Pragmas: Literal Messages as Powerful Method Annotations

Stéphane Ducasse, Eliot Miranda, Alain Plantec

Pragmas: Literal Messages as Powerful Method Annotations

S. Ducasse
RMoD - INRIA
http://stephane.ducasse.free.fr

E. Miranda
http://www.mirandabanda.org

A. Plantec
alain.plantec@univ- brest.fr
Université de Bretagne Occidentale

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 metadata. However, ad-hoc registration mechanisms have some drawbacks: they are often not uniform and do not fit well with code navigability. In addition, metadata 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 [8], the launcher's menubar was static and had lots of disabled entries for launching tools that were sold separately such as DLLAndCConnect. It was a clear sign 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 are 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 metadata and method in sync. With ad-hoc registration mechanisms, metadata 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 [8, 9] and now it is an important extensibility mechanism of Pharo [3]. Method pragmas are method level annotations 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 the 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 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 metadata with a particular method. Building an ad-hoc registration mechanism is not complex. Typically, ad-hoc registration mechanisms use a collector object holding a list of object representing metadata. Users should explicitly call the collector to register to it and the system using the metadata uses such collector. However such practice is a problem because the method and its metadata are not automatically kept in sync. So another layer of triggering should be put in place to make sure that each time a method changes its metadata is (or not) updated. The programmer has to look in different places to find and update the information.

Now we list the properties that a good program annotation mechanism should exhibit.

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 the annotation needs. Since method annotations are static in the sense that they annotate program elements, they cannot access 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 matching (e.g. message sends), it is important that annotations can be found as a 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, a method level annotation system integrating smoothly with the Smalltalk syntax and the tools, and keeping minimal distance with the annotated method. But we start first with a little history of pragmas, since pragmas have been designed around 2003 for VisualWorks.

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 redesign 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. This development decoupled the launcher from introducing new tools.

VisualWorks then started using pragmas 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 [9]. pragmas allowed one to decouple these tools where they collided in places like menu definition and tool registration.

Then, Tami Lee, who was managing the COM connection that turned a VisualWorks image into a COM server, became the first "user" of pragmas. She used pragmas 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 defined in the method itself.

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.

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 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: the argument identifies the Virtual Machine primitives to be executed.

```
LargeInteger >> // anInteger
    "Primitive. Divide the receiver by the argument and return the result. Round the result down towards negative infinity to make it a whole integer. Fail if the argument is 0. Fail if either the argument or the result is not a SmallInteger or a LargePositiveInteger less than 2-to-the-30th (1073741824)."

  <primitive: 32>
  ^ super // anInteger
```

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]
```

browses all methods with pragmas in the system. The expression:

```
SystemNavigation new
  browseAllSelect: [:m| m primitive isZero
                        and: [m pragmas notEmpty]]
```

browses 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 that used to mark primitive methods [6]. Here we see that the method gtInspectorColorIn: of the class Color is annotated with the pragma gtInspectorPresentationOrder: 30. This pragma takes 30 as argument.

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

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

The pragma syntax follows that of message sends. But since pragmas 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 annotated method 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 be renamed selector to match the message API).

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

Accessing annotated method. From a pragma we can access the method it annotates using the message method. The message selector returns the method selector. As we will discussed in following section such API could be improved.

