Understanding Pharo’s global state to move programs through time and space
Guillermo Polito, Noury Bouraqadi, Stéphane Ducasse, Luc Fabresse

To cite this version:
Guillermo Polito, Noury Bouraqadi, Stéphane Ducasse, Luc Fabresse. Understanding Pharo’s global state to move programs through time and space. International Workshop on Smalltalk Technologies, Aug 2014, Edinburgh, United Kingdom.

HAL Id: hal-01070964
https://hal.archives-ouvertes.fr/hal-01070964
Submitted on 2 Oct 2014

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.
Understanding Pharo’s global state
to move programs
through time and space

Guillermo Polito, Noury Bouraqadi, Stéphane Ducasse, Luc Fabresse
Mines-Telecom Institute, Mines Douai
RMOD INRIA Lille Nord Europe

Abstract
Code mobility is a mechanism that allows the migration of running programs between different environments. Such migration includes amongst others the migration of application data and resources. Application’s data is usually composed by elements of different nature: from printers and files, to framework and domain objects. This application data will be transported along with the code of its program in space (when serialized and deployed in another environment) or time (when a new session is started in a different point of time). The main problem when moving around code resides, in our understanding, to global state. While un REFERENCED leaf objects are garbage collected, those referenced (transitively) by some global object will remain alive.

In order to support code mobility in time and space, we need to understand how global application data is used. With this purpose, we study and classify Pharo’s global state. This classification uncovers some common patterns and provides a first insight on how global state should be managed, specially in code mobility scenarios. As a minor contribution, we also discuss solutions to each of the found categories.

1. Introduction
Code mobility is a mechanism that allows the migration of programs between different environments. It provides support for e.g., load balancing, adjusting an application’s resources dynamically and functionality customization. Fuggetta et al. define informally code mobility as the capability to rebind a piece of code with the location it is running [? ]. Such rebinding may consist, depending on the style of mobility, in the mobility of execution state, application data and resources, or both of them. Execution state mobility is the ability to suspend the actual execution of a program and transfer its internal execution information (e.g., code, execution stacks, instruction pointers) to some other environment. Data mobility is the ability to transfer the application’s data (e.g., objects, database connections, files) between different environments.

Application data is usually composed by elements of different nature. Files are used for configuration and logging. Network connections such as sockets are used to communicate with remote systems. External libraries provide with code reuse. We can also find objects local to the application, of two different categories: domain objects modeling the application’s specific concerns and application objects modeling those concerns that are cross-cutting between applications.

In our experience manipulating the language kernel of Pharo, we identified several cases where data mobility presents some issues. We can generalize those issues as mobility either in time (i.e., creating or recreating a program), or in space (i.e., moving a program between different environments):

Transporting code in space. When moving a program from one environment to another one, some of its state becomes invalid. For example, files existing in one machine will not exist in some other. Because of this, the migration mechanism should be aware of the state it migrates, to either reinitialize it, re-bind it in the new environment, or by keep it with its same value [? ].

Transporting code in time. Image-based systems allow one to persist the state of a program to restart it at some other point of time from the last check-point, introducing the idea of program sessions: every time the system is restarted, a new session is started. These programming sessions introduce the concern of session specific state i.e., state that is only valid during a programming ses-
sion. The mechanism in charge of stopping and restarting the image has to recognize the session specific state to re-initialize or rebind it every time a new session is started.

Creating for the first time. The initial creation of the system is a combination of transporting the program in space and time, since the issues of both appear in it. When creating or recreating the language kernel from scratch, for example during a bootstrap process [?], we must deal with its initialization. All the initial objects must be created, and their state is initialized by either binding it for the first time to some resource or assigning it some value. This state should be initialized in a proper order.

One of the main problems when moving code around resides in the existence of global state. While unreferenced leaf objects are garbage collected, those referenced (transitively) by some global object will remain alive. Because of this, we focus our attention on global state (cf. Section 2). Migrating global state in the cases described above in a generic way shows itself challenging (cf. Section 3). Global state is used for many different and unrelated purposes in the Pharo base libraries e.g., from caches to constants values. Also, the intention of such usage is not explicit in the source code: its identification requires the developer to read the complete implementation.

In this paper we present an empirical study on the global state of Pharo base libraries. Our contribution is twofold:

• We present a classification of the usage of global state, identifying patterns built with global state constructs in Pharo (cf. Section 4).
• We discuss our findings and solutions to the issues we found. Our main goal with this is to make explicit those patterns. In such a way, client libraries and frameworks in charge of program migration can be simplified. (cf. Section 6 and Section 5).

2. Background

Global state is a simple and handy mechanism to share state between different objects. It is also a simple persistency mechanism: state hold by it will persist as long as the program is alive and running. Additionally, in image-based systems as Pharo it will remain alive through different program executions because the image persists its state taking as root the global objects. In this paper we put focus on global state because of this persistency property.

Global state is indeed not a bad mechanism per se, and is often used in applications to implement globally needed concerns. For example, Pharo implements through it a global process scheduler and the system dictionary holding all classes. However, its usage is discouraged in general terms because it introduces hidden dependencies in the software it is used.

