Pharo's Vision: Goals, Processes, and Development Effort

Stéphane Ducasse, Marcus Denker, Damien Pollet

Pharo's Vision

Goals, Processes, and Development Effort

http://www.pharo-project.org

Stéphane Ducasse
stephane.ducasse@inria.fr

Marcus Denker
marcus.denker@inria.fr

Damien Pollet
damien.pollet@inria.fr

Version 1.0 — March 1st, 2012
Public

Abstract

This document presents the goals, processes, architectural vision and current and future development efforts in Pharo core. It will serve as a working document to structure the effort for the next versions of Pharo core. By Pharo core, we mean the essential parts of the system, such as the compiler, basic libraries, and key infrastructure such as canvas, events, etc. We hope that this document will also bring good energy to achieve some of the goals described in the roadmap. It document should be read as a proposal and explanation of some of our effort. Now by no means what we describe here is carved into stone. We are really interested in constructive criticism, comments and suggestions. In addition, people who want to participate to the definition of this document are welcome. Finally, we are convinced that the best core is nothing if it has no cool libraries; so we encourage people to build libraries and we will support them.

We hope that this document will put a good light on what we want to achieve and share with the community. We also hope that it will make clear what the system we want and that we can build it together.

Pharo Zen

Our values and convictions are condensed in this simple list.

Easy to understand, easy to learn from, easy to change.
Objects all the way down.
Examples to learn from.
Fully dynamic and malleable.
Beauty in the code, beauty in the comments.
Simplicity is the ultimate elegance.
Better a set of small polymorphic classes than a large ugly one.
Classes structure our vocabulary.
Messages are our vocabulary.
Polymorphism is our esperanto.
Abstraction and composition are our friends.
Tests are important but can be changed.
Explicit is better than implicit.
Magic only at the right place.
One step at a time.
There is no unimportant fix.
Learning from mistakes.
Perfection can kill movement.
Quality is a emerging property.
Simple processes to support progress.
Communication is key.
A system with robust abstractions that a single person can understand.

Contents

CHAPTER 1

Goals

Pharo is an open-source Smalltalk inspired system started in 2008. The official web site is http://www.pharo-project.org.

It may happen that people misunderstand the goals that drive Pharo and our effort on the core of Pharo. This section will make some of these goals more evident. We will work to make them real during the next five years (may be less if more people help). This is why we believe that this is important for people interested in Pharo to read at least this chapter. We sketch some possible scenarios of use cases for Pharo. Such scenarios are important since they shed light on the reasons behind some of our actions.

Now it is important to make clear that we are talking about the core of Pharo. We are convinced that the best core is nothing if it has not cool libraries. Seaside is a typical and successful example. Therefore we strongly encourage people to build libraries and we will support them. The distribution infrastructure we want to put in place is clearly an important point to help library developers.

1.1 Goals of this document

The goals of this document are:

- Explain our goals and course of actions.
- Describe the processes we want to put in place to achieve our goals.
- Expose our current plans and action list: some people can be too busy to read the mailing-list discussions. Sometimes people think that because Pharo does

not do or have a given feature, it is because we are not interested in it. Usually the reason is the lack of resources, not of interest. This document is the map for our goals. In addition, this document puts in perspective our actions. They are concerted and centered on our vision of what an excellent system should be.

- Gather feedback and share a common vision. If you want to help improving this document or give feedback do it in the pharo mailing-list. Tag the mail with [Vision]
- Build a momentum. We were not really good at describing and explaining the exact action map we have in mind. We hope that this document will put a good light on what we want to achieve and share with the community. We also hope that it will make clear what the system we want and that we can build it together.

1.2 General goals

Pharo's goal is to deliver a clean, *robust*, *innovative*, free open-source Smalltalk *inspired* environment. We want to provide a stable and small core system, excellent development tools, maintained libraries and releases, and to continue making Pharo an attractive platform to build and deploy mission critical Smalltalk applications.

Robust. We want Pharo to be a really robust and tested system. We have already developed an infrastructure based on an integration server that automatically generates new images, tests them and notifies its results. We will also put in place automatic rule-checking based on integration servers.

