



HAL
open science

Pharo with Style

Stéphane Ducasse

► **To cite this version:**

| Stéphane Ducasse. Pharo with Style. Square Bracket Associates, 2019. hal-02299550v1

HAL Id: hal-02299550

<https://hal.science/hal-02299550v1>

Submitted on 27 Sep 2019 (v1), last revised 3 Jun 2022 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Pharo with Style

Stéphane Ducasse

August 12, 2019

Copyright 2018 by Stéphane Ducasse.

The contents of this book are protected under the Creative Commons Attribution-ShareAlike 3.0 Unported license.

You are **free**:

- to **Share**: copy and redistribute the material in any medium or format,
- to **Adapt**: remix, transform, and build upon the material for any purpose, even commercially.

Under the following conditions:

Attribution. You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

Share Alike. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page:

<https://creativecommons.org/licenses/by-sa/3.0/>

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.



Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license):

<http://creativecommons.org/licenses/by-sa/3.0/legalcode>

Contents

Illustrations	iii
1 General naming conventions	3
1.1 Guideline: Favor simple direct meaning	3
1.2 Guideline: Avoid underscores and favor camel case	3
1.3 Guideline: Use camel case	4
1.4 Guideline: Use descriptive names	5
1.5 Guideline: Pay attention to meaning	5
1.6 Guideline: Method selectors start with lowercase	6
1.7 Guideline: Follow domain	6
1.8 Guideline: Shared variables start with uppercase	7
1.9 Guideline: Private variables start with lowercase	7
1.10 Guideline: Avoid underscore in variable identifiers	7
1.11 Guideline: Favor unique meaning and pronunciation	8
1.12 Guideline: Class names should indicate the class' parent	8
1.13 Guideline: Avoid name collisions	9
2 About variable names	11
2.1 Guideline: Favor semantic variables	11
2.2 Guideline: Use typed variables to indicate API	12
2.3 Guideline: Get the best from semantic and type variable	13
2.4 Guideline: Use semantics for state variable	13
2.5 Guideline: Use predicate for Boolean	14
2.6 Guideline: Use common nouns and phrases	14
3 Selectors	15
3.1 Guideline: Choose selectors to form short sentences	15
3.2 Guideline: Use imperative verbs for actions	15
3.3 Guideline: Prefix with as for conversion	16
3.4 Guideline: Indicate flow with preposition	16
3.5 Guideline: Indicate return types	17
3.6 Guideline: Use interrogative form for testing	18
3.7 Guideline: Avoid using parameter and variable name type	18
3.8 Guideline: Use accessors or not	19
3.9 Guideline: Name accessors following variable name	19
3.10 Guideline: Avoid prefixing with set public accessors	20

3.11	Guideline: Use basic or raw to access low-level	20
3.12	Guideline: Follow conventions and idioms	21
3.13	Guideline: Distinguish class vs. instance selectors	21
3.14	Guidelines: Follow existing protocols	23
4	Comments	25
4.1	Guideline: Method comments	25
4.2	Guideline: Avoid relying on a comment to explain what can be reflected in code	26
4.3	Guideline: Use active voice and short sentences	27
4.4	Guideline: Include executable comments	27
4.5	Guideline: Use CRC-driven class comments	28
4.6	Guideline: Comment the unusual	28
5	About code formatting	29
5.1	Guideline: Be consistent	29
5.2	Guideline: Use the general method template	29
5.3	Guideline: Indent method body	30
5.4	Guideline: Separate signature and comments from method body	31
5.5	Guideline: Use space to give breath to code	31
5.6	Guideline: Align properly	32
5.7	Guideline: Use tabs not spaces to indent and use spaces to help reading . .	32
5.8	Guideline: Do not break lines randomly	33
5.9	Guideline: Highlight control flow	34
6	Powerful coding idioms	35
6.1	Guideline: Do not query twice for the same object	35
6.2	Guideline: Move returns outside branches	35
7	Potential traps	37
7.1	Guideline: Use parentheses to disambiguate messages with the same priority	37
7.2	Guideline: no need for extra parentheses	38
7.3	Guideline: receiver of ifTrue:ifFalse: is a boolean	40
7.4	Guideline: receiver of whileTrue: is a block	40
7.5	Guideline: Use a Block when you do not know execution time	41
8	Conclusion	43
	Bibliography	45

Illustrations

Programming is a lot more than just writing algorithms or programs. Programming is all about communication. Communication with others: the other programmers that will participate to your development effort but also with yourself. Indeed finding good names is a really important task because using the right name often opens the door to new spaces where your design can bloom and expand.

The purpose of a programming style guide such as this book is to provide a simple vehicle for addressing the needs of a good communication. The goal is to make source code clear, easy to read, and easy to understand and extend.

These conventions are not cast in stone but they set the foundation of a common culture. Culture is important when programming.

We got influenced by the excellent little book called *Smalltalk with Style*. We hope that you will enjoy this one and that it will help you to become a better communicating designer.

Feedback and suggestions are welcome at stephane.ducasse@inria.fr. PullRequests on <https://github.com/SquareBracketAssociates/Booklet-PharoWithStyle> are also welcome.

Special thank to Christopher Fuhman, Benoit St Jean and Masashi Fujita, Nathan Reilly, Esteban Maringolo for their feedback. S. Ducasse - 12 August 2019.

General naming conventions

Names are important. We will never repeat it enough. A good name is often driven by a domain.

1.1 **Guideline: Favor simple direct meaning**

Some native English writers use often more precise but less common terms. Consider that your software may be read by people from different cultures. So use simple, mainstream, and common terms. Avoid hidden or implied meanings that can only be understood by a limited group of people. Make information explicit.