2.1 Global State in Pharo

Global state in Pharo can be expressed in many forms with many constructs of the language. In this paper we will focus on the elements we present following. Note that equivalent language constructs can be found in other languages such as Java (for example, with static variables).

Global Variables. Global variables are variables that share their values to all objects in the system. A global variable can be accessed from any method, from any object. In Pharo, these kind of variables are stored in the global SystemDictionary object. Global variables may reference either (a) global instances such as Processor or Smalltalk, either (b) Classes and Traits.

Class Variables. Class variables are variables that belong to a class. These variables can be accessed by both classes and instances from the hierarchy below its owner class. Their value is shared between all the objects that can access them. In Pharo, these kind of variables are stored in a Dictionary object in its owner class.

Class Instance Variables. Class instance variables are instance variables of the classes. Their value is not directly accessible from subclasses and subinstances of the class. However, they are often made globally accessible with accessors.

Shared Pools. Shared pools are sets of class variables shared amongst many classes. Their values are accessible to all classes (and their instances) that import the shared pool. In Pharo, Shared Pools as implemented as classes treated specially by the compiler at binding time.

Method Literals. Each method contains a collection of those literal objects used in e.g., strings, literal arrays or numbers. As classes are globally accessible, their methods are too, and so their literals.

2.2 About the State in Image-Based Systems

Pharo, as a Smalltalk inspired language, is an image-based language such as Lisp. Image-based languages present the following two main properties: direct object manipulation and persistence. Direct object manipulation provides with instant feedback during development and a flexible way to understand the state of applications. Persistence allows one to store those changes made by direct object manipulation without the need of recreating the system every time it is started. Indeed, an image-based language can be persisted and restarted later on, possibly in another machine. We refer as a session to the time elapsed between the startup of an image and its shutdown.

These programming sessions have session specific state, i.e., state that is valid only within a session. For example, we can name as such file and socket descriptors, handles to external libraries, operating system information, and time and date information. These kind of objects become invalid
when their session is finished. Using them in an invalid state may lead to unexpected behavior, exceptions and virtual machine crashes. The language runtime must ensure that this state is correctly handled on session startup and shutdown: e.g., reinitialize it or discard it.

3. Motivation
In this section we show why understanding and making explicit the usage of global state is important. We introduce first an example based on two Pharo’s cache implementations, and their problems. Then, we explore three different situations in which those problems are made more evident.

3.1 Problems on Global State Usage: an Example
To exemplify the problems on global state usage, we present here two different global cache implementations we find in Pharo 3.0. First, in Figure 1, we present a simplified version of the AST cache. Second, in Figure 2, we find an extract of the HelpIcons class, with the code related to an icon cache. By looking at these two ad-hoc implementations of caches, we identify the following issues:

Incompleteness. Both cache implementations were written to solve only particular issues. The AST cache presents weak references as it inherits from the WeakIdentityKeyDictionary class and also presents methods to be flushed. The icon cache does not present code for any of those features. None of them cover some concerns a cache may want to address such as specifying a maximum amount of elements or a recycling strategy (LRU, FIFO, etc.).

Non-Explicitness. In order to identify the examples as caches we need to read their code: the names of the classes and variables gives us an idea of its responsibility as caches. The default method in the AST cache hints us about having found also a singleton. This problem uncovers the existence of hidden information in the system. One cannot query the system to, for example, obtain a list of the existing caches in order to flush them, or make a report on their memory usage.

3.2 Creating Programs from Scratch: Bootstrapping
While bootstrapping Pharo [?], we must initialize the image’s global state. We observed the need for an order in this initialization, showing off a hidden coupling between code pieces. For example, some global tables must be initialized before initializing the classes state, which in turn must be initialized before the rest of the language kernel (i.e., the startup and shutdown lists, the main processes, etc.).

Since the global state language constructs are used for different concerns implicitly, it is difficult to discern whether they are responsibility of the language kernel, of basic libraries such as Collections, or other not-basic ones such as Networking. This makes the bootstrap process difficult to maintain. A lot of ad-hoc code should be written to handle

```
| WeakIdentityKeyDictionary subclass: #ASTCache |
| classVariableNames: 'Default'. |
| ASTCache>>at: aCompiledMethod |
| ^ self |
| at: aCompiledMethod |
| ifAbsentPut: [ self newASTFor: aCompiledMethod ]. |
| ASTCache>>newASTFor: aMethod |
| "creation of the AST..." |
| ASTCache>>reset |
| self removeAll. |
| ASTCache class>>default |
| ^ Default ifNil: [ Default := self new ]. |
| ASTCache class>>shutdown |
| self default reset. |
```

Figure 1. Simplified code of Pharo’s AST cache.

```
| Object subclass: #HelpIcons |
| classVariableNames: 'Icons'. |
| HelpIcons>>icons |
| ^ Icons ifNil: [Icons := Dictionary new] |
| HelpIcons>>iconNamed: aSymbol |
| ^ self icons at: aSymbol ifAbsentPut: [self perform: aSymbol] |
| HelpIcons>>refreshIcon |
| ^ "creates a new icon object" |
```

Figure 2. Code of Pharo’s Help Icon class with an icon cache.

the dependencies between the global state in Pharo’s language kernel.

