Pharo with Style

Stéphane Ducasse

December 30, 2018

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

Programming is a lot more than just writing algorithms or programs. Programming is all about communication. Communication with others: the other programmers that we will participate to your development effort but also with yourselves. 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 fundation 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.

S. Ducasse - 30 December 2018.

CHAPTER 1

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 culture so use simple, mainstream, and common terms. Avoid hidden or implied meaning that can only be understood by a limited group of people Make information explicit.

1.2 Guideline: Avoid underscores and favor camelcase.

Prefer

```
timeOfDay
```

over

```
Not timeofday
Not time_of_day
```

Prefer

```
GnatXmlNode
```

over

```
Not GNAT_XML_Object
```

When creating private low level methods binding to external C-libraries you may wanto 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 CamelCase

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

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

Still Pharo favors camelCase so use it systematically for words.

Prefer

```
maxLimit
MaxLimit
```

over

```
Not maxlimit
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.

Prefer

```
superclass
```

over

```
Not superClass
```

1.4 Guideline: Use descriptive names

Choose descriptive names that capture domain entity 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 speaker often misplaced word qualifier. 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
```

From that perspective favor words with unique pronunciation. About `readSize`: does this mean that the size was just read (red) or it is the size to read (reed)?

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 ]
```

Here the method selector is not good. It should be `gatherSelectorsFrom:` or something similar.

1.7 Guideline: Follow domain

Follow domain and culture of the project. Do not invent your own because you think it is better. Favor regularity 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 convention. The name should convey that the class is a abstract syntax tree node hence `GnatXmlNode` is much better that `GnatXmlObject`.

In Moose, an importer is an object creating FAMIX entities (classes methods....) from the datastructure representing a language usually an Abstract Syntax Tree. Therefore `GNATInstaller` which creates entities from AST should be renamed `GnatImporter` and `GNATImporter` which load AST in memory should be renamed `GnatASTLoader`.

1.8 Guideline: Shared variables starts with uppercase

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

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

1.9 Guideline: Private variables starts with lowercase

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

```
address
classExtensionSelectors
classTags
```

1.10 Guideline: Unique pronunciation

Choose names that have a unique pronunciation.

Prefer

```
sizeToRead
sizeJustRead
```

```
Not readSize
```

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

1.11 Guideline: Class names should exhibit their parent

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

Below 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
```

Here not mentioning the Presenter suffix makes the reader unclear about the fact that it is a Presenter object by opposition to a view object. Prefer

```
ApplicationWithToolBarPresenter
```

over

```
Not ApplicationWithToolbar
```

Here `DynamicWidgetChange` does not conver the fact that this is not a *domain* object representing a change but a Presenter object in the Model View Presenter triad.

```
DynamicWidgetChangePresenter
```

over

```
Not DynamicWidgetChange
```

1.12 Guideline: Avoid name collision

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 in presence of namespace you will have to pay attention that your namespace name does not collide with another one.

CHAPTER 2

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 finding 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 to use 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
```

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

2.2 Guideline: Use typed variables to indicate API

Suppose a String, a Symbol, and nil are valid. A developer may be tempted to use the name `aStringOrSymbolOrNiI`; however, most developers choose `aString` or `anObject`. `anObject` is a better choice with an accompanying comment that says, "anObject can be a String or aSymbol"

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

Prefer

```
properties: aDictionary
```

over

```
Not properties: aCollection
```

You may also want to stress a specific API.

Prefer

```
properties: aPuttable
```

over

```
Not properties: aCollection
```

2.3 Guideline: Get the best from semantic and type variables

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 if False: 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
```

CHAPTER 3

Selectors

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

3.1 Guideline: Choose selectors to form little 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 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
```

over

```
Not aFace surprised
```

```
skipSeparators
```

Pay attention that some word can be interpreted as interrogative while you want to give them an imperative meaning. For example compare

```
optimized
```

and

```
triggerOptimization
```

3.3 Guideline: Indicate flow with preposition

When state process is going from one object to another one, 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 is no ambiquities.

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

Here are more examples

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

Prefer

```
ReadWriteStream 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
```

3.4 Guideline: Use interrogative form for testing

Use an 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
```

3.5 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
[[[type=not
Not fileSystem atKey: aKey putFiIe: aFile
```

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

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

Prefer

```
aFace changeTo: expression
```

over

```
Not aFace changeExpressionTo: expression
```

3.6 Guideline: Accessors

There are different schools about the use of accessors or not. If you use accessors, use them consistently. Name them also consistently:

For getter, prefer

```
tiles
  ^ tiles
```

over

```
getTiles
  ^ tiles
```

Do not use get in accessor selectors!

For lazy initialization:

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

For Setters

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

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

3.7 Guideline: Use basic or raw to access low-level

When two methods are needed for the same state variable, for example 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
```

3.8 Guideline: Follow conventions and idioms

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

Example. Since a stylesheet acts as a dictionary of properties it is much better to use `at:` instead of `get:` especially when the message to set a value to a property is `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
```

3.9 Guideline: Distinguish class vs. instance selectors

When defining a class method, we may name it the same way than an accessor of the class. Such practice hampers code readibility in the sense that identifying rapidly class methods is then difficult. 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 better enriched the class method with a distinct word.

Example

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

In the following code it is difficult to understand fast the code: 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 is better to name the method `withParameter:`. This way we can immediatelly 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.

CHAPTER 4

Comments

Comments are important. Comments say to the reader, yes you are a smart guy and you guessed right this is what we are doing.

Do not believe people that say that methods do not need comments. Obviously they are meaning that:

1. obvious methods such as accessors do not need comments,
2. a good comment is not describing in english how the code execute,
3. it is better to split long methods into smaller one with a single responsibility,
4. but a good comment is always welcomed 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 is apply.

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 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 sub-class)
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 could be reflected in the 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 obvious facts that is expressed simply as plain code.

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

```
Not
| 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 circumvoluted 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 synchronised because the tools can check that the example 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 implement responsibilities (mainly one) and collaborate 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. Class Responsibility Collaboration (CRC) pattern is powerful to comment classes. Use it for commenting class.

Knowing the instance variables is the least importance! Follow the template given by Pharo and show hereafter.

```
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 that is unusual, performing unexpected actions or using an ununexpected algorithms, it is important to comment it. In ganeral comment should make irregular and unusual aspects clearer. You may want to include implementation-dependent, or platform specific idiosynchrasies.

CHAPTER 5

About Code Formatting

Code formatting improves code comprehension. Again follow the convention.

5.1 Guideline: Be consistent

One important guideline when writing code is to follow conventions and in addition to be consistent. Apply systematically a formatting style. Keep the violations to convention to the 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 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
```

5.3 Guideline: Indent method body

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

```
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

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

Put space to help reader reads code and identify clearly variable, arguments, assignment, block delimiters and return.

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 since they show that an expression fits together but favor space for [and].

```
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

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 tab not space to indent and use space to help reading

Avoid extra spaces everywhere. Use one space to separate instructions. Avoid extra space at the end of the line. Avoid extra space at the beginning and use tab to indent.

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 line 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
```

CHAPTER 6

Powerful coding idioms

Some coding idioms will make your 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.

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

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

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

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

CHAPTER 7

Potential traps

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

7.1 Guideline: Use parentheses to disambiguate keyword messages

```
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.

```
testDefineASimpleClass

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

7.2 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 ]
```

7.3 Guideline: receiver of whileTrue: is a block

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

The following line is incorrect:

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

The following line is correct:

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

7.4 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 | ...]
```