the data or behavior it describes. It is often the responsibility of the user to keep such information up to date.
- Finally ad-hoc registration mechanisms do not fit well with the variability of arguments.

In this article we present Pragmas, method annotations, as it was introduced in VisualWorks [6, 7] and now it is an important extensibility mechanism of Pharo [3]. Method pragmas are method level annotation that integrate smoothly with the Smalltalk syntax and tools.

The outline of the paper is the following: first we present a simple set of requirements for code annotations. Then we present the history and motivation behind first implementation of Pragmas. In the subsequent sections we present the API and propose an analysis of Pragmas. Finally, we present some examples of pragmas within Pharo. In particular, we show that while pragmas as method annotations are inherently a static constructs they are the basis to build dynamic solutions that react to method annotation changes.

2. An Analysis for Program Annotations

A good use of method annotations is to associate meta-data with a particular method. Having calls off to the side always introduces the need for book-keeping to keep those methods off to the side in sync with the methods they're describing. Typically everyone rolls their own registration mechanism. But at the expense of adding a level of triggering just to keep the metadata methods in sync.

Annotation Requirements. Here is a simple list of requirements for program annotation mechanisms.

- *Uniformity.* Introducing a special syntax for annotations can lead to large engineering efforts and should be minimized when possible.
- *Handle variability.* A good annotation system should be able to handle the variability of of the annotation needs. Since method annotations are static in the sense that they annotate program elements, they cannot access to runtime elements such as receiver and method arguments.
- *Discoverable/Searchable.* The introduction of a new mechanism should also consider the impact on the discovery of such new constructs. When cross-referencers

are more advanced than mere textual references, it is important that annotations can be found as an high-level concept.

- *Synchronized metadata.* The annotation and its associated element should be kept synchronized. The distance between the annotation and its element should be as short as possible to make sure that the users can understand that an element is annotated.
- *Any type of program element.* An annotation mechanism should be able to annotate any program elements.

We now present method pragmas an method level annotation system integrating smoothly with the Smalltalk syntax and the tools, and keeping the distance minimal with the annotated method. But we start first with a little history of Pragmas.

3. Some History First

Steve Dahl and Eliot Miranda developed pragmas at ParcPlace, with Vassili Bykov adding abstractions for accessing them. The first step was to replace some ugly class-side code to set unwind bits in ensure: and ifCurtailed: [1, 2] by a pragma the compiler would recognize and set the bits itself. The first real use was to make the VisualWorks launcher's menus extensible. With pragmas the launcher's menu was defined with the base system's tools and then extended as each tool package was loaded, or cut-back as each tool was unloaded. So that decoupled the launcher from introducing new tools.

VisualWorks then started using it for the browser and one could plug-in a single tool without redefining the browser's menu methods, which decoupled each extension. All this was done in the context of the parcel system [7]. Pragmas allowed one to decouple these tools where they collided in places like menu definition, tool registration.

Then Tami Lee, who was managing the COM connection that turned a VW image into a COM server, became the first "user" of pragmas. She used it to replace a lot of class-side methods that defined the signatures of methods that comprised the server. One could define the COM signature for a method in the method itself, and the class side lost about three separate methods that defined all that metadata. One could read the server method itself and understand its semantics without having to consult the class-side methods. One didn't have to know that there was metadata hidden on the class side because it was right there in your face.

Then Vassili Bikov used it for the inspector framework, Trippy, which was a huge improvement over the old Inspector framework, again resulting in a much more pluggable, decoupled and extensible system. Vassili also added the abstractions for accessing pragmas in methods.

Then VisualWorks added checking so that one could restrict the compiler to accept only legal pragmas for a given class. But if we defined the legal pragmas in a class-side method, say legalPragmas, then this would be exactly the kind of single point for extensions that causes collisions between packages, each of which might want to add its own set of pragmas. The solution was to use a pragma to mark a class-side method as defining a set of legal pragmas for a class. One could have more than one method defining a set of legal pragmas; packages wishing to add their own cool pragmas were decoupled. Once the system because recursive.