3.3 Transporting Programs in Time: Session Awareness
Image-based systems introduce the concern of session specific state. State holding references to for example, files, caches, or platform specific information, may become invalid when a new session is started in a different moment. Pharo presents a startup and a shutdown mechanism to support this. The language runtime raises events on its startup and shutdown. Classes subscribed to such events are notified and will execute some code according to the event. The handler of these events is responsibility of the class developer. This mechanism hides information in two different levels:

Dependencies between classes. The subscribed classes receive the startup and shutdown events in an explicitly defined order. This order is present in a list which is defined by the developers. This list express a dependency between classes e.g., some classes must receive the startup event before others to satisfy its invariants. However, this
list does not actually express the reason of this dependency i.e., which is the state or invariants that should be guaranteed before each class receives the proper event.

**Semantics of class state.** The startup and shutdown event handlers, which are in charge of the clean-up and reinitialization of some of the global state, are written in an imperative fashion. This imperative fashion hides the semantics and invariants of this state.

This hidden information makes difficult to change the startup and shutdown mechanism. Some questions appear when doing so: Can we remove some class from these lists? Can we alter the order without changing the behavior? When we register a new class, in which position should we put it?

3.4 Transporting Programs in Space: Serialization

Migrating objects, and specially code (classes and methods), from one image to another requires in general customizations for the global state it carries and references. References to external classes and global variables may not be serialized but just re-binded in the new environment. Class variables containing constant value objects may be transported with the program. Session specific state should be re-initialized, as program migration implies session change also.

A migration mechanism needs information about the semantics of the state in migration, so it knows whether it should reinitialize it, re-bind it, or keep it as it is. As this information is not usually explicitly available in the program under migration, the developer must add it in the form of extensions or descriptions, external to the program. For example, the serialization library Fuel presents special clusters to handle and customize the serialization of global variables and class variables. The user must customize these clusters externally.

4. Classification of Pharo’s Global State

4.1 Classification Methodology

Our universe of study is the latest release of Pharo, Pharo 3.0. We selected as individuals to study all those usages of global state language constructs as we presented it in Section 2 i.e., class variables and class instance variables, shared pools and global variables. For simplicity, we excluded from our analysis the classes referenced by global variables. We also excluded method literals because analyzing them would mean to read every single method in the language kernel.

The global state in Pharo is present mostly in ad-hoc implementations, making difficult the use of automated methods for its classification. Since the goal of this paper is not to obtain an automatic classification, we built our classification using purely empirical observation: reading the code. We took each of the selected individuals, read all the code related to it and made a qualitative evaluation of it. We put special emphasis on the side-effects on such individuals, which showed useful to recognize the individual’s semantics in the program.

As a result, we distinguished some patterns of usage, which lead us to the categories in Section 4.2. Note that the individuals under study can fall into more than one category e.g., a cache made globally as a singleton. Also, to avoid noise we excluded from the classification those individuals whose role in the source code was very specific, thus they did not conform a representative category.

4.2 Categories

**Constants.** Constants are values that are initialized once and never updated. Pharo has no construct to express constant values. Thus, they are expressed using the other available constructs. This means that the semantics of constants must be ensured explicitly in the code or they are not ensured at all.

**Settings and Configurable Default Values.** Settings and configurable default values provide a single point to configure and share values amongst several instances. They are publicly accessible so they can be modified and customized by developers. Pharo uses settings to store for example maximum size of UI widgets, code completion configurations and network configurations.

**Singletons.** Singletons are well-known objects globally accessible in the system. They are used to provide a single access point to some shared state or behavior. Pharo presents several different singleton implementations: global leaf objects (not classes nor traits) such as e.g., the Processor or the Transcript, leaf objects stored in class variables or class instance variables often accessible through the #uniqueInstance message, and some classes which are indeed used as singletons.

**Caches.** A cache is a buffer that stores duplicated information to reduce the consumption of resources such as CPU or memory. Caches store usually up to a maximum amount of elements, discarding old ones following a given strategy e.g., First In First Out (FIFO) and Least Recently Used (LRU). Pharo presents several caches which store for example images, fonts and package metadata.

**Registries.** A registry stores a list of possible service providers and resolves which one of them is the appropriate to handle a task. They are usually used as a factory, to decouple the users of a service from a particular implementation. For example, a compiler registry may store all the compiler implementations available and provide a default one. A registry allows users to subscribe and unsubscribe services into it. For example, when a notification has to be shown to the end user, the UIManager registry decides how to show it according to its registered providers: either by using the standard output or the graphical user interface. Pharo uses registries to manage different kinds of concerns such as the compiler suite, the fonts or the UI interactions.
Session Specific State. Session specific state is the global state that is tied to a particular session e.g., information gathered from the current platform, file handles and library handles. This state should be reinitialized or reset when a new session is started either in a new machine or a different one, to avoid misbehaviors and unexpected errors.