1.2 **Guideline: Avoid underscores and favor camel case**

Prefer

[`timeOfDay`

over

[`Not timeofday`
[`Not time_of_day`

Prefer

[`GnatXmlNode`

over

[`Not GNAT_XML_Object`

Prefer

```
[ releasedX
over
[ released_X
```

When creating private low-level methods that bind to external C-libraries, you may want to use underscores to follow C conventions to ease tracing back the communication between libraries. In such a case, limit your use to carefully thought cases.

1.3 Guideline: Use camel case

Remember `MaxLimit`, `maxLimit`, `maxlimit`, and `MAXLIMIT` are all different identifiers in Pharo.

```
[ | MaxLimit maxLimit |
MaxLimit := 10.
maxLimit := 20.
MaxLimit
>>> 10
```

Still, Pharo favors camel case, so use it systematically for words. Wikipedia defines camel case as: Camel case (stylized as `camelCase`) is the practice of writing phrases such that each word or abbreviation in the middle of the phrase begins with a capital letter, with no intervening spaces or punctuation.

For local variables, method parameters and instance variables, use

```
[ maxLimit

instead of

[ Not maxlimit
[ Not MAXLIMIT
```

For classes, or shared variables, use

```
[ OrderedCollection
[ MaxLimit

instead of

[ Not ORDEREDCOLLECTION
[ Not MAXLIMIT
```

Note In a compound word, do not confuse a prefix or suffix with a word when trying to determine which words should begin with an upper case letter. For example, some readers may think that the "c" in `subclass` should be upper case, but `sub` is a prefix, not a word. When in doubt about prefixes and suffixes, check a dictionary.

1.4 Guideline: Use descriptive names

Prefer

[superclass

over

[Not superClass

1.4 **Guideline: Use descriptive names**

Choose descriptive names that capture domain entities unambiguously.

Prefer

[timeOfDay

over

[Not tod

Prefer

[milliseconds

over

[not mil

Prefer

[editMenu

over

[not eMenu

1.5 **Guideline: Pay attention to meaning**

Non-English native speakers often misplace word qualifiers. In English the qualifier is often before the word it qualifies.

Prefer

[DateParser

over

[Not ParserForDate

Prefer

[userAssociation

(an association of users)

over

```
[ Not associationUser
```

Compare the three following variables:

```
[ sizeToRead
  sizeJustRead
  readSize
```

In this situation, avoid homographs (<https://en.wiktionary.org/wiki/homograph>). That is, words that are written the same way but can have different meanings or pronunciations. For example, *Did you read that book? ... Yes, I read it yesterday.* About `readSize`: does this mean that the size was just read (red) or it is the size to read (reed)? Favor words with unique pronunciation.

1.6 Guideline: Method selectors start with lowercase

Prefer

```
[ getMethodsNamesFromAClass: aClass
  | methodsNames |
  methodsNames := aClass selectors.
  methodsNames do: [ :each | names add: each ]
```

over

```
[ GetMethodsNamesFromAClass: aClass
  | methodsNames |
  methodsNames := aClass selectors.
  methodsNames do: [ :each | names add: each ]
```

Also, in this example the method selector is not good because method names are called selectors in Pharo. In addition in English `methodsNames` should be written `methodNames`. It should be `gatherSelectorsFrom:` or something similar.

1.7 Guideline: Follow domain

Follow the domain concepts and culture of the project. Do not invent your own terms because you think they are better. Favor regularity (consistency) over preciseness.

Prefer

```
[ GnatXmlNode
```

over

```
[ not GNAT_XML_Object
```

When representing the XML Ada abstract syntax tree in Pharo, we should not follow Ada naming conventions. The name should convey that the class is an

1.8 Guideline: Shared variables start with uppercase

abstract syntax tree node. Hence `GnatXmlNode` is much better than `GnatXmlObject`.

Another example is the following: In Moose, an importer is an object creating FAMIX entities (classes, methods, etc.) from the data structure representing a language element, usually an Abstract Syntax Tree (AST). Therefore `GNATInstaller`, which creates entities from an AST, should be renamed `GnatImporter` and `GNATImporter`, which loads an AST in memory should be renamed `GnatASTLoader`.

1.8 Guideline: Shared variables start with uppercase

Begin class names, global variables, pool variables, and class variables with an uppercase letter. If the word is compound, then use camel case for the rest.

```
[ Point "Class"
  Transcript "global variable"
  PackageGlobalOrganizer "class variables"
```

1.9 Guideline: Private variables start with lowercase

Begin instance variables, temporary variables, method parameters, and method selectors with lower case. If the word is compound, then use camel case for the rest.

```
[ address
  classExtensionSelectors
  classTags

Prefer

[ | dataset f xMatrix scale x |

over

[ | dataset f Xmatrix scale X |
```

1.10 Guideline: Avoid underscore in variable identifiers

In Pharo variables are using camelCase, just follow this convention.

```
Prefer

[ | dataset pca scaledX reducedX |

over

[ | dataset pca scaled_X reduced_X |
```

1.11 **Guideline: Favor unique meaning and pronunciation**

Choose names that have a unique meaning. Avoid homographs.

Prefer

```
[ sizeToRead
  sizeJustRead
```

```
[ Not readSize
```

Does this mean that the size was just read (red) or is it the size to read (reed)?

1.12 **Guideline: Class names should indicate the class' parent**

Suffix class names with the root class to convey the kind of object we are talking about.

For example, without the `Morph` suffix, the reader is forced to check the superclass to understand if the class is about a graphical object or not.

Prefer

```
[ ClyBrowserButtonMorph
```

over

```
[ Not ClyBrowserButton
```

Prefer

```
[ ClyQueryViewMorph
```

over

```
[ Not ClyQueryView
```

In the following, not mentioning the `Presenter` suffix makes it unclear to the reader that it is a `Presenter` object as opposed to a `Model` object.