4. Pragma: Method Annotation for Smalltalk

Pragmas are a Smalltalk-centric way of adding arbitrary metadata to methods; Smalltalk-centric in that a pragma is a Message instance. It may be queried for senders, executed, etc, and that it can be parsed using the standard compiler - they add no new syntax.

A pragma represents the occurrence of an annotation in a compiled method. A pragma is a literal message pattern that occurs between angle brackets at the start of a method after any temporaries. A common example is the primitive pragma:

```
<primitive: 123 errorCode: 'errorCode'>
```

But one can add one's own and use them as metadata attached to a method. Because pragmas are messages, one can browse senders and implementors and perform them. One can query a method for its pragmas by sending it the pragmas message, which answers an Array of pragma instances, one for each pragma in the method. A pragma holds information about its defining class, method, its selector, as well as the information about the pragma keyword and its arguments. Instances are retrieved using one of the pragma search class methods.

In Pharo, the expression SystemNavigation new browseAllSelect: [:m| m pragmas notEmpty] browse all methods with pragmas in the system. The following expression SystemNavigation new browseAllSelect: [:m| m primitive isZero and: [m pragmas notEmpty]] let us browse all non-primitive methods with pragmas.

5. Discovering the API

In this section, we present the essential aspects of the Pragma API as in Pharo [3]. We start with the static navigation and then we show how pragmas can be executed.

Declaring a Pragma. First a pragma should be declared or attached to a method using the < > syntax. Such syntax is the same as the ones used to mark primitive methods [5]. Here we see that the method gtInspectorColorIn: of the class Color is annotated with the pragma gtInspectorPresentationOrder: 30. Such pragma takes 30 as argument.

```
Color >> gtInspectorColorIn: composite
  <gtInspectorPresentationOrder: 30>

  composite morph
```

Figure 1. Pragma structural API.

```
    title: 'Color';
    display: [ BorderedMorph new color: self ]
```

The pragma syntax follows the one of message sends. But since they are static code annotations their argument can only contain literal objects.

Accessing method annotation. A method can be annotated by several pragmas. We can access a pragma from the method it annotated using the pragmas message (see Figure 1).

```
pragma := (Color >> #gtInspectorColorIn:) pragmas first
pragma arguments
> 30
```

Once we get the pragma object itself we can access its selector using the message keyword (which should have been named selector to match the message API).

```
pragma keyword
> #gtInspectorPresentationOrder:
```

Accessing annotated method. From a pragma we can access the method its annotates using the message method. The message selector returns the method selector. It should have been named methodSelector. to be coherent with the message methodClass which returns the class defining the method (see Figure 1).

```
pragma selector
> #gtInspectorColorIn:
pragma method
> Color>>#gtInspectorColorIn:
pragma methodClass
> Color
```

Querying Pragmas. Pragmas act as a registration mechanism since they can be queried at different scope (full system, package, class). Once the pragmas are collected the programmer can have access to the pragma itself and its annotated method.

```
Pragma allNamed: #alarm: in: Pragma
```

The Pragma class provides some functionality to query the methods. The previous expression gathers all the pragma named #alarm: limited to the class Pragma itself. The next expression shows that we can scope the lookup to a branch in the hierarchy.

```
Pragma allNamed: aSym from: Point to: Object
```

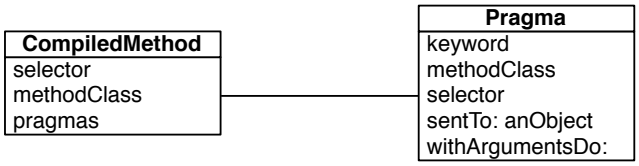

Figure 2. Pragma behavioral API.

The PragmaCollector tool developed in Pharo offers more advanced querying facilities.

Executing a Pragma. A pragma is not just a method annotation. Pragma are similar to messages (Instance of the class Message). As a message, a pragma it can also be executed once provided with a receiver. The message sendTo: anObject allows one to execute pragma by providing one receiver.