Process Controllers. Process controllers manage the life cycle of well known processes such as the idle process, the user interface (UI) process or the low space watcher process. They control how and when these well known processes are started, terminated, suspended and resumed.

Finalizables. Resources external to the language, such as files, sockets, or handles to external libraries, must be finalized accordingly when they are garbage collected or new session are started. For such a task, the classes of those objects implement a finalization mechanism to be aware of garbage collections and handle such situations.

Graphical Resources. Graphical resources are objects such as images, fonts, icons or bitmaps. These resources are embedded in the system using the global state constructs. As such, there is no general solution to discard them or reload them.

4.3 Results and Discussion of Impact

Table 1 lists the results of applying each of our categories to our set of individuals under study: how many of them apply to each category. The details of such a classification can be found in the Appendix A.

These results present some particularities we should take into account before doing a deep analysis. First, the number of detected graphical resources does not really represent the reality. A lot of graphical resources are represented as byte arrays in method literals (which we did not measure because of its complexity). With respect to the numbers in our results, we can argue that they give us an idea of the impact produced by each category i.e., code-migration libraries have to potentially handle each appearance of these patterns in an ad-hoc fashion, since they are not explicit in the source code. For example, if we would decide that on serialization all caches should be flushed, we must add custom code to handle each of the 43 caches.

<table>
<thead>
<tr>
<th>Category</th>
<th>Amount satisfying</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants</td>
<td>1722</td>
</tr>
<tr>
<td>Settings and Configurable Default Values</td>
<td>236</td>
</tr>
<tr>
<td>Singletons</td>
<td>65</td>
</tr>
<tr>
<td>Graphical Resources*</td>
<td>47</td>
</tr>
<tr>
<td>Caches</td>
<td>43</td>
</tr>
<tr>
<td>Registries</td>
<td>31</td>
</tr>
<tr>
<td>Session Specific</td>
<td>27</td>
</tr>
<tr>
<td>Process Controllers</td>
<td>11</td>
</tr>
<tr>
<td>Finalizables</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 1. Amount of individuals classified under each of the identified categories.

Because of its complexity. With respect to the numbers in our results, we can argue that they give us an idea of the impact produced by each category i.e., code-migration libraries have to potentially handle each appearance of these patterns in an ad-hoc fashion, since they are not explicit in the source code. For example, if we would decide that on serialization all caches should be flushed, we must add custom code to handle each of the 43 caches.

5. Discussion: a need for Reification

5.1 Concepts to Reify in Pharo.

Bouraqadi et al. [? ] presented already the need for the reification of resources used in a mobile code. The reification of resources provide support for an open architecture and facilitates the task of object migration. They also make explicit the concepts that are part of the program, providing with information the system can benefit from. We identify in particular the need for reification of the following elements part of our categories:

Processes. Pharo processes, although they are already objects, are managed from other objects. Process specific state is controlled by objects other than the process itself, breaking encapsulation. As such, the life-cycle of processes are tied to those objects that create them or keep their state. A first class representation of processes, on the other hand, will encapsulate the process specific state, avoid conflicts on its access, and provide a common interface for their manipulation.

Finalizables. First class finalizable resources provide a framework supporting uniform finalization and resource deallocation.

Caches. First class caches provide a uniform and complete implementation of caches libraries can rely upon. Additionally they will enable the system with introspection and self-modification of such caches.

Variables. First class variables, namely slots, were sketched by Verwaest et al. [? ] and a first version introduced into Pharo 3.0. Slots introduce the ability to refine instance variables, give them specific behavior and annotate them with meta-information. Specialized slots can be used to implement e.g., session specific state, constant values or settings.

5.2 Using explicit metaclasses.

Finding all singletons installed in the system could be easily achieved through the usage of explicit metaclasses [? ] or traits [? ]. Explicit metaclasses and traits allow the sharing of behavior between classes, and thus, they eliminate the need for ad-hoc implementations of e.g., singletons. Additionally, reifying the singleton abstraction in the language, provides with the ability to query and act upon the installed singletons. Implementing them with traits, however, presents as main limitation that the current trait implementation in Pharo is stateless. Thus, it does not allow to express class variables to hold the singleton instance.
6. Discussion: Moving responsibilities to the language runtime

Within our classification, we understand there are some concerns that should be moved under the umbrella of the language or its runtime system. The language may provide its own abstractions for recurrent problems such as caching or registering services. This will provide with the proper and needed meta-information to handle services. Additionally, providing end users with correct and complete implementations will avoid the ad-hoc implementations with repeated logic.

6.1 Resource manager

As we noted in the results, graphical resources such as images, icons and fonts are present as globally accessible resources in Pharo. We can add also that Pharo’s memory is occupied in great percentage by instantiated bitmaps\(^1\). There is not, however, a possibility to inspect all available resources, understand their origin (the package, class and method that defines them), or recreate them from files. This poses the need for a resource manager.

A sketch implementation of such a resource manager was implemented as a in-memory file system. In such a prototype, each Pharo package contains an associated file system that stores resources of that package. Images, icons, configuration files, and other files are stored in this file system. Package resources can be accessed from within and outside the package in an structured way, and serialized along with its package.