Prefer

```
[ ApplicationWithToolBarPresenter
```

over

```
[ Not ApplicationWithToolBar
```

In the next example, `DynamicWidgetChange` does not convey that this is not a *domain* object representing a change, but a `Presenter` object in the `Model-View-Presenter` triad:

```
[ DynamicWidgetChangePresenter
```

over

1.13 Guideline: Avoid name collisions

[Not DynamicWidgetChange

1.13 **Guideline: Avoid name collisions**

To avoid name space collisions, add a prefix indicative of the project to the name of the class.

[PRDocument
CmdMessage

Note You may find that Pharo is lacking a namespace. If you have a couple hundred thousands euros, we can fix that!

Note, however, that even with a namespace you will have to pay attention that your namespace name does not collide with another one.

About variable names

When choosing an appropriate name for a variable, the developer is faced with the decision: *Should I choose a name that conveys semantic meaning to tell the user how to use the variable, or should I choose a name that indicates the type of object the variable is storing?* There are good arguments for both styles. Let us see what are the guidelines that can help us find the right balance.

2.1 Guideline: Favor semantic variables

A semantic name is less restrictive than a type name. When modifying code, it is possible that a variable may change type. But unless one redefines the method, the semantics of it will not change. We recommend using semantically meaningful names wherever possible.

In the example below, the typed variable does not indicate how it will be used whereas the semantic variable does.

Prefer

```
"Semantic variable"  
newSizeOfArray := numberOfAdults size max: numberOfChildren size
```

over

```
"Typed variable"  
anInteger := numberOfAdults size max: numberOfChildren size
```

Note that semantic name can convey variable roles. Having more information is definitively useful for clients of the code.

Prefer

```
["Semantic variable"
selectFrom: aBeginningDate to: anEndDate

over
["Type variable"
selectFrom: aDate to: anotherDate
```

Finding a semantic name is not always as obvious as demonstrated above. There are cases in which choosing a descriptive semantic name is difficult.

2.2 **Guideline: Use typed variables to indicate API**

Using type variable is interesting to convey the API (set of messages) that the object held in the variable responds to.

Below `aDictionary` conveys that the argument should have the same API as a `Dictionary` (`at:;at:put:`)

Prefer

```
[properties: aDictionary

over
[Not properties: map
```

You may also want to stress specific types in an API referencing to interface that the object implements.

Prefer

```
[properties: aPuttable

over
[Not properties: aCollection
```

Suppose a `String`, a `Symbol`, and `nil` are valid for a parameter. A developer may be tempted to use the name `aStringOrSymbolOrNil`.

You may be tempted to `aString` or `anObject`. `anObject` is not really helping the developer that will have to use such variables. At the minimum such a use should be accompanied with a comment that says, "anObject can be a `String` or a `Symbol`"

Some developers may argue that type variables should not refer to classes that do not exist. We disagree. As shown in the following guideline it is a lot better to indicate that an argument is block expecting two arguments (hence a binary block) than to just mention a block. And this even if there is no binary block class in the system.

Prefer

```
[inject: anObject into: aBinaryBlock
```

2.3 Guideline: Get the best from semantic and type variable

over

```
[inject: anObject into: aBlock
```

Note that for `inject: into:` the best naming is to mix semantic and type naming as in

```
[inject: initialValue into: aBinaryBlock
```

2.3 Guideline: Get the best from semantic and type variable

A good practice is to use a mixture of both semantic and typed variable names. Method parameter names are usually named after their type. Instance, class, and temporary variables usually use a semantic name. In some cases, a combination of both can be given in the names.

Prefer

```
[ifTrue: trueBlock ifFalse: falseBlock
```

over

```
[Not ifTrue: block1 ifFalse: block2  
Not ifTrue: action1 ifFalse: action2
```

The following are other examples of good names.

```
[inject: initialValue into: aBinaryBlock  
copyFrom: start to: stop  
findFirst: aBlock ifNone: errorBlock  
paddedTo: newLength with: anObject
```

2.4 Guideline: Use semantics for state variable

State variable names (instance variables, class variables, or class instance variables) are usually semantic-based. A combination of semantic and type information can be really powerful, too.

Prefer

```
["In class PhoneBook"  
phoneNumber  
name
```

over

```
[Not number  
Not labelForPerson
```

2.5 **Guideline: Use predicate for Boolean**

Use predicate clauses or adjectives for Boolean objects or states. Do not use predicate clauses for non-Boolean states.

```
[ alarmEnabled
```

2.6 **Guideline: Use common nouns and phrases**

Use common nouns and phrases for objects that are not Boolean.

```
"In class Vehicle..."
  numberOfTires
  numberOfDoors

"In class AlarmClock..."
  time
  alarmTime

"In class TypeSetter..."
  page
  font
  outputDevice
```

Note that you can also use count instead of numberOf as in the following example:

```
[  numberOfTires
   tireCount
```

Selectors

Method names in Pharo are called selectors. They are used in messages and are the main vehicle to convey adequate meaning. The correct use of words and design of selectors is important.

3.1 **Guideline: Choose selectors to form short sentences**

Choose method names so that someone reading the message can read the expression as if it were a sentence.

Prefer

```
[FileDescriptor seekTo: word from: self position
```

```
[Not FileDescriptor lseek: word at: self position
```

Write the test first, and make sure that your test scenario reads well.

3.2 **Guideline: Use imperative verbs for actions**

Use imperative verbs for message which perform an action.

```
[transform
  selectors do: [:each | self pushDown: each].
  selectors do: [:each | class removeMethod: each]
```

Prefer

```
[aReadStream peek
```

over

```
[Not aReadStream word
```

Prefer

```
[ aFace lookSurprised
  aFace beSurprised
```

over

```
[ Not aFace surprised
  skipSeparators
```

Pay attention that some words can be interpreted as interrogative, whereas you want to give them an imperative meaning.