Innovative and Smalltalk inspired. We want to stress the fact that Pharo will certainly derive from ANSI and other Smalltalks. We will not change for the sake of change. What we want to say is that if there is something that can be improved but does not conform the ANSI Smalltalk, we will do it anyway. We favor innovation and better systems. We want, for example, first class instance variables because they will open a wide range of possibilities for metamodeling, optimization at the compiler level, expressing all kinds of information in a more compact form ...

Strong and Powerful Infrastructure as a Motto

We do not believe that we can build a nice system on top of a weak infrastructure. What we do believe, is that the infrastructure should be improved so that the complete ecosystem improves. Not only do we want that people build their own libraries on top of Pharo, but also we want the core of Pharo to be built on top of an improved

infrastructure. This is why we want both users and the general quality of the system to benefit from the best libraries.

We believe in software engineering tools:

- We want to have validated packages based on automatic tests and rule checking.
- We want Virtual Machines to run regression tests to identify possible regressions. In addition, we would like to run benchmarks and see how each commit affects performance.
- We want to have good interaction with the rest of the world.

What do we mean by a small core?

We want Pharo to be modular. In addition we want it to be designed around a small core. Given a such small core and specifications, we want to be able to load a set of packages to produce applications or development environments (see Figure 1.1).

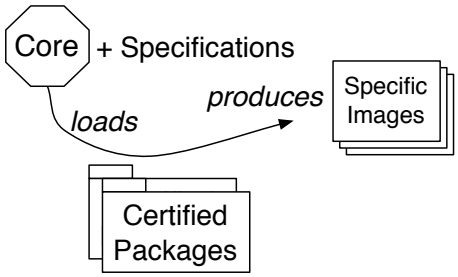

Figure 1.1: Given a small kernel and specifications we should load a set of certified packages to produce intermediate (or not) systems expressed as full images.

1.3 Some possible future applications for Pharo

The success of Seaside is an important factor in the success of Pharo and in general a large contributor to the renewal of Smalltalk. However, it should not make us forget about the future of Pharo as an innovation and business platform. Here we would like to sketch some possible future use cases for Pharo. Doing so we want to share with you some of the requirements that are needed to achieve such case and that are driving some of our current efforts in the core of the system.

Some of the points raised below affect the core of Pharo, its Virtual Machines while others the libraries built on top. We will mainly focus on the requirements that affects the core and that constitute our roadmap for the next two years. To make the scenario more precise we list some libraries that are clearly not in the core (and not in

our focus). The list is not exhaustive but should allow us to structure the development effort.

Again read carefully this sentence: the following scenarios are not what we want to build but what we want you to possibly build with Pharo. We are focusing on making sure that such scenario can happen.

New IDEs, new interaction frameworks

We believe that Smalltalk is a good language and environment for building new advanced user interfaces. By user interfaces we mean: new widgets and UI frameworks but also new ways of handling events and notifications, new devices and their specific interactions (multitouch, gestures), etc.

In addition, this is important not only to be able to develop applications using such new devices but also to invent new abstractions to handle them. Therefore the possibility to explore the design space is crucial. DrGeoII running on iPad or android is a typical example of such a need.

Requirements

- Zoomable interface. We should be able to adapt to size and configuration of new devices, as well as support explorating new kind of IDEs, of desktop metaphors, etc.
- Extensible Event Infrastructure. We should be able to add new events or totally replace them. In addition, the event loop handling and interpretation (semantics of mouse event for example) should be either replaced or simply extended.
- Pluggable UI frameworks and interaction. The infrastructure should make sure that two UI frameworks should be able to coexist (probably in separate worlds) so that we can use Morphic while designing new framework. Early refactorings made in Pharo were driven by this vision and Alain Plantec's prototype of a Cairo based multi-windowing with its own event loop proved that the refactorings were worth doing.

The infrastructure should enable new approaches to emerge, such as Gaucho, Mars or others.

Large applications

We believe that Pharo is a good infrastructure to support the development of large applications. Moose is a typical example of a large and complex application built out of several large frameworks and manipulating hundreds of thousand of objects.