6.2 Session manager

How session specific state is handled nowadays denotes the need for a session manager. Currently, in the presence of session specific state, the class that stores it has to be subscribed to the startup and shutdown events of the runtime system. These two events are used to reset and initialize the class state when a new session is started.

We sketched a session manager to ease the management of session specific state. First class instance variables (Slots) describe declaratively their initialization when a new session is started. Then, during the startup of a new session, the session manager will reinitialize each of these slots using their description. This session manager encapsulates the need for the startup and shutdown lists, and removes such responsibility from the developer.

7. Related Work

Fuggetta et al.\(^2\) present also a classification of the state of mobile systems, but using as criteria the strategy used for migration. As such, their classification is orthogonal and complementary to ours. They present two properties to characterize the data to migrate

\(^{1}\) 24.50% according to our measures in latest Pharo version

\(^{2}\) Fuggetta et al. present also a classification of the state of mobile systems, but using as criteria the strategy used for migration. As such, their classification is orthogonal and complementary to ours. They present two properties to characterize the data to migrate.

Transferrable. A transferrable element is the one that can be physically migrated \(e.g.,\) a file. Oppositely, a non transferrable one is the one that cannot be migrated, \(e.g.,\) as a printer.

Desirability to transfer it. An application can mark some data as \textit{fixed} or \textit{free} according to its needs. Fixed data is associated permanently with its original environment, while free data migration is allowed.

and three ways to bind an application to a given resource

By identifier. Resources binded by identifier are tied with a particular instance of a resource \(e.g.,\) a socket. When a program is migrated, all its resources binded by identifier are kept in their original environment. A network communication is enforced between them.

By value. Resources binded by value are interested in the value of a resource and not in their identity, \(e.g.,\) the contents of a file. These kind of resources can be copied along with the program upon migration.

By type. Resources binded by type are intended to provide some kind of service despite their value or identity \(e.g.,\) a display. These kind of resources are rebinded to local resources of the same type after migration.

Unger et al. implemented a transporter for the Self programming language\[^ 2\]. This transporter had to deal with many of the difficulties we presented above, in particular the lack of explicit usage information. They provided a generic solution to the problem: let the developer annotate the objects’ slots to guarantee the desired state of the program upon a migration. However, a question remained: How should developers annotate the slots? To answer this question, they provided with a series of properties that must help in such analysis.

Does identity matter? The developer has to identify those objects whose identity matters, and those whose it doesn’t. When identity matters, the transporter must ensure that references to the same object are kept the same after migration. When it does not, the transporter can simply duplicate the object.

An initial value must always be enforced? Some objects must be reinitialized every time they are migrated. This is for example the case of caches.

An object must be written in an abstract or concrete way? Some objects can be rebuilt as the result of an expression, while some others must be built by concretely enumerating its slots.

8. Conclusion and Future Work

In this paper we studied the usage of global state in Pharo. The study of global state is interesting since references kept from global state are persisted in image-based systems. Global state is also a concern when working in
code mobility because resources globally available must be reinitialized or rebinded when code is migrated.

We present a classification of Pharo’s global state based on its usage, and found many patterns that are recurrent in the kernel of the language, though not explicit in the code. We discuss how to make explicit these patterns so the language kernel can benefit from it, either by reifying them or moving some responsibilities to the language kernel.

This work is a first step to prepare Pharo to the mobile code world. To be able to transport Pharo programs either in time or space, the abstractions we found should be made explicit in the language, and so, libraries and frameworks can take advantage of them. As future work we also consider that the discussed sketches have to be iterated and developed further.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, FEDER through the ’Contrat de Projets Etat Region (CPER) 2007-2013’, the Cutter ANR project, ANR-10-BLAN-0219 and the MEALS Marie Curie Actions program FP7-PEOPLE-2011- IRSES MEALS.
A. Appendix: Classification

A.1 Finalizables
FileHandle -> #Registry
FT2Handle -> #Registry
Socket -> #Registry
StandardFileStream -> #Registry
WeakRegistry -> #Default

A.2 Process Controllers
CPUWatcher -> #CurrentCPUWatcher
Delay -> #TimerEventLoop
MessageTally -> #Timer
MorphicUIManager -> #UIProcess
ProcessBrowser -> #WellKnownProcesses
ProcessorScheduler -> #BackgroundProcess
SmalltalkImage -> #LowSpaceProcess
UpdateStreamer -> #UpdateDownloader
WeakArray -> #FinalizationProcess