Imagine that we have the following code: In a class we define the method test. This method is annotated with a pragma named alarm:. Then we define a class named Alarm. This class defines the method alarm:.

```
AClass >> test
    <alarm: 'Executing pragma' >
    ^ 12

Alarmer >> alarm: aString
    UIManager default alert: aString
```

The following code snippet then asks the pragma associated to the method AClass»#test to execute itself with an instance of Alarmer. As a result the alarm: method of such class is executed.

```
(AClass >> #test) pragmas first sendTo: Alarmer new
```

The message withArgumentsDo: aBlock executes a block on the values of the arguments passed as block values. We can get a similar result than with the message sendTo: using the message withArgumentsDo: as follows:

```
(AClass >> #test) pragmas first withArgumentsDo: [ :each | UIMan-
ager default alert: each ]
```

6. Analyzing Pragmas

Now we analyze the Pragmas both at the conceptual level and implementation.

6.1 First class method annotations

We now evaluate the design of Pragmas.

Uniformity. Introducing a special syntax for annotations can lead to large engineering efforts and should be minimized when possible. In addition, the user has to learn a new syntactic.

Pragmas do not introduce any new syntactical elements and as such their integration and tooling is really easy.

Figure 3. Alternate API.

Handle variability. Pragmas are generic enough as all the use cases in the Pharo system shows it. Since pragmas are static annotation, only literal objects can be used as parameters. However, since we can invoke the associated methods, it is possible to use argument less pragmas and pass an argument to the method that acts as a builder (The Setting framework [2] uses this technique).

Discoverable/Searchable. Pragmas are perfectly discoverable using plain normal message browser. The tools managing the navigation in the IDE are able to handle pragmas. As already mentioned SystemNavigation queries return messages as well as pragma usage.

Synchronized metadata. As a summary, pragmas are always in sync with the methods they describe because they are embedded in their methods. In addition we can use triggering to do useful things, adding a pane to open inspectors as soon as we define the method that describes the pane, adding or removing a menu entry, etc.

Any type of program element. Pragmas are limited to method annotations. Class annotations are missing, even though developers annotate class methods to represent class annotations.

6.2 API

It is a bit confusing that while a pragma is supposed to be using the message syntax it does not follow its API. Hence keyword should be renamed selector and selector should be renamed methodSelector as in Figure 3.

6.3 Compiled-time vs. Runtime

Some pragmas do cause processing at compile time. For example, an FFI signature pragma can be checked at compile-time. But it's in keeping with Smalltalk that type checking is not performed at compile time in most cases, isn't it? Why should one require that pragmas be semantically checked at compile time when normal Smalltalk code isn't? At least one knows that the message instance the pragma is compiled to is a valid object and *can*be performed. So one does know at least that the pragma is executable. Obviously whether that ability to be executed only becomes potent with the right receiver. So any compiled pragma as the potential to be usefully evaluated.

6.4 Coupled Actions: Declaration and Execution

The ability to execute a pragma is a key element to its design. Indeed in many use cases the pragma helps to identify the method to be executed. This is the case in inspector, settings, and menu extensions.

The method is a component to be included in some larger structure, e.g., it is an action method on a menu, or it is an implementation of a pane in an inspector. The pragma is the message to be sent to the object that manipulates that larger structure to add the method to it. This is how menu pragmas work in VisualWorks and Pharo. There is an menu builder object. To add a method to a menu (and which menu is described by the pragma) the menu builder sets the method as its current method and then performs the pragma. In VisualWorks, the parameters in the pragma allow the MenuBuilder to add the method in the right way to the menu. In Pharo the pragma is without argument but the method has an argument that acts as a builder. The design is similar in the Setting framework [2] But the execution of the pragma is what actually adds the method to the menu. So its a combination of specification and execution.

7. Managing Pragmas Dynamically with the PragmaCollector

Querying pragmas can be achieved by using dedicated services provided by the Pragma class. But a tool may depend on the actual set of pragmas. In such a situation, a tool may need to adapt its internal state whenever a method containing a particular pragma is added, removed or updated. This is the role of the PragmaCollector.

This section describes the PragmaCollector and the pattern that is typically used by tools to dynamically update their internal state according to the actual set of pragmas.