Requirements

- Fast data structures and rich libraries. Obviously everybody prefers using a fast library over a slow one. We should develop rule checking that includes speed regression testing. Now as the core development team we will focus on a clear set of packages, but we will offer a testing and deployment infrastructure for external tool and library builders (automatic builds, better tools to check quality). The libraries we will maintain and focus on are mainly the ones needed for Pharo Core to work.
- Fast object loader. Manipulating a lot of data often requires to save and load large amount of objects. Having an efficient and fast object serializer is a plus.
- Solid and multithreaded FFI connection. Connecting to external libraries such as databases is a key aspect. Therefore we need a robust and multithreaded FFI library.
- 64 bits. To manage a large amount of data, the 64 bit architecture is also important.
- A VM capable of running large images. Although a 32 bit VM can theoretically run images of up to 2 GB, the currently available VMs are more limited. For example, the Windows VM does not support images bigger than 512 MB.

Advanced Multimedia Applications

OpenSophie proved that Smalltalk, and Squeak (Pharo's ancestor) in particular, are good platforms for building advanced multimedia applications. We believe that Pharo is well-suited for this.

Requirements

- Fast data structures and rich libraries. As above.
- Solid FFI connection. Communicating with multimedia libraries such as GStreamer.
- Solid I/O and network libraries are needed.
- Strong and flexible widget library.
- Fast object loader. Multimedia applications manipulates large objects. Having an efficient fast object serializer is a plus.
- Supporting large amount of memory.
- Fast and feature wide stream implementation. Streams should be fast but also support special features such as compression, encryption, etc.
- Ways of communicating with different clients (in case they are not built in Pharo): SOAP, web services, object serialization to other languages using JSON or similar.

- Excellent error handler. If there is an error in a server, there should be always a log, a trace, a dump, etc. As much information as possible. It cannot happen that the server just goes down and there is "no log". This impacts not only in the language side but also in the VM.
- Load balancer. It would be nice a tool to let us run different images in different VMs running in different CPUs or cores.

Language and Remote IDEs for Small Devices

There is new range of devices named Plug Computers http://plugcomputer.org/, with a need for IDEs to develop, deploy and debug on such computers. Pharo can really position itself as a strong IDE for such computers. We can imagine to deploy and remotely manage applications on such devices.

Requirements

- Small core but not that small. Indeed plug computer are small computers but they do not have real memory constraints.
- Customizable compiler. We should be able to support remote working and compilation.
- Remote and network tools.
- Reliable serializer to move objects between images.

Scripting Systems and Glue

Pharo has the potential to get the best of two worlds: script/shell and interactive environment. We imagine that we can write scripts and get a debugger on demand, edit the script in an interactive mode, save it and continue. Similarly we could imagine that an image can act as a smart cache that can get recreated on demand based on a list of packages loaded into a core.

Requirements

- Small core. Having a small core is important to be able to startup fast Pharo and to have multiple executions.
- Scripting syntax. Coral proposes a scripting syntax for Pharo.
- Good C interaction. Being able to invoke C libraries is required.
- Good Shell interaction. Having a good integration with the host shell and system is also important.
- Libraries with scriptable API. We need to have a rich set of libraries to connect to a large numbers of protocols and others (XML, JSON, REST...).

- A VM and images that work properly in headless mode. Ideally, the VM should be just a C application that can be embedded in any C application.

Robotics

There is a demand for good IDEs for robot application development. Being able to debug robot while the robots are running and on the fly fixing the code is needed. Pharo can be the basis for building IDEs and runtime engine for robots.

Requirements

- Network. We need a good network library.
- Small core.
- C integration. Interacting with external devices is important.
- Support for different platforms. We should pay attention that the architecture of robots is not as homogeneous as for computer, therefore being able to run on different processor is important.

Web/Deployment Infrastructure

Nowadays some systems use the web as their user interface, which means they have to replicate, load balance, as well as store a lot of data and manage a lot of load requests.

Requirements

- Network infrastructure. Solid network infrastructure is a must. The work around Zinc (HTTP), Ocean (Socket) and Zodiac (HTTPS) is definitively important.
- Dynamic web servers. Seaside is clearly one of the reason of the success of Pharo. Now it is important that new approaches are not killed by the success of Seaside. The biotope should let multiple approaches to flourish. The support for REST in Seaside is typically one big plus.
- Object serialization.
- VM should offer object immutability to help identifying changed objects and their storage.
- Good proxy. There is a need to have one powerful robust and engineered proxy library to support external object management.