A.3 Registries
Beeper -> #default
ChangeSet -> #AllChangeSets
ChangeSet -> #current
EncodedCharSet -> #EncodedCharSets
ExternalDropHandler -> #DefaultHAndler
ExternalDropHandler -> #RegisteredHandlers
FileServices -> #FileReaderRegistry
FreeTypeFontProvider -> #current
FreeTypeGlyphRenderer -> #current
HelpBrowser -> #DefaultHelpBrowser
LanguageEnvironment -> #ClipboardInterpreterClass
LanguageEnvironment -> #Current
LanguageEnvironment -> #FileNameConverter
LanguageEnvironment -> #InputInterpreterClass
LanguageEnvironment -> #KnownEnvironments
LanguageEnvironment -> #SystemConverter
Locale -> #KnownLocales
MCPackageManager -> #registry
MCServerRegistry -> #registry
MetacelloProjectRegistration -> #registry
Nautilus -> #PluginClasses
PluggableTextMorph -> #StylingClass
RBProgramNode -> #FormatterClass
RGFactory -> #CurrentFactories
SmalltalkImage -> #CompilerClass
SmalltalkImage -> #Tools
SoundSystem -> #Current
TestResource -> #current
UIManager -> #Default
UITheme -> #Current
ZnServer -> #ManagedServers
ZnSingleThreadedServer -> #Default

A.4 Caches
ASTCache -> #default
AbstractMethodWidget -> #MethodsIconsCache
AbstractNautilusUI -> #ClassesIconsCache
AbstractNautilusUI -> #GroupsIconsCache
AbstractNautilusUI -> #PackagesIconsCache
BitBlt -> #CachedFontColorMaps
BitBlt -> #ColorConvertingMaps
CairoBackendCache -> #soleInstance
Color -> #CachedColormaps
Color -> #MaskingMap
FreeTypeCache -> #current
GLMUThemeExtraIcons -> #icons
GradientFillStyle -> #PixelRampCache
HelpIcons -> #Icons
KomitClass -> #classes
KomitMethod -> #methods
KomitPackage -> #packages
KomitRemote -> #icon
Komitter -> #lastMessage
LoadingMorphState -> #image
LogicalFont -> #all
MCDefinition -> #Instances
MCGitHubRepository -> #DownloadCache
MCMethodDefinition -> #Definitions
MCSaveVersionDialog -> #PreviousMessages
NECSymbols -> #cachedSymbols
RPackageSet -> #cachePackages
ScrollBar -> #ArrowImagesCache
ScrollBar -> #BoxesImagesCache
SettingDeclaration -> #ValueListCache
SingleCodeCriticResultList -> #icons
SugsSuggestionFactory -> #collectorForAll
SugsSuggestionFactory -> #collectorForAssignment
SugsSuggestionFactory -> #collectorForClassVariable
SugsSuggestionFactory -> #collectorForClass
SugsSuggestionFactory -> #collectorForInstancesVariable
SugsSuggestionFactory -> #collectorForLiteral
SugsSuggestionFactory -> #collectorForMessage
SugsSuggestionFactory -> #collectorForMethod
SugsSuggestionFactory -> #collectorForSourceCode
SugsSuggestionFactory -> #collectorForTemporaryVariable
SugsSuggestionFactory -> #collectorForUndeclaredVariable

A.5 Graphical Resources
AbstractMethodWidget -> #MethodsIconsCache
AbstractNautilusUI -> #ClassesIconsCache
AbstractNautilusUI -> #GroupsIconsCache
AbstractNautilusUI -> #PackagesIconsCache
Cursor -> #BlankCursor
Cursor -> #BottomLeftCursor
Cursor -> #BottomRightCursor
Cursor -> #CornerCursor
Cursor -> #CrossHairCursor
Cursor -> #CurrentCursor
Cursor -> #DownCursor
Cursor -> #MarkerCursor
Cursor -> #MenuCursor
Cursor -> #MoveCursor
Cursor -> #NormalCursor

2014/8/19
A.6 Session Specific State

MCGitHubRepository -> #DownloadCache
MCCacheRepository -> #default
DiskStore -> #CurrentFS
NOCCompletionTable -> #table
NOCCompletionTable -> #classTable
Locale -> #Current
Locale -> #CurrentPlatform
DateTime -> #LocalTimeZone
FTHandle -> #Session
FileLocator -> #Resolver
FileStream -> #Stdin
FileStream -> #Stdout
FileStream -> #TheStdioHandles
FileStream -> #StdioFiles
FileStream -> #Stderr
LanguageEnvironment -> #SystemConverter
LanguageEnvironment -> #FileNameConverter
UUIDGenerator -> #Default
VirtualMachine -> #WordSize
WeakFinalizationList -> #HasNewFinalization
AthensCairoSurface -> #uniqueSession
AthensCairoSurface -> #dispatcher
AthensCairoSurface -> #dispatcherStruct
CairoLibraryLoader -> #session
CairoLibraryLoader -> #libHandle
Session -> #current
MultiByteFileStream -> #LineEndDefault