7.1 The PragmaCollector

PragmaCollector responsibilities are to store a set of particular Pragmas and to dynamically keep its set of pragmas up-to-date. The selection of Pragmas is based on a filter which can be passed as a valuable with one argument at instantiation time. As an example, the following code shows how to instantiate a PragmaCollector to get the actual set of primitives.

```
(PragmaCollector
  filter: [:pragma | pragma keyword = 'primitive:']) reset
```

At initialization time, a PragmaCollector registers itself as a SystemAnnouncer subscriber. (SystemAnnouncer is the central notification for system event such class creation, method modifications...). The consequence is that a particular message is sent to the PragmaCollector each time a method is added, removed or updated in the system. When such an event occurs, an announcement is sent to all the registered PragmaCollector instances. Then a PragmaCollector may update its set of pragmas accordingly if the method is defined with a valid pragma according to the PragmaCollector filter. As an example, the sequence diagram of Figure 4 depicts

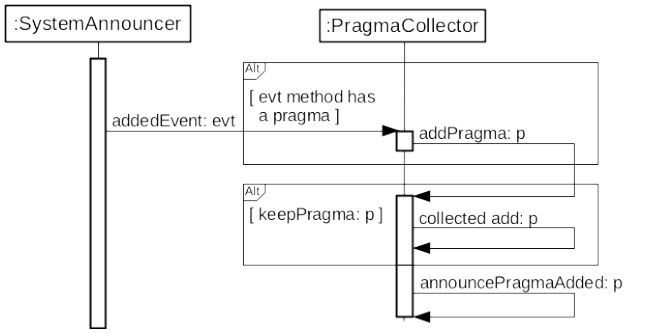

Figure 4. A sequence diagram for the method adding case

how a PragmaCollector can update its set of pragmas dynamically when a method is added in the system.

A PragmaCollector owns an announcer that can register objets which needs to be notified each time the PragmaCollector set of pragmas is changed. PragmaAnnouncement is the superclass of all pragma related announcement classes. In case of an adding, a removing or an updating, corresponding announcement classes are, respectively, PragmaAdded, PragmaRemoved and PragmaUpdated. Thus, a tool can register itself as a listener of its PragmaCollector announcer to be able to adapt its internal state whenever a method tagged with a particular kind of pragma is added, removed or updated. The next part of this section explains the design pattern which is typically used by tools to keep their internal state up-to-date.

7.2 Menu Builder Pattern

The Pharo root menu building uses pragmas. The menu tree is built by evaluating all the class methods declared with the pragma <*worldMenu*>: the receiver is the class owning the method and a menu builder is passed as argument. The menu builder is stored as an instance variable value of the *WorldState* singleton. A menu builder stores the menu tree. It uses a PragmaCollector instance to get the actual set of methods with the <*worldMenu*> pragma. Note that pragmas are spread over the classes supporting a modular design. Here we show two examples on in WorldState and one in StartupPreferencesLoader.

```
WorldState class >> quitItemsOn: aBuilder
  <worldMenu>
  (aBuilder group: #QuitPharo)
    order: 9999;
    with: [
      (aBuilder item: #'Save')
        target: self;
        selector: #saveSession;
        help: 'save the current version of the image on disk';
        keyText: 'S';
        icon: Smalltalk ui icons smallSaveIcon.
      (aBuilder item: #'Save as...')
        target: self;
        selector: #saveAs;
        help: 'save the current version of the image on disk under a new name.';
        icon: Smalltalk ui icons smallSaveAsIcon.
      (aBuilder item: #'Save and quit')
        target: self;
        selector: #saveAndQuit;
        help: 'save the current image on disk, and quit Pharo.';
        icon: Smalltalk ui icons smallQuitIcon.
      ... ]

StartupPreferencesLoader class >> systemStartupMenuOn: aBuilder
  <worldMenu>

  (aBuilder item: #SystemStartup)
    label: 'Startup';
    parent: #System;
    order: 2;
    help: 'System startup related';
    icon: Smalltalk ui icons scriptManagerIcon
```

8. Some Pragma Applications

Pragmas are heavily used both in VisualWorks and Pharo. The examples cover different categories: Pragmas are used for pluggable UIs (extensible menus, inspectors, setting declaration) where the method specifies an operation within the framework and the pragma specifies where and how the operation fits within a UI. Pragmas are also used as metadata used by a compilation system: the VW COM server exports Smalltalk methods through COM to make a VW COM server; the types for Smalltalk methods used to be specified in a single class-side initialize method; the use of pragmas allowed the metadata to be added to each server method, allowing the system to be extensible again.