More Secure Languages

We believe that Pharo should provide a good infrastructure for inventing the next generation of languages, and in particular in the context of secure dynamic languages.

Requirements

- Modifiable VM.
- Customizable Compiler.
- Flexible IDE.
- Small core.
- Cleaned and revisited MetaObject Protocol.

Support for Language Innovation

Helvetia [**?**], language boxes [**?**] and PetitParser are typical examples showing that language innovation can be done in Smalltalk. Pharo should be the environment to support such type of activities. In addition, Pharo should benefit from research on type inferencers, traits...

Requirements

- Smalltalk is a simple language: therefore its compiler should be simple and allow us to make crazy extensions.
- The tool chain should be parametrizable.

CHAPTER 2

The Pharo Consortium

Since more than a year we are working with INRIA and the Pharo's community to build a consortium to support Pharo and its ecosystem. Our goal is to build a legal infrastructure that will be able to sustain the development of Pharo and improve its future.

2.1 Goals

There are several goals behind the creation of this consortium.

Promote. The consortium will promote Pharo by supporting conference, books, videos, conference participation. It will help Pharo and its community can expand and structure itself.

Participate to the vision. The role of the consortium is also to help deciding the future development of Pharo and gathering inputs from a business perspective.

Support our common future. There is a need to have an organization that can collect money. The consortium is a chance for the community to structure itself and support the future of Pharo and its virtual machine. The consortium should be able to collect funds (ways as to be determined - we foresee a membership model or moral license) to pay engineering tasks to be performed such as improving the virtual machine, network libraries, better JIT support.

Provide a solid and trustable visibility. The consortium will show that our technology is mature and relevant from a business perspective.

Provide credibility. The consortium will support a business eco-system around Pharo. The consortium may offer some support and certification again to enhance credibility of our technology.

First Phase. At the beginning, the consortium will be hosted by INRIA but the goal is to create a separate and autonomous entity if we succeed to make the consortium viable.

2.2 What does it change for you?

You only have to gain.

- Pharo's license will stay the same: It will continue to be MIT. You will be able to make the same business with it.
- Pharo will continue to be free. Nothing change.
- The consortium will make Pharo better.

Advantages for you. You will

- Have preferential conditions for conferences, seminars...organized by the consortium.
- Influence/have a role in defining development priorities for PHARO. Participate to the governance of the project.
- Have preferential access to development team.
- Have access to preferential support.
- Be able to pass a PHARO certification program for free and participate to the PHARO certified companies.
- Freely publish job offers on the PHARO Consortium web site.
- Publish a CV for individual consultants on the consortium web site.
- Follow lectures at half price and can become a Pharo trainer.
- Have free access to training materials.
- Market that you are part of a powerful consortium.
- Support the development of the next generation of the system.
- Enable the future of Pharo.
- Show that you are participating to a living, full of energy and powerful community.

Support Pharo. To make it short we would like to give a chance to our community to grow and structure itself so that Pharo can get stronger and that risk (truck factor)

gets minimized. INRIA in the context of the consortium building sponsored an expert engineer position in addition to the position of Igor Stasenko.

If Pharo has a value for you and you want to support it, sign the letter of support you can find at http://www.pharo-project.org/community/consortium and contact us.

CHAPTER 3

Processes

In this chapter we present the processes to support the generation of a small kernel and a set of validated packages. We sketch also the future release process we want to put in place.

3.1 Towards a small kernel

As mentioned in the goals chapter, we want and we are steadily working *since years* step by step on code cleaning to obtain a small core. We are getting there but it requires a lot of rewriting.

In the long-term, we expect to have a minimal core with a binary loader that only load packages on demand and where an image acts as a smart cache *e.g.,* a cache with loaded packages from a core that we can rebuild on demand. With such approach we get the best of both world: use the image to develop or to package applications but also be able to build applications from a small kernel and a set of packages.

Process. We want a process that will be centered around this small core. The path to arrive there is not trivial and not as fast as we would like. It requires a lot of cleaning because there are many dependencies and not modular concepts. We made already a lot of progress with for example the Settings framework usage. We expose some of the non technical problems we are facing in the following.