For example, compare:

```
[ optimized
```

and

```
[ triggerOptimization
```

This is why using `beOptimized` would be better than a simple `optimized` and why `isOptimized` is better for the interrogative form.

3.3 Guideline: Prefix with `as` for conversion

When converting an object to another one, the convention is to prefix the class name of the target with `as`.

```
[ anArray asOrderedCollection
```

Favor the use of existing classes.

3.4 Guideline: Indicate flow with preposition

When a process state is going from one object to another, indicate the direction using meaningful names.

For example `flattenProperties:` is not a good name because it does not convey where the properties will be flattened.

```
[ aConfiguration flattenProperties: aDictionary
```

Better names such as `flattenPropertiesFrom:` and `flattenPropertiesInto:` are much better because there are no ambiguities.

```
[ aConfiguration flattenPropertiesFrom: aDictionary
  aConfiguration flattenPropertiesInto: aDictionary
```

Here are more examples

```
[ changeField: anInteger to: anObject
```

3.5 Guideline: Indicate return types

Prefer

```
[ReadStream on: aCollection.
```

over

```
[Not ReadWriteStream for: aCollection.
```

Prefer

```
[File openOn: stream
```

over

```
[Not File with: stream
```

Prefer

```
[display: anObject on: aMedium
```

over

```
[Not display: anObject using: aMedium
```

3.5 Guideline: Indicate return types

When a method is returning an object (different from the receiver) and that this object is not polymorphic with the receiver it is important to mention it. Since Pharo is not statically typed, we can use the selector name to give such information to the sender of the message.

For example, the method `characterSeparatorMethodSignatureFor:` of the pretty printer did not return a character but a block as shown below:

```
characterSeparatorMethodSignatureFor: aMethodNode
  ^ [
    (self needsMethodSignatureOnMultipleLinesFor: aMethodNode)
      ifTrue: [ self newLine ]
      ifFalse: [ self space ] ]
```

Favor `characterSeparatorMethodSignatureBlockFor:` over `characterSeparatorMethodSignatureFor:` when the method returns block and not a character as `characterSeparatorMethodSignatureFor:` indicates.

A much better design is to rewrite this method and its users to use a character. Returning a block in such situation is overkill and it hampers reusing the method without being forced to send a message.

The following method is corresponding to its name.

```
characterSeparatorMethodSignatureFor: aMethodNode
  ^ (self needsMethodSignatureOnMultipleLinesFor: aMethodNode)
    ifTrue: [ self newLine ]
    ifFalse: [ self space ]
```


A good example is the API of the `FileReference` class. The message path-String indicates clearly that it returns the path as a string while to access the path object the message path should be used.

3.6 **Guideline: Use interrogative form for testing**

When interrogating the state of an object, use a selector beginning with a verb such as `has`, `is`, `does`,...

Prefer

```
[isAtLineEnd
```

over

```
[Not atLineEnd
```

```
[aVehicle hasFourWheels
```

over

```
[Not aVehicle fourWheels
```

3.7 **Guideline: Avoid using parameter and variable name type**

Avoid the parameter type or name in the method name if you are using typed parameter names.

Prefer

```
[fileSystem at: aKey put: aFile
```

over

```
[Not fileSystem atKey: aKey putFile: aFile
```

```
["for semantic-based parameter names"
fileSystem atKey: index putFile: pathName
```

```
["useful when your class has several #at:put: methods"
fileSystem definitionAt: aKey put: definition
```

Prefer

```
[aFace changeTo: expression
```

over

```
[Not aFace changeExpressionTo: expression
```

3.8 Guideline: Use accessors or not

There are different schools about whether to use accessors. In his seminal book Kent Beck discusses it in depth. Here we give a list of arguments for and against and you should decide and follow the conventions of the project you work on. In any case, if you use accessors or not, be consistent.

Arguments in favor of accessors:

- Accessors abstract from the exact state internal representation.
- Accessors may hide that values are derived or not.
- Subclasses may freely redefine the way accessors are implemented.

Arguments against accessor use:

- Accessors expose the internal state of an object.
- When the class is small, using accessors may blow up the number of methods.
- Using refactorings, we can always easily introduce accessors.

3.9 Guideline: Name accessors following variable name

When you use accessors, name them consistently: The getter is name as the variable it refers to. The setter is the same but with an extra terminating colon `:`.

For getter, prefer

```
[ tiles
  ^ tiles
```

over

```
[ getTiles
  ^ tiles
```

Do not use `get` or `set` in accessor selectors!

Watch out.

Pay attention a Setter is just setting a value and just returning (implicitly) the receiver. The following setter definition is not correct.

```
[ BinaryNode >> root: rootNode
  "Set a root node"
  ^ root := rootNode
```

Favor the following one instead:

```
[ BinaryNode >> root: rootNode
  root := rootNode
```

Note that we do not need to comment a basic setter.

For lazy initialization:

```
[ tiles
  ^ tiles ifNil: [ tiles := OrderedCollection new ]
```

For setters

```
[ tiles: aCollection
  tiles := aCollection
```

Put accessors in the 'accessing' protocols. When you have accessors doing extra work place them in a separate protocols to stress their difference.

3.10 **Guideline: Avoid prefixing with set public accessors**

Some developers may be tempted to name setter methods by prefixing the variable name

Prefer

```
[ tiles: aCollection
  tiles := aCollection
```

over

```
[ setTiles: aCollection
  tiles := aCollection
```

Following K. Beck's advice, use `setTitles:` only for private messages to initialize objects from class side methods.