In the following we present examples that are heavily used in Pharo: the customization of inspector panes and the setting declarations.

8.1 Use 1: GTInspector Panes

There is a problem for extending a system where there is a separate registration method that specifies the set of operations to be integrated. This method is a point of collision between different subsystems that want to extend a shared resource. There is a combinatorial explosion of all the different combinations of subsystems resulting in all the different possible versions of this registration method. Pragmas solve this problem by putting the registration coordinates into the operations, so that the registration method only has to register, it doesn't have to decide *what* to register.

GTInspector uses pragmas to extend classes with the different views that they want to expose the user in the inspector. The following methods show three of the views proposed by CompiledMethod. Figure 5 shows some of the different panes that the programmer has access to.

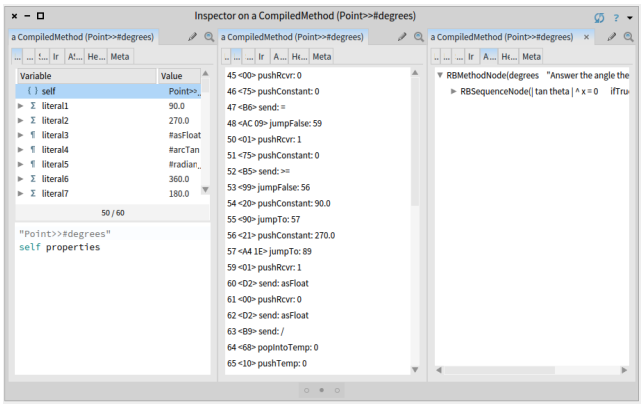

Figure 5. GTInspector shows different aspects of a compiled method.

```
CompiledMethod >> gtInspectorASTIn: composite
   <gtInspectorPresentationOrder: 35>
   (GTSimpleRBTreeBrowser new treeIn: composite)
      title: 'AST';
      display: [ :anObject | anObject ast ]

CompiledMethod >> gtInspectorBytecodeIn: composite
   <gtInspectorPresentationOrder: 30>
   ^ (GTBytecodeBrowser new treeIn: composite)
      title: 'Bytecode'

CompiledMethod >> gtInspectorSourceIn: composite
   <gtInspectorPresentationOrder: 30>
   ^ composite pharoMethod
      title: 'Source';
      smalltalkClass: [ self methodClass ];
      display: [ self getSource ];
        act: [ self browse ] icon: GLMUIThemeExtraIcons glamorousBrowse entitled: 'Browse'
```

8.2 Use 2: Settings

A *setting* is a 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 <systemsettings> serves as a classification tag which is used to automatically identify the method as a setting.

One 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 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

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.
```

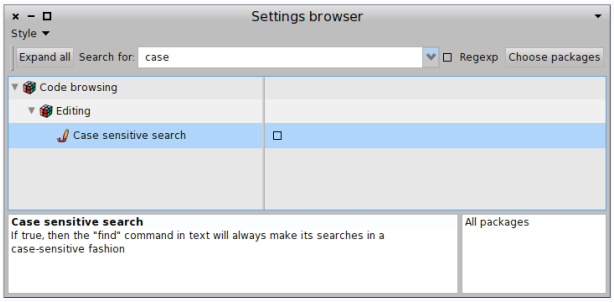

Figure 6. The *caseSensitiveFinds* setting.

The domain of preferences is large: To describe all possible preference kinds (color, strings, boolean, url, emails) and default values, we would need a lot of pragma parameters - many of which would not be relevant for certain settings. The method definitions below show variations of such parameters.Therefore Settings used pragmas as a simple tag and coupled it with a builder whose responsibility is to offer an adequate and flexible API to specify settings.

In the method declaring a setting, the pragma <systemsettings> identifies the method and the Setting framework invoke it with a setting builder that the method uses to specify a specific setting.