Figure 3.1 shows that we want to have a small core (See Section 4.5) and use some specification (probably in terms of Metacello configuration) to load a set of certified packages (The package certification is the responsibility of the catalog builder

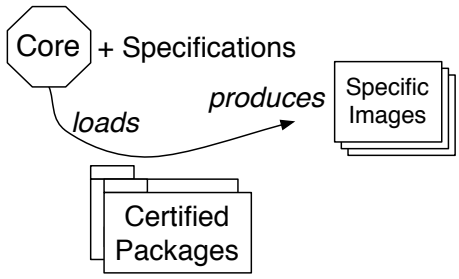

Figure 3.1: Given a core and specifications we load a set of certified packages to produce applications/images.

described in Section 3.2). Note that in presence of a fast binary loader (such a Fuel) we could avoid to create images and delay the creation of image.

Benefits. There are obvious benefits to have a small core and this is why also we are working on bootstrapping the system (See Section 4.5). We discovered and fixed already a lot of problems.

Dealing with changes. One key question is how do we deal with changes. This question is important because it has some requirements and challenges. Figure 3.2 shows that given a core, specification, a set of packages and a change we should get potentially a new core, a new set of specifications and new set of packages. For us this is the ideal situation we would like to reach.

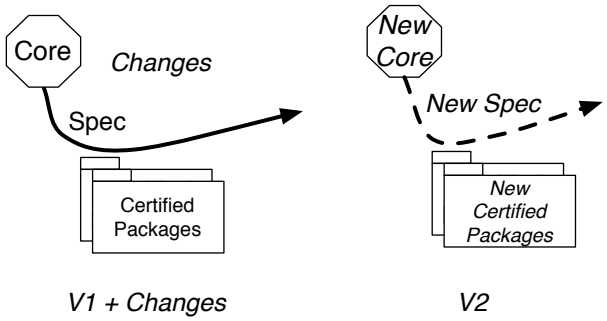

Figure 3.2: An ideal scenario: Given a core, specifications, a set of packages and a change we should get a new core, new specifications and new set of packages.

Problems. We currently face some problems to apply the ideal scenario we depicted above.

- Loading all the packages should take a reasonable amount of time because it is not reasonable to wait couple of 10 of minutes that an integration engine build the system. Already publishing changes takes too much time. So recreating in addition a working system would slow us down too much.
- Maintaining a core image and an image with some non-core but important packages that are shipped is *really* difficult for the following reasons:
  - In general, we do not have the knowledge or time to maintain externally developed packages.
  - When we do a refactoring in the core it may touch parts used by the tools and since we are working on alpha branches it means that we should modify the non-core packages to reflect our changes.
- Not loading certain packages such as Morphic Example, Sounds can lead to a difficult situation since such packages can get less care and desynchronized from the rest of the system slowly making them less adequate. Now loading permanently such packages goes against our goal to get a small core. This is why being able to load fast all the packages that we maintain when doing a change would be really a good feature to have.
- Since not everybody is using Metacello we have to maintain Configuration for packages we do not know, we cannot simply maintain stable and development branches.
- Having good tools to work cleaning the system is important. Right now we cannot load fast enough the refactoring engine and tools, change the system without impacting the refactoring tools and save the modified package.
- All the packages that are required in the core or to be loaded by the process (for example the AST package) should be owned by Pharo to be able to evolve when needed. For example to bootstrap the compiler in itself (and avoid to have to maintain two compilers) we need to have the compiler tool chained parametrized (there are other really good reasons - Read Section 4.6).

A midterm solution to get the process working is:

- To load some predefined tools or packages into the "Core" and we take the responsibility to maintain these even at the cost to make mistakes;
- Modify these elements as part of the packages required by the system;
- Make sure that we do not introduce new dependencies.

A better solution would be that:

- All the important tools are managed by their developers which works with us and test the changes with us and maintained with stable and development Metacello tags.

- Get a really fast loader.
- Do some steps in the direction of been able to reapply changes.

We could record some changes to be applied on the packages that were not managed, but so far we do not have the right infrastructure and experience to do that.

Business Value. Having such goal and process can help getting solid, scalable systems that can also be deployed on constrained devices.