A.7 Singletons

ASTCache -> #default
ActiveEvent -> #ActiveEvent
ActiveHand -> #ActiveHand
ActiveWorld -> #ActiveWorld
Author -> #uniqueInstance
BorderStyle -> #Default
CPUWatcher -> #CurrentCPUWatcher
CairoBackendCache -> #soleInstance
ChangesLog -> #DefaultInstance
Clipboard -> #Default
CommandLineArguments -> #singleton
CriticWorkingConfiguration -> #Current
Display -> #Display
EditorFindReplaceDialogWindow -> #Singleton
EmptyLayout -> #instance
FreeTypeCache -> #current
FreeTypeSettings -> #current
IdentityTransform -> #Default
InputEventFetcher -> #Default
KMBuffer -> #uniqueInstance
KMPpragmaKeymapBuilder -> #UniqueInstance
KMRpository -> #Singleton
KomitterManager -> #instance
LayoutEmptyScope -> #instance
LogicalFontManager -> #current
MBConfigurationRoot -> #Current
MCFileeTreeFileUtils -> #Current
MCRepositoryGroup -> #default
MCServerRegistry -> #uniqueInstance
MetacelloPlatform -> #Current
NBExternalResourceManager -> #soleInstance
NEController -> #uniqueInstance
NNAviation -> #Instance
NNAviation -> #Instance
NativeBoost -> #Current
NautilusMonticello -> #Default
OSPlatform -> #Current
PackageOrganizer -> #default
PharoFilesOpenener -> #Default
PharoTutorial -> #Instance
ProcessSpecificVariable -> #soleInstance
Processor -> #Processor
RBRefactoringManager -> #Instance
RBRefactoringChangeManager -> #Instance
RPackageOrganizer -> #default
RecentMessageList -> #UniqueInstance
Sensor -> #Sensor
SharedValueHolder -> #instance
Smalltalk -> #Smalltalk
SoundTheme -> #Current
SourceFiles -> #SourceFiles
Spotlight -> #Current
StartupPreferencesLoader -> #UniqueInstance
SystemAnnouncer -> #announcer
A.8 Settings and Configurable Default Values

AbstractNautilusUI -> #NextFocusKey
AbstractNautilusUI -> #PreviousFocusKey
AlphaImageMorph -> #DefaultImage
BalloonMorph -> #BalloonFont
CCompilationContext -> #WarningAllowed
CPUWatcher -> #CpuWatcherEnabled
ChangeSet -> #DefaultChangeSetDirectoryName
ChangeSet -> #MustCheckForSlips
CodeHolder -> #AnnotationRequests
CodeHolder -> #BrowseWithPrettyPrint
CodeHolder -> #DecorateBrowserButtons
CodeHolder -> #DiffsWithInChangeList
CodeHolder -> #DiffsWithWithPrettyPrint
CodeHolder -> #OptionalButtons
CodeHolder -> #ShowAnnotationPane
CodeHolder -> #SmartUpdating
CommandLineUIManager -> #SnapshotErrorImage
DangerousClassNotifier -> #enabled
Deprecation -> #RaiseWarning
Deprecation -> #ShowWarning
DialogItemsChooserUI -> #alreadySearchedSelectedItemsListMaxSize
DialogItemsChooserUI -> #alreadySearchedUnselectedItemsListMaxSize
DisplayScreen -> #DeferringUpdates
DisplayScreen -> #DisplayChangeSignature
DisplayScreen -> #LastScreenModeSelected
DisplayScreen -> #ScreenSave
Editor -> #BlinkingCursor
Editor -> #CmdKeysInText
Editor -> #DumbbellCursor
Editor -> #SkipOverMultipleSpaces
EyeInspector -> #useAutoRefresh
FLCompiledMethodCluster -> #transformationForSerializing
FinderUI -> #Icon
FinderUI -> #searchedTextListMaxSize
Form -> #FloodFillTolerance
FreeTypeSettings -> #UpdateFontsAtImageStartup
FreeTypeSystemSettings -> #LoadFT2Library
GrowlMorph -> #DefaultBackgroundColor
GrowlMorph -> #Position
HaloMorph -> #CurrentHaloSpecifications
HaloMorph -> #HaloEnclosesFullBounds
HaloMorph -> #HaloWithDebugHandle
HaloMorph -> #ShowBoundsInHalo
HandMorph -> #DoubleClickTime
TextEditor -> #UseFindReplaceSelection
TextEditor -> #UseSecondarySelection
TextEditor -> #UseSelectionBar
TextEditor -> #cmdActions
TextEditor -> #shiftCmdActions
TextEntryDialogWindow -> #MinimumWidth
UITheme -> #defaultSettings
UserInterruptHandler -> #CmdDotEnabled
Week -> #StartDay
WorldState -> #CanSurrenderToOS
WorldState -> #DebugShowDamage
WorldState -> #DesktopMenuPragmaKeyword
WorldState -> #DesktopMenuTitle
WorldState -> #EasySelectingWorld
WorldState -> #MinCycleLapse
WorldState -> #ServerMode
WorldState -> #ShowUpdateOptionInWorldMenu
ZnConstants -> #DefaultMaximumEntitySize
ZnServer -> #AlwaysRestart