```
SourceCodeFonts class >> settingsOn: aBuilder
   <systemsettings>

   (aBuilder setting: #useSourceCode)
      parent: #appearance;
      order: 4;
      target: self;
      icon: Smalltalk ui icons smallConfigurationIcon;
      label: 'Source Code Fonts';
      description: 'Use Source Code Pro Fonts';
      precondition: [ FT2Library current notNil ];
      dialog: [ self fontSourceCodeRow ].
```

A Layered Architectural as Benefits. The use of pragmas supported the building a layered architecture. Figure 7 shows important points of the architecture put in place: 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 7, 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 7, the package *UI-Basic Setting* defines a method. 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 settings for user presentation. The Settings package defines tools to manage settings such as a Setting Browser that the user opens to change her/his preferences. The Setting Browser collects settings by querying pragmas 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.

9. Conclusion

In this paper we presented Pragmas: method annotations that act as statically described message sends. Pragmas do not require the modification of the syntax and are fully integrated in the IDE and tools supporting code navigation. In addition, we presented the PragmaCollector a tool to dynamically keep a set of pragmas up-to-date. Each time a method is recompiled or redefined the pragmas are updated. We present to use cases in production in Pharo since a couple of years. Finally we showed that pragmas support the design of modular libraries and as such more modular systems.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020.

References

[1] American National Standards Institute, Inc. *Draft American National Standard for Information Systems — Programming Languages — Smalltalk*. American National Standards Institute, 1997.

[2] A. Bergel, D. Cassou, S. Ducasse, and J. Laval. *Deep Into Pharo*. Square Bracket Associates, 2013. ISBN 978-3-9523341-6-4. URL http://rmod.inria.fr/archives/books/Berg13a-PBE2-ESUG-2013-09-06.pdf.

[3] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009. ISBN 978-3-9523341-4-0. URL http://pharobyexample.org/,http://rmod.inria.fr/archives/books/Blac09a-PBE1-2013-07-29.pdf.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002. ISBN 1-55860-639-4. URL http://www.iam.unibe.ch/~scg/OORP.

[5] A. Goldberg. *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, Reading, Mass., 1984. ISBN 0-201-11372-4.

[6] VisualWorks. Cincom Smalltalk. http://www.cincomsmalltalk.com/, archived at http://www.webcitation.org/5p1rRxls5, 2010. URL http://www.cincomsmalltalk.com/.

[7] R. Wuyts and S. Ducasse. Unanticipated integration of development tools using the classification model. *Journal of Computer Languages, Systems and Structures*, 30(1-2):63–77, 2004. . URL http://scg.unibe.ch/archive/papers/Wuyt04aClassifications.pdf.

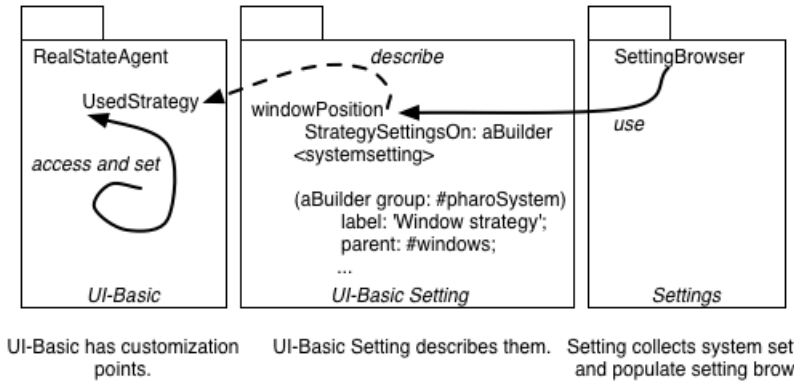

Figure 7. A package declares and uses customization points as variables. As an example, *UsedStrategy* is declared as a class variable of RealEstateAgent. Such customization points are described with Setting instances that are created by the automatic running of setting declaration methods. The Setting Browser collects the setting instances by querying pragmas and presents them to the user.