3.2 Towards a verified package catalog

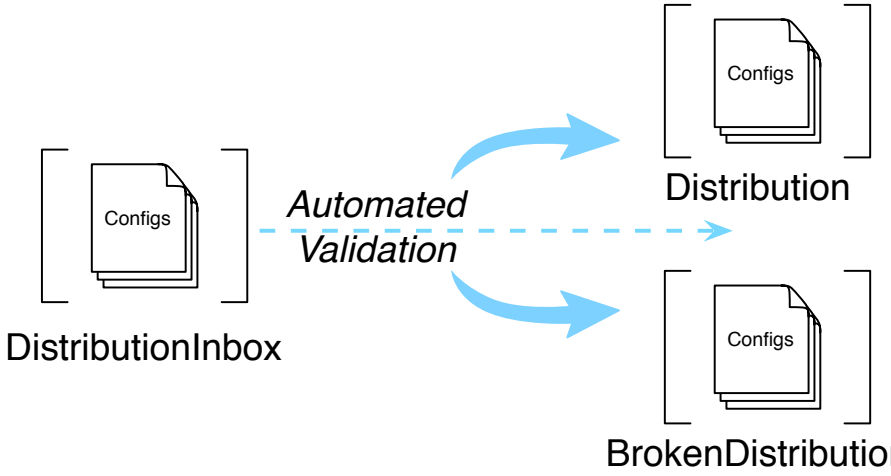

Figure 3.3: ConfigurationOfProject published in the DistributionInbox, the Configurations will be loaded and their tests and checks will be executed.

We want to provide a more robust package distribution to the end-programmer. For this we plan to put in place the following process.

Process. Figure 3.3 illustrates the principle. For each Pharo distribution we want to have one repository containing the validated configuration of projects. We foresee a validation process that works as follows: (1) configurations are published in the distribution inbox. (2) an automated process: loads, run the tests and the rules of the projects. (3) depending on the success of the previous steps the configuration is included in the distribution or moved in a broken repository for study.

In a second step, the transitive closure of all the needed packages will be copied automatically in the distribution repository to create a self contained distribution. Publishing a configuration will then mean more than simply copying the distribution.

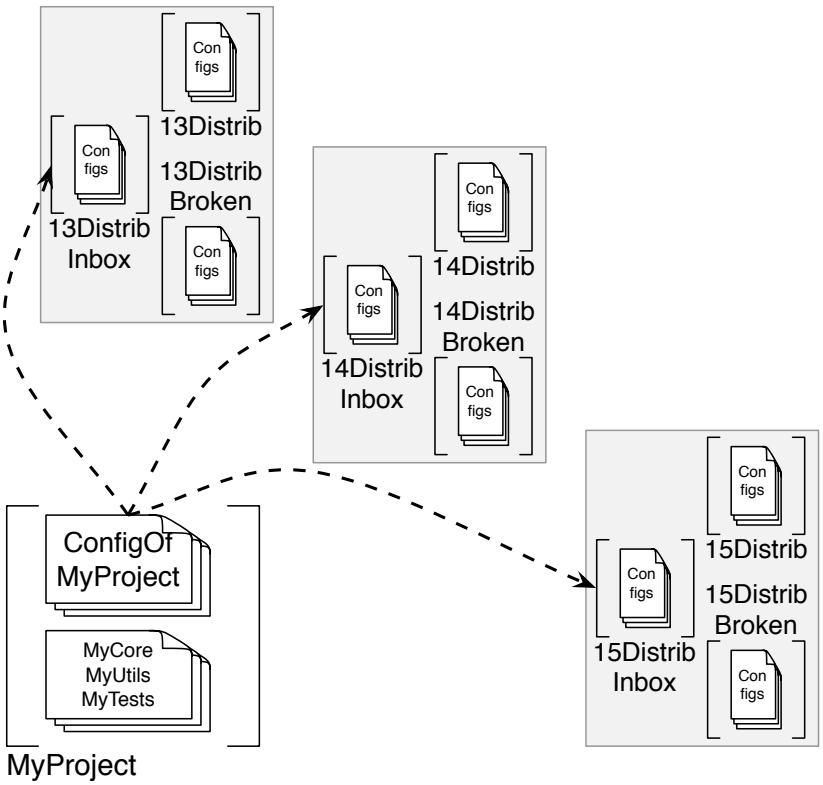

Figure 3.4: You should save your configuration in your repository. Then publish it when ready in the different repositories for validation.