```
pragma selector
> #gtInspectorColorIn:
```

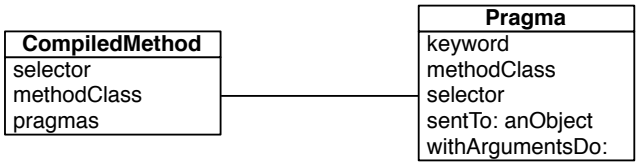

Figure 1. Pragma API.

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

Querying pragmas. pragmas act as a registration mechanism since they can be queried at different scopes (full system, package, class). Once pragmas are collected, the programmer can have access to the pragma itself and its annotated method. The Pragma class provides some functionality to query the methods. The following expression gathers all the pragma named #alarm: limited to the class Pragma itself.

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

The next expression shows that we can scope the lookup to a branch in the hierarchy.

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

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

Executing a Pragma. A pragma is not just a method annotation. Pragmas are similar to messages[1]. Similarly to a message, a pragma 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 Alarmer. 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 is executed.

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

The class Pragma defines another method supporting its execution. The message Pragma»#withArgumentsDo: aBlock

[1] Messages in Smalltalk are instances of the classes Message

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

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

6. 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 and we describe it now.

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.

6.1 The PragmaCollector

PragmaCollector responsibilities are to store a set of particular pragma instances 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 events such as class creation, method modifications, etc). 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 2 depicts how a PragmaCollector can update its set of pragmas dynamically when a method is added in the system.

A PragmaCollector also owns an announcer that registers objects which need to be notified each time the PragmaCollector set of pragmas is changed. PragmaAnnouncement is the superclass of all pragma related announcement classes. In the case of an addition, removal or update, the 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.

The following presents a pattern which is typically used by tools to keep their internal state up-to-date.

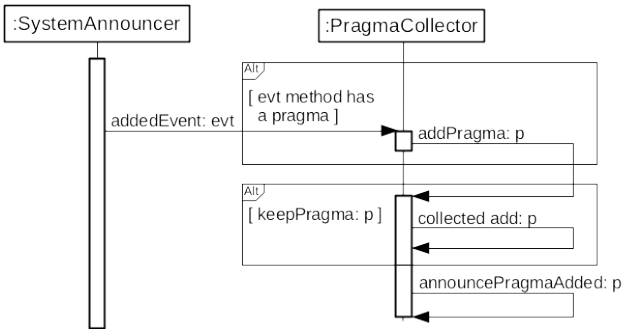

Figure 2. A sequence diagram for the method adding case

6.2 Menu Builder Pattern

The Pharo root menu building uses pragmas. The menu tree is built by evaluating all the class methods declared using the pragma <*worldMenu*>: the receiver is the class owning the method and a menu builder is passed as argument. A pragma without argument is used and the annotated method is called with a builder as argument: this pattern is used to handle the fact that building menu can have multiple exclusive parameters (See Section 7.2 for another application of this pattern).

A menu builder builds and stores a menu tree. It uses a PragmaCollector instance to get the actual set of <worldMenu> annotated methods. Note that pragmas are spread over the classes supporting a modular design. Here we show two examples one 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
```

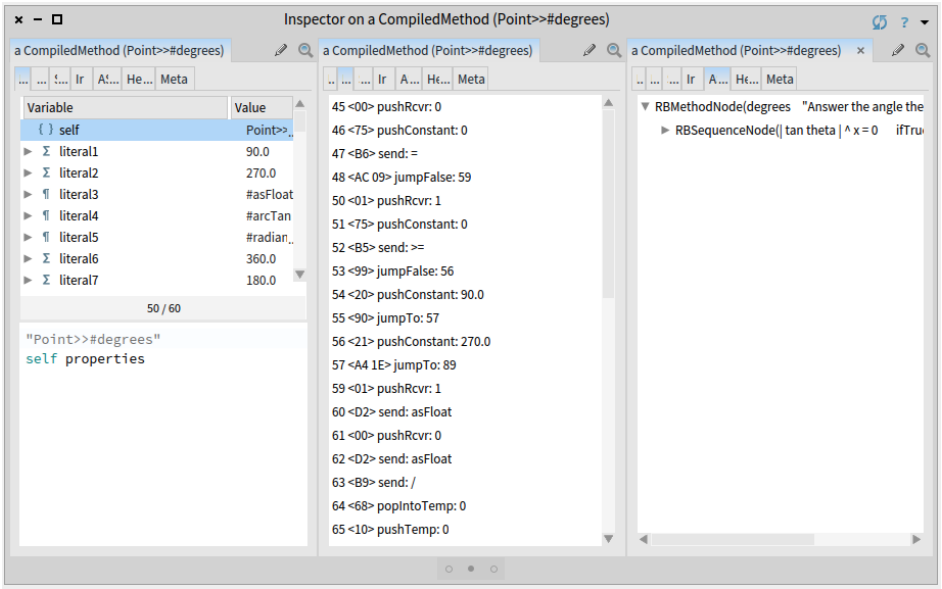

Figure 3. GTInspector in action. Three of the different views of a compiled method exposed to the developer.

```
<worldMenu>

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

7. 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 VisualWorks COM server exports Smalltalk methods through COM to make a VisualWorks 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 setting declarations.

7.1 Use 1: GTInspector Panes

GTInspector is an extensible inspector. It uses pragmas to extend classes with the different views that are exposed the user in the inspector. The following methods show three of the views exposed by the CompiledMethod class. Figure 3 shows some of the different panes that the programmer has access to.

```
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 glam-
orousBrowse entitled: 'Browse'
```

7.2 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.
```

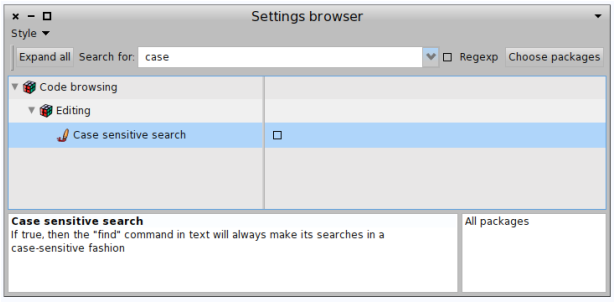

Figure 4. The Case sensitive search setting.