3.11 **Guideline: Use basic or raw to access low-level**

When two methods are needed for the same state variable, e.g., one returning the actual object stored and one returning and raising an event, prefix the one returning the actual object with the word `basic` or `raw`.

```
[ method: aCompiledMethod
  self basicMethod: aCompiledMethod.
  self signal: MethodChanged
[ basicMethod: aCompiledMethod
  method := aCompiledMethod
```

When you have a getter that returns an object and a getter than return a different representation of the same object add a suffix

```
[ path
  ^ path
[ pathString
  ^ self path asString
```

3.12 Guideline: Follow conventions and idioms

When designing new objects, you may mimic some practices that the system uses already with for example dictionaries, sets, etc.

Example. Since a style sheet acts as a dictionary of properties it is much better to use `at:` instead of `get:`, especially if you define the message to set a value to a property as `at:put:` and not `set:`.

Prefer

```
[ stylesheet at: #fontColor
```

over

```
[ stylesheet get: #fontColor
```

Prefer

```
[ aCollection groupedBy: [ :each | each odd ]
```

over

```
[ Not aCollection groupBy: [ :each | each odd ]
```

Prefer

```
[ series at: #k3 put: 'x'.
```

over

```
[ Not series atKey: #k3 put: 'x'
```

Prefer

```
[ aCollection at: #toto
```

over

```
[ Not aCollection atKey: #toto
```

3.13 Guideline: Distinguish class vs. instance selectors

When defining a class method, we may name it the same way as an accessor of the class. Such practice hampers code readability in the sense that it is difficult to identify rapidly class methods. The senders will report both the instance and class usage. You may think that you will identify the message because the receiver is a class or an instance but there are many situations

were this is not the case. So it's better to enrich the class method with a distinct word.

Example

In Pillar, annotations have parameters and the accessor method `parameters:`. Now, in some version, an instance creation method with the same name than the accessor method.

Reading the code of the parser, it is not clear whether `array second` is a class or an instance.

```
[ annotation
  ^ super annotation
  ==>
  [ :array | array second parameters: (array third ifNil: [
    SmallDictionary new ]) ]
```

To create an instance, it is better to name the method with `withParameter:`. This way we can immediately spot that the second element is a class.

```
[ annotation
  ^ super annotation
  ==>
  [ :array | array second withParameters: (array third ifNil: [
    SmallDictionary new ]) ]
```

```
PRAbstractAnnotation class >> withParameters: aCollection

| parameters |
parameters := self checkKeysOf: aCollection.
^ self new
  hadAllKeys: aCollection = parameters;
  parameters: parameters;
  yourself
```

is better than

```
PRAbstractAnnotation class >> parameters: aCollection
| parameters |
parameters := self checkKeysOf: aCollection.
^ self new
  hadAllKeys: aCollection = parameters;
  parameters: parameters;
  yourself
```

Now if you favor a fluid interface with many parameters, using `withParameters:` may not be good.

3.14 **Guidelines: Follow existing protocols**

Protocols are ways to sort methods. It is important to place your methods in adequate protocols since it will ease future exploration of your class.

Pharo provides auto categorisation of protocol for the common methods. So use it as much as possible. When you override your specific methods, place them in similar protocols.

CHAPTER 4

Comments

Comments are important. Comments tell readers that they are smart guys and that they correctly guessed your intentions or your code. Do not believe people that say that methods do not need comments. Obviously here is what they mean:

1. obvious methods such as accessors do not need comments,
2. a good comment is not describing in English how the code executes,
3. it is better to split long methods into smaller ones with a single responsibility,
4. but a good comment is always welcome because it reinforces the understanding of the reader.

A comment should be adapted to the level of granularity (i.e., package, class, method) to which it applies.

4.1 Guideline: Method comments

Method comments should contain sufficient information for a user to know exactly **how to use** the method, **what** the method does including any side effects, and **what it answers** without having to look at the source code. Imagine that the source code is not available.

The main method comment is not about its implementation. Do not rephrase the implementation. The second level comments can include information about the implementation. Insert a new line to separate method comments from the method body.


```

Collection >> asCommaString
  "Return collection printed as 'a, b, c' "
  "#('a' 'b' 'c') asCommaString >>> 'a, b, c'"

  ^ String streamContents: [:s | self asStringOn: s delimiter: ',
  ' ]

```

The comments of a method should typically include:

1. the method purpose (even if implemented or supplemented by a subclass)
2. the parameters and their types
3. the possible return values and their types
4. complex or tricky implementation details
5. example usage, if applicable, as a separate comment

Finally accessors do not need comments; the only comment that accessor could have is the purpose of the instance variable.

```

day
  "Answer number of days (an instance of Integer) from
  the receiver to January 1, 1901."

  ^ day

```

4.2 Guideline: Avoid relying on a comment to explain what can be reflected in code

Good Pharo source code is self-documenting, often making comments on statements redundant. Statements need only be commented to draw the reader's attention. If the source code implements an algorithm that requires explanation, then the steps of the algorithm should be commented as needed.

Do not comment an obvious fact that is expressed simply as plain code.

Prefer

```

| result |
result := self employees
  collect: [:employee | employee salary > amount].

```

over

```

| result |
"Store the employees who have a salary greater than in result."
result := self employees
  collect: [:employee | employee salary > amount].