About your project. Figure 3.4 shows the process we think that projects should follow.

- Always publish the configuration of your project in your project repository. Think about modularity.
- Push to the given distribution inbox when ready for it.

3.3 Pharo Releases

After a couple of year we are stabilizing our release process. We plan to follow the Eclipse process or a similar one. http://wiki.eclipse.org/SimRel/Overview http://wiki.eclipse.org/Simultaneous_Release. In essence we want to have:

- One main release every year.
- Two bugfix releases every year.

- After the two bugfix releases a new main release is done and the old main release becomes unsupported.

The releases are date driven, not feature driven. The dates for the release candidates and milestone releases as well as feature and API freezes are set years in advance. This has some downsides. For example if your feature misses a deadline, you have to wait for more than a year to see it in a release. But it makes everything clear.

CHAPTER 4

Current Development Effort

Now we explain the current development efforts and why they are done that way and their impact on the resulting system. The scenarios presented in the first section are also a reason why we are doing the developments we show now. The list is not exhaustive but we realize that is is important to finish a task before passing to the next one.

Note that some efforts are simpler to integrate or require less development effort. We take a pragmatic approach: if a change is ready to get integrate we do it. Now we are conscious that we should finish the work because else we risk to lose it.

4.1 A Robust and Extensible System Events

We need an infrastructure to easily be able to plug multitouch, Genie like behavior (Genie was a hand recognition system developed by Nathanael Schaerli in 2002) and experiment with new devices or UI metaphors.

Context. Before Pharo, the VM filled up a buffer for each new event and the system checked at regular interval if there was something present (See Figure 4.1).

Then different UI frameworks (Morphic and MVC) were calling the InputSensor which had a polling logic to see if a new events was present in the buffer. In the case of Morphic, HandMorph interprets itself this buffer and creates and interprets events. Notice that there was confusion and code in Morphic was directly calling InputSensor logic and not the HandMorph one, breaking layers. In addition HandMorph handles events using an explicit case statement logic which hampers extensibility.

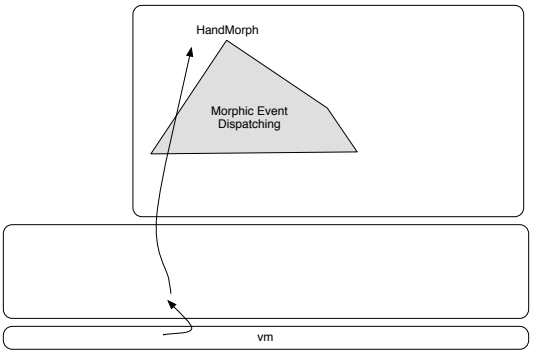

Figure 4.1: Old days: VM filled up a buffer and the system checked at regular interval if there was something present. HandMorph interprets this buffer and creates and interprets events.

Current ongoing effort. Since Pharo 1.0, several actions were made:

- An effort to use an event-driven structure was made first by Michael Rueger then followed up by Igor Stasenko and Stéphane Ducasse. An input fetcher and a input handler were introduced. This new infrastructure was already validated by the fact that it is possible to build an independent UI event interpretation loop.
- Igor Stasenko built a wait-free collection so that the events are managed in a solid data-structure.
- We systematically fixed layer violations. Morphic code should not shortcut anymore the HandMorph logic and should not access low-level input sensor.
- Improved modal logic. We introduced abstraction to HandMorph to simplify the handling of modal interaction (dropping certain event while a certain condition holds). It simplifies the logic and encapsulates it.

Pharo 1.4 can now drop the polling semantics. All the VMs support now the event infrastructure. This moves could have been done earlier but the Windows VM did not integrate a fix adding a semaphore input during one and half year. With the new VM infrastructure (jenkins and GIT/CMake) put in place by Igor Stasenko. We can now compile daily on Mac and Linux. The Windows setup will follow soon.

Current ongoing effort.

- We are introducing a registration mechanism, so that it can be easy to build a record mechanism of events or other broadcasting systems.
- We are introducing a major refactoring (dashed line in Figure 4.2): the event fetcher interprets the eventBuf and creates SystemInputEvents. Such events