A.9 Constants

AJConstants -> #CcA
AJConstants -> #CcABOVE
AJConstants -> #CcABOVEEQUAL
AJConstants -> #CcAE
AJConstants -> #CcB
AJConstants -> #CcBE
AJConstants -> #CcBELOW
AJConstants -> #CcBELOWEQUAL
AJConstants -> #CcC
AJConstants -> #CcE
AJConstants -> #CcEQUAL
AJConstants -> #CcFNPNOTUNORDERED
AJConstants -> #CcFPUNORDERED
AJConstants -> #CcG
AJConstants -> #CcGE
AJConstants -> #CcGREATER
AJConstants -> #CcGREATEREQUAL
AJConstants -> #CcL
AJConstants -> #CcLE
AJConstants -> #CcLESS
AJConstants -> #CcLESSEQUAL
AJConstants -> #CcNA
AJConstants -> #CcNAE
AJConstants -> #CcNB
AJConstants -> #CcNBE
AJConstants -> #CcNC
AJConstants -> #CcNE
AJConstants -> #CcNEGATIVE
AJConstants -> #CcNG
AJConstants -> #CcNGE
AJConstants -> #CcNL
AJConstants -> #CcNLE
AJConstants -> #CcNO
AJConstants -> #CcNOCONDITION
AJConstants -> #CcNOOVERFLOW
AJConstants -> #CcNOTEQUAL
AJConstants -> #CcNOTSIGN
AJConstants -> #CcNOTZERO
AJConstants -> #RIDESP
AJConstants -> #RegCodeMask
AJConstants -> #RegGPB
AJConstants -> #RegGPD
AJConstants -> #RegGPQ
AJConstants -> #RegGPW
AJConstants -> #RegHighByteMask
AJConstants -> #RegMM
AJConstants -> #RegProhibitsRexMask
AJConstants -> #RegRequiresRexMask
AJConstants -> #RegTypeMask
AJConstants -> #RegX87
AJConstants -> #RegXMM
AJConstants -> #SegmentCS
AJConstants -> #SegmentDS
AJConstants -> #SegmentES
AJConstants -> #SegmentFS
AJConstants -> #SegmentGS
AJConstants -> #SegmentNONE
AJConstants -> #SegmentSS
AJConstants -> #SizeByte
AJConstants -> #SizeDQWord
AJConstants -> #SizeDWord
AJConstants -> #SizeQWord
AJConstants -> #SizeTWord
AJConstants -> #SizeWord
AJx86InstructionDescription -> #instructions
AJx86Registers -> #AH
AJx86Registers -> #AL
AJx86Registers -> #AX
AJx86Registers -> #BH
AJx86Registers -> #BL
AJx86Registers -> #BP
AJx86Registers -> #BPL
AJx86Registers -> #BX
AJx86Registers -> #CH
AJx86Registers -> #CL
AJx86Registers -> #CX
AJx86Registers -> #Codes
AJx86Registers -> #DH
AJx86Registers -> #DI
AJx86Registers -> #DL
AJx86Registers -> #DX
AJx86Registers -> #EAX
AJx86Registers -> #EBP
AJx86Registers -> #EBX
AJx86Registers -> #ECX
AJx86Registers -> #EDI
AJx86Registers -> #EDX
AJx86Registers -> #EIP
AJx86Registers -> #ESI
AJx86Registers -> #ESP
AJx86Registers -> #IP
AJx86Registers -> #MM0
AJx86Registers -> #MM1
AJx86Registers -> #MM2
AJx86Registers -> #MM3
AJx86Registers -> #MM4
AJx86Registers -> #MM5
AJx86Registers -> #MM6
AJx86Registers -> #MM7
AJx86Registers -> #R10
AJx86Registers -> #R10B
AJx86Registers -> #R10D
AJx86Registers -> #R11
AJx86Registers -> #R11B
AJx86Registers -> #R11D
AJx86Registers -> #R11W
AJx86Registers -> #R12
AJx86Registers -> #R12B
AJx86Registers -> #R12D
AJx86Registers -> #R12W
AJx86Registers -> #R13
AJx86Registers -> #R13B
AJx86Registers -> #R13D
AJx86Registers -> #R13W
AJx86Registers -> #R14
AJx86Registers -> #R14B
AJx86Registers -> #R14D
AJx86Registers -> #R14W
AJx86Registers -> #R15
AJx86Registers -> #R15B
AJx86Registers -> #R15D
AJx86Registers -> #R15W
AJx86Registers -> #R8
AJx86Registers -> #R8B
AJx86Registers -> #R8D
AJx86Registers -> #R8W
AJx86Registers -> #R9
AJx86Registers -> #R9B
AJx86Registers -> #R9D
AJx86Registers -> #R9W
AJx86Registers -> #RAX
AJx86Registers -> #RBP
AJx86Registers -> #RBX
AJx86Registers -> #RCX
AJx86Registers -> #RDI
AJx86Registers -> #RDX
AJx86Registers -> #RIP
AJx86Registers -> #RSI
AJx86Registers -> #RSP
AJx86Registers -> #SI
AJx86Registers -> #SIL
AJx86Registers -> #SP
AJx86Registers -> #SPL
AJx86Registers -> #ST0
AJx86Registers -> #ST1
AJx86Registers -> #ST2
AJx86Registers -> #ST3
AJx86Registers -> #ST4
AJx86Registers -> #ST5
AJx86Registers -> #ST6
AJx86Registers -> #ST7
AJx86Registers -> #XMM0
AJx86Registers -> #XMM1
AJx86Registers -> #XMM2
AJx86Registers -> #XMM3
AJx86Registers -> #XMM10
AJx86Registers -> #XMM11