```

4.3 Guideline: Use active voice and short sentences

When writing comments, use active voice and avoid long and convoluted sentences. A method comment should state what the method does, its arguments, its effects and output.

```
"Active voice"
createShell
  "Create the receiver's shell. Hook the focus callback."

Not "Passive voice"
createShell
  "The receiver's shell is created. The focus callback is hooked."
```

4.4 Guideline: Include executable comments

Pharo offers executable examples in comment using the message >>>. Executable examples in comments are super cool because as the reader you can execute the code and understand the parameters. In addition the documentation is always synchronized because tools such as the test runner can check that examples are correct.

```
ProtoObject >> ifNil: nilBlock ifNotNil: ifNotNilBlock
  "If the receiver is not nil, pass it as argument to the
   ifNotNilBlock block
   else execute the nilBlock block "

  "(nil ifNil: [42] ifNotNil: [:o | o + 3 ] ) >>> 42"
  "(3 ifNil: [42] ifNotNil: [:o | o + 3 ]) >>> 6"

  ^ ifNotNilBlock cull: self

Object >> split: aSequenceableCollection
  "Split the argument using the receiver as a separator."
  "optimized version for single delimiters"
  "($/ split: '/foo/bar')>>>#(' 'foo' 'bar') asOrderedCollection"
  "[:c| c isSeparator] split: 'aa bb cc dd') >>> #('aa' 'bb' 'cc'
   'dd') asOrderedCollection"

  | result |
  result := OrderedCollection new: (aSequenceableCollection size /
    2) asInteger.
  self split: aSequenceableCollection do: [ :item |
    result add: item ].
  ^ result
```

4.5 Guideline: Use CRC-driven class comments

A class is not in isolation, but *implements* responsibilities (mainly one) and *collaborates* with other entities. Therefore a class comment should be composed of at least 3 parts: the class, its responsibilities and how it uses its collaborators. The Class Responsibility Collaboration (CRC) pattern is powerful to design but also to comment classes. Use it for commenting class.

Knowing the instance variables is the least importance! Follow the template given by Pharo that is shown below.

```
Please comment me using the following template inspired by Class
    Responsibility Collaborator (CRC) design:

For the Class part:
    State a one line summary. For example, "I represent a paragraph of
    text".

For the Responsibility part:
    Three sentences about my main responsibilities - what I do, what I
    know.

For the Collaborators Part:
    State my main collaborators and one line about how I interact with
    them.

Public API and Key Messages
- message one
- message two
- (for bonus points) how to create instances.

One simple example is simply gorgeous.

Internal Representation and Key Implementation Points.

Implementation Points
```

4.6 Guideline: Comment the unusual

When a behavior is unusual, performing unexpected actions or using an unexpected algorithm, it is important to comment it. In general comments should make irregular and unusual aspects clearer. You may want to include implementation-dependent, or platform specific idiosyncrasies.

About code formatting

Code formatting improves code comprehension. Again follow the conventions.

5.1 **Guideline: Be consistent**

One important guideline when writing code is to follow conventions and in addition to be consistent. Systematically apply a formatting style. Keep the violations to conventions to a minimum.

5.2 **Guideline: Use the general method template**

- Separate method signature and comments from method body with an empty line.
- Add an extra tab to the comments.
- Add an extra line to stress the beginning of the method body.
- Use a tab to separate the method body from the left margin.

```
message selector and argument names  
    "A comment following the guidelines."  
  
    | temporary variables |  
    statements
```

For example:

```

addLast: newObject
    "Add newObject to the end of the receiver. Answer newObject."

    lastIndex = array size ifTrue: [ self makeRoomAtLast ].
    lastIndex := lastIndex + 1.
    array at: lastIndex put: newObject.
    ^ newObject

```

Do not let space before the first word of the comment, align comment with method body, make sure that the reader can identify the beginning of the method body by using an empty line.

Prefer the following

```

collectionNotIncluded
    "Return a collection for wich each element is not included in
    'nonEmpty'"

    ^ collectionWithoutNil

```

over:

```

collectionNotIncluded
    " return a collection for wich each element is not included in
    'nonEmpty' "

    ^ collectionWithoutNil

```

and over:

```

collectionWithoutEqualElements

    " return a collection not including equal elements "
    ^collectionWithoutEqualElements

```

5.3 Guideline: Indent method body

Use indentation to convey structure! Do not glue everything on the left margin.

Don't indent your method like this:

```

initialize
super initialize.
symbols := Bag new.
names := Set new

```

Prefer

```

initialize

    super initialize.
    symbols := Bag new.
    names := Set new

```

5.4 Guideline: Separate signature and comments from method body

Separating method comments from the method implementation favor focusing our understanding to the right level. When we want to understand what the method does, we just have to read the comments. When we want to understand how the method is implemented, we just read the method body.

Prefer

```
performCrawling: aName
  "Takes the last word in uppercase as a symbol and eventually add
   it to the bag symbols"

  name := aName copy.
  self getUpperCase.
  self stemSymbolFrom: aName.
  self toUpperCase.
  ^ symbol
```

over

```
performCrawling: aName
  "Takes the last word in uppercase as a symbol and eventually add
   it to the bag symbols"
  name := aName copy.
  self getUpperCase.
  self stemSymbolFrom: aName.
  self toUpperCase.
  ^ symbol
```

5.5 Guideline: Use space to give breath to code

Gluing all the characters together slows down reading. The reader needs to separate expressions. Gluing characters also hampers the identification of logical groups such as conditional branches.

Put horizontal space to make it easier to read code and clearly identify variables, arguments, assignments, block delimiters and returns.

Prefer

```
stemSymbolFrom: aName

  | stemmer symbol |
  stemmer := SymbolStemmer new.
  symbol := stemmer performCrawling: aName.
  ^ symbol
```

Over

```
stemSymbolFrom:aName
    |stemmer symbol|
    stemmer:=SymbolStemmer new.
    symbol:=stemmer performCrawling:aName.
    ^symbol
```

Parentheses do not need spaces (after (and before)) since they show that an expression fits together. But favor space for [and], since they may contain complex expressions.

```
drawOnAthensCanvas: aCanvas bounds: aRectangle color: aColor
    (self canDrawDecoratorsOn: aCanvas) ifFalse: [ ^ self ].
    self drawOnAthensCanvas: aCanvas.
    next drawOnAthensCanvas: aCanvas bounds: aRectangle color: aColor
```

5.6 Guideline: Align properly

Understanding that a piece of code is a coherent expression eases understanding of more complex expressions.

Make sure that your indentation reinforce the identification of block of functionality.

Prefer

```
self phoneBook add:
    (Person new
     name: 'Robin';
     city: 'Ottawa';
     country: 'Canada').
```

Over

```
self phoneBook add:
    (Person new
     name: 'Robin';
     city: 'Ottawa';
     country: 'Canada').
```

5.7 Guideline: Use tabs not spaces to indent and use spaces to help reading

When navigating from one element to the next one, spaces and tabs are the same. Since they do not have a visual representation the reader cannot know in advance if the white space in front of word is a tab or multiple spaces. It is then annoying to handle spaces manually.

5.8 Guideline: Do not break lines randomly

- Avoid extra spaces at the beginning and use tabs to indent.
- Use one space to separate instructions.
- Avoid extra spaces everywhere: one is enough!
- Avoid extra spaces at the end of the line.

Prefer

```
stemSymbolFrom: aName
    | stemmer symbol |
    stemmer := SymbolStemmer new.
    symbol := stemmer performCrawling: aName.
    ^ symbol
```

over

```
stemSymbolFrom: aName
    | stemmer symbol |
        stemmer:=SymbolStemmer new.
    symbol:=stemmer performCrawling: aName.
    ^symbol
```

5.8 Guideline: Do not break lines randomly

White lines attract the eyes and force the reader to ask himself why the code is separated that way. Only separate method signature and comment from method body with a new line.

Prefer

```
paragraph
    "this method is here to find the paragraph in the chain, instead
    of relying on implementing #doesNotUnderstand: !!!"

    | p |
    p := next.
    [ p isNotNil and: [ p isKindOf: RubParagraph ] ]
        whileFalse: [ p := p next ].
    ^ p
```

Over

```
paragraph
    "this method is here to find the paragraph in the chain, instead
    of relying on implementing #doesNotUnderstand: !!!"

    | p |

    p := next.
```



```
[ p isNotNil and: [ p isKindOf: RubParagraph ] ] whileFalse: [
  p := p next.
].
^p
```

5.9 Guideline: Highlight control flow

Help the reader to understand control flow logic of your code by using indentation.

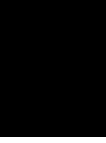
```
size
  "Returns size of a tree - number of nodes in a tree"
  self root isNotNil
    ifTrue: [ ^0 ].
  ^ self size: self root
```

Over

```
size
  "Returns size of a tree - number of nodes in a tree"
  self root isNotNil
    ifTrue: [ ^0 ].
  ^self size: self root.

depth:aNode
  "Returns depth of a tree starting from the given node"
  | leftDepth rightDepth |
  leftDepth := -1.
  aNode leftChild isNotNil
    ifTrue: [ leftDepth := self depth: aNode leftChild ].
  rightDepth := -1.
  aNode rightChild isNotNil
    ifTrue: [ rightDepth := self depth: aNode rightChild ].

  ( leftDepth > rightDepth )
  ifTrue: [ ^ (1 + leftDepth) ]
  ifFalse: [ ^ (1 + rightDepth) ].
```



Powerful coding idioms

Some coding idioms will make your code a lot clearer. Knowing them is also good because you will code faster.

6.1 **Guideline: Do not query twice for the same object**

The message `ifNotNil:` expects a block with one argument. This argument is the object that is not nil.

For example,

```
[self doThat ifNotNil: [ :that | self doSomethingWith: that ]
```

is better than:

```
[ | that |  
  that := self doThat.  
  that ifNotNil: [self doSomethingWith: that]
```

Similarly have a look at the messages containing the `ifPresent:` variations.

```
[aCol at: key ifPresent: [ :present | self doSomethingWith: present]
```

6.2 **Guideline: Move returns outside branches**

When two branches of a condition are returning a value, better move the return out of the blocks.

Prefer

```
depth: aNode  
  "Returns depth of a tree starting from the given node"  
  ...  
  ^ leftDepth > rightDepth  
    ifTrue: [ 1 + leftDepth ]  
    ifFalse: [ 1 + rightDepth ]
```

over

```
depth:aNode  
  "Returns depth of a tree starting from the given node"  
  
  ^ leftDepth > rightDepth  
    ifTrue: [ 1 + leftDepth ]  
    ifFalse: [1 + rightDepth ]
```

Potential traps

Understanding possible mistakes is a nice way to avoid them or to spot errors made in your code. Here are some common mistakes.

7.1 **Guideline: Use parentheses to disambiguate messages with the same priority**

For keyword-messages

The Pharo compiler does not know where to cut an expression composed on multiple keyword-based messages. For example, `assert:includes:` in the expression `self assert: uUMLClass variables includes: 'name'` can be a message that the object `self` can understand. For example testcases understand the message `assert:equals:` and the following expression is fully valid: `self assert: uUMLClass variables equals: 'name'`.

Therefore, this is the programmer responsibility to use parentheses to separate correctly the messages having the same priority. The following example illustrates this point.

```
testDefineASimpleClass
| uUMLClass |
uUMLClass := UMLClass named: 'ComixSerie'.
uUMLClass instVar: 'name'.
self assert: uUMLClass variables includes: 'name'
```

There is no message `assert:includes:.` The expression `uUMLClass variables includes: 'name'` should be parenthesized, because this is the re-

sult of the execution of this expression that should be passed as argument of the message `assert:`.

```
testDefineASimpleClass
| uUMLClass |
uUMLClass := UMLClass named: 'ComixSerie'.
uUMLClass instVar: 'name'.
self assert: (uUMLClass variables includes: 'name')
```

Between binary messages

Pharo does not make any assumption about the possible mathematical meaning of messages. As a programmer you cannot describe the weight of a binary messages. It means that in an expression composed of multiple binary messages, they will be executed from left to right.

For example `1 + 2 * 3` returns 9 since first the message plus is resolved and its result is the receiver of the message `*`.

```
1 + 2 * 3
>>> 9
```

To get the correct mathematical behavior, one should use parentheses.

```
1 + (2 * 3)
>>> 7
```

7.2 Guideline: no need for extra parentheses

There is no need for parentheses surrounding unary message. There is not much benefit to add parentheses around unaray messages. In Pharo unary messages are the messages that have the highest priority. They are executed first.

Prefer

```
xMatrix := PMMatrix rows: x asArrayOfRows.
```

over

```
xMatrix := PMMatrix rows: ( x asArrayOfRows ).
```

No parentheses around message with higher priority

In the similar way, there is no need to put parentheses around binary messages involved in keyword-based expressions. Binary messages are executed prior to keyword-messages. In the following `=` is executed before `ifTrue:`.

Prefer

```
[ reducedX do: [ :row |
  (row at: 'target') = 'Iris-setosa'
    ifTrue: [ a add: row asArray ] ].
```

over

```
[ reducedX do: [ :row |
  ( (row at: 'target') = 'Iris-setosa')
    ifTrue: [ a add: (row asArray) ] ].
```

Prefer

```
depth: aNode
  "Returns depth of a tree starting from the given node"

  | leftDepth rightDepth |
  leftDepth := -1.
  aNode leftChild isNotNil
    ifTrue: [ leftDepth := self depth: aNode leftChild ].
  rightDepth := -1.
  aNode rightChild isNotNil
    ifTrue: [ rightDepth := self depth: aNode rightChild ].
  ^ leftDepth > rightDepth
    ifTrue: [ 1 + leftDepth ]
    ifFalse: [ 1 + rightDepth ]
```

over

```
depth:aNode
  "Returns depth of a tree starting from the given node"
  | leftDepth rightDepth |
  leftDepth := -1.
  aNode leftChild isNotNil
    ifTrue: [ leftDepth := self depth: (aNode leftChild) ].
  rightDepth := -1.
  aNode rightChild isNotNil
    ifTrue: [ rightDepth := self depth: (aNode rightChild) ].

  ( leftDepth > rightDepth )
  ifTrue: [ ^ (1 + leftDepth) ]
  ifFalse: [ ^ (1 + rightDepth) ].
```

No parentheses around variable

Putting parentheses around a variable does not produce an array, it has no effect. Do not confuse parentheses and curly braces. Curly braces is a shortcut to produce an array with the elements they surround: { a } produces an array with one element whose value is the value held by the variable a as shown by the examples below.

TIn this code snippet, {} creates an array.

```
[ | a |
  a := 12.
  {a} printString
  >>> #(12)
```

In this snippet, the parentheses do not do anything.

```
[ | a |
  a := 12.
  (a) printString
  >>> 12
```

No parentheses around single message

There is no need to put extra parentheses over a single message. It has no effect. Parentheses make sense to disambiguate one message over a set of messages composing an expression.

Prefer

```
[ m := pca transform: xMatrix

over

[ m := (pca transform: xMatrix)
```

7.3 Guideline: receiver of ifTrue:ifFalse: is a boolean

Do not use a block as receiver of a ifTrue:, ifFalse:, ifTrue:ifFalse: or ifFalse:ifTrue: messages.

The following expressions does not work

```
[ [lastNode =0] value
  ifTrue:[ lastNode := curNode ]
  ifFalse:[ lastNode next: curNode ]

[ [lastNode =0]
  ifTrue:[ lastNode := curNode ]
  ifFalse:[ lastNode next: curNode ]
```

The correct is the following

```
[ lastNode = 0
  ifTrue:[ lastNode := curNode ]
  ifFalse:[ lastNode next: curNode ]
```

7.4 Guideline: receiver of whileTrue: is a block

The receiver of the message whileTrue: is a block, and its argument is, too.

7.5 Guideline: Use a Block when you do not know execution time

The following line is incorrect:

```
[ (number < limit) whileTrue: [ do something ]
```

The following line is correct:

```
[ [ number < limit ] whileTrue: [ do something ]
```

7.5 Guideline: Use a Block when you do not know execution time

Often newcomers get confused about when to use () and []. A good way to understand is that we should use [] when we do know whether an expression will be executed (may be multiple times).

ifTrue:ifFalse:

The conditional is always executed, while each of the arguments is a block because we do not know which ones will be executed.

```
[ lastNode = 0
  ifTrue: [ lastNode := curNode ]
  ifFalse: [ lastNode next: curNode ]
```

timesRepeat:

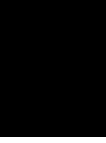
timesRepeat:'s argument is a block because we do not know how many times it will be executed.

```
[ n timesRepeat: [ lastNode := curNode next ]
```

do:/collect:

The argument of iterators such as do:, collect:,... is a block because we do not know how many times (if any) the block will be executed.

```
[ aCol do: [ :node | ...]
```

Conclusion

Remember that you write code once and will read it a thousand times. Take the time to give good names. However finding good names is not an easy task, but you can use refactorings to improve things easily. This goes in pair with tests. You write a test once and it gets executed million times. Therefore, write tests to exercise the names you use and change them until they help you telling stories that can be understood.

Bibliography