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 The Settings framwork uses pragmas as a simple tag and associate this pragma usae 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 as declaring a setting. The Settings framework invokes this identified method with a builder that the method uses to define the actual setting object.

```
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 Benefit. The use of pragmas supported the building of a layered architecture. Figure 5 shows three packages: The *Settings* package defines tools to manage settings such as a Setting Browser that the user opens to change her/his preferences. It uses descriptions packaged in package *UI-Basic Setting*. Such descriptions describe behavior of elements packaged in package *UI-Basic*. The class RealStateAgent follows the behavior expressed in its class variable UsedStrategy.

Figure 5 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 5, 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 parameterized.

Description of customization point. In Figure 5, 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 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.

8. Analyzing pragmas

Now we analyze the pragmas both from a conceptual and implementation point of view.

8.1 First class method annotations

We now evaluate how pragmas answer the requirements for method annotations.

Uniformity. Pragmas do not introduce any new syntactical elements and as such their integration and tooling is facilitated. In particular, no special handling has been necessary to be able to query pragmas as message senders. Pragmas are

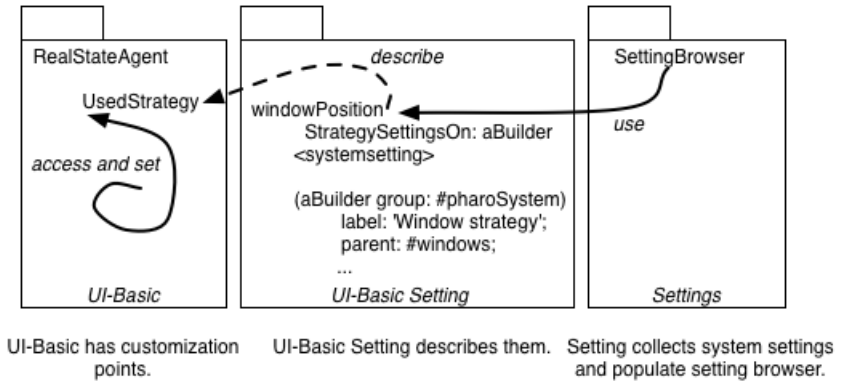

Figure 5. A package declares and uses customization points as variables. As an example, *UsedStrategy* is declared as a class variable of RealEstateAgent. Such customization points are described with Setting instances that are created by the automatic running of setting declaration methods. The Setting Browser collects the setting instances by querying pragmas and presents them to the user.

automatically part of "Senders of..." results and this eases discoverability.

Handle variability. Pragmas are generic enough to handle the use cases in the Pharo system (as well as the ones of VisualWorks). Since pragmas are static annotations, 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 - see Section 6.2 for example). In Pharo 50 there are 143 different pragmas for 4337 annotations with the following distribution based on their arity: 95-0, 30-1, 11-2, 5-3, 1-4 and 1-5. The pragma (preference:category:description:type:) with 4 arguments with 4 elements is not used in Pharo but a behavior compatible with the Squeak settings. One of the biggest limits of pragmas is that they do not handle class annotations.

Discoverable/Searchable. Pragmas are perfectly discoverable using normal message browsers. 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. Debugging pragmas has nothing really specific, a developer can query all the annotated methods with a given pragma and use the Pragma API to access all the data necessary to debug.

Synchronized metadata. Metadata, when it cannot be extracted automatically from the entity it describes, can always be out of sync. When metadata can be extracted directly from the entity it would describe then there is no need to have metadata expressed since it would duplicate such information and lead to potential desynchronization. While pragmas per se do not ensure that methods and their annotations will not be desynchronized, their locality and minimal distance to the entity they describe is a good incentive for the programmers to make sure that a method and its annotations are kept in sync.

As a summary, pragmas favor synchronized metadata because pragmas are embedded in their methods. In addition Pharo offers automatically trigger notifications on pragma modifications. This help building advanced behavior such as 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 and package annotations are missing. Developers can annotate class methods to represent class and package annotations. However, annotating class methods to define class annotations relies on the interpretation by the pragmas user, since it could conflict with the annotation of a single class methods.

8.2 About decoupled information

Using a monolithic design where all metadata is described in a single place makes sure that the programmer fully control the order of the declaration. Such control comes at the price of offering a modular way to build applications. Such order may be not relevant in certain cases but it is important for example in UI elements such as menu items or inspector panes.

When pragmas are used to describe menu items or panes, the pragma collector client has several possibilities to sort items: order given by the collection of the pragmas or any alphabetic sort based on a pragma properties. When order is relevant, often metadata includes an explicit order. In case of conflict a local ordering is done. Such practice is not tight to pragmas, metadescription frameworks such Magritte [5, 7] use the same simple strategy.

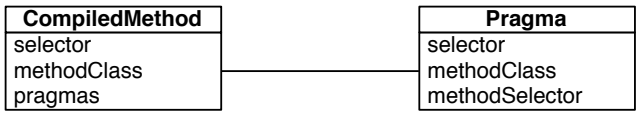

Figure 6. Suggested API polymorphic to Message API.

8.3 For a polymorphic API with Message

It is a bit confusing that while a pragma is supposed to be using the message syntax it does not follow the Message API. keyword should be renamed selector and selector should be renamed methodSelector as in Figure 6. Hence methodSelector returns the annotated method selector, similarly methodClass returns the class of the annotated class and selector returns the selector of the pragma.

8.4 Compile-time vs. Runtime

The question of pragma validation at compile-time is worth discussions. Most pragmas are annotations that are queried at runtime and do not lead to any computation at compile-time. However, some pragmas do cause processing at compile time. For example, an FFI signature pragma can be checked at compile-time. By default, there is no type or arity checking in the current implementation. It follows the general that no message is Smalltalk is checked at compile-time. When a new method with different arity (and name) is defined, the compiler just compiles it without checking that it may not correspond to a given family. The same applies to pragmas. If checking would be required, it could be possible to define Pragma 'Class' or 'Signature' and the compiler could check whether a pragma is compliant with its Signature.

8.5 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 specify 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 a 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 Settings framework [2]. The execution of the pragma is what actually adds the method to the menu. So it's a combination of specification and execution.

9. Related Work

Java, C# and Javascript support annotations and used them really frequently. In Java, a method annotation is defined by @ For example, the following @Test declares that the method is a test method. Annotations are before methods. For exemple, JUnit requires an annotation before test methods:

```
@Test
Public void the method()
```

In Java, annotations are defined directly close to the language elements they target. Annotations can target classes, methods, variables, parameters and packages but also to local variables, method parameters, packages, even other annotations and also some Java specific program elements such as constructors, interfaces, enums [2].

Java defines a set of annotations that are built into the language. For example, @Override checks that the method is an override. @Deprecated marks the method as obsolete. @SuppressWarnings instructs the compiler to suppress the compile time warnings specified in the annotation parameters. In Java 8, new type annotations has been introduced @NonNull, @ReadOnly, @Regex, @Tainted and @Untainted, @m (for measure). Java SE 8 allows type annotations anywhere that a type is used. Previously, annotations were only allowed on definitions.

Annotations can be applied to other annotations such as @Retention to specify how the marked annotation is stored (whether in code only, compiled into the class, or available at runtime through reflection), @Target marks another annotation to restrict what kind of Java elements the annotation may be applied to.

Annotations are either used at compile-time or runtime. The annotation has to be itself annotated to be available at runtime. For example, the following specifies that the annotation interface is available at runtime and can be applied to packages, fields, constructors, types and methods.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE, ElementType.METHOD, Element-
Type.CONSTRUCTOR, ElementType.PACKAGE, ElementType.FIELD)
public @interface Note
  String value();
  Priority priority() default Priority.MEDIUM;
```

Annotations without the meta-annotation are not available on runtime.

Statically we can manipulate annotation using the APT (annotation processing tool). It allows you to write a meta program that can get information of the classes and the annotations they have. This is intended to be used in the generation of code, documentation and any infrastructure

[2] The full list is here: https://docs.oracle.com/javase/7/docs/api/java/lang/annotation/ElementT

previous to the running of your code. APT is often used as a preprocessor.

The other way is to use the reflection objects provided by the environment, a developer can ask a class all the annotated things (methods, the class it self and attributes) and perform all the sort of reflection Java lets you perform. The programmer can only access known classes (by reference or by name). In Java developers use classloaders to access all the classes and then access their annotations.

10. Conclusion

In this paper we presented pragmas: method annotations that act as statically described message sends. Pragmas do not require Smalltalk syntax modification and are fully integrated in the IDE and tools supporting code navigation. In addition, we presented the PragmaCollector: a tool that dynamically keeps a set of pragmas up-to-date. Each time a method is recompiled or redefined the pragmas are updated. We presented two use cases deployed 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. The authors want to thank the anonymous reviewers for their excellent reviews.

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] S. Ducasse, T. Gîrba, A. Kuhn, and L. Renggli. Meta-environment and executable meta-language using Smalltalk: an experience report. *Journal of Software and Systems Modeling (SOSYM)*, 8(1):5–19, Feb. 2009. . URL http://scg.unibe.ch/archive/drafts/Duca08a-Sosym-ExecutableMetaLanguage.pdf.

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

[7] L. Renggli. Magritte — meta-described web application development. Master's thesis, University of Bern, June 2006. URL http://scg.unibe.ch/archive/masters/Reng06a.pdf.

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

[9] 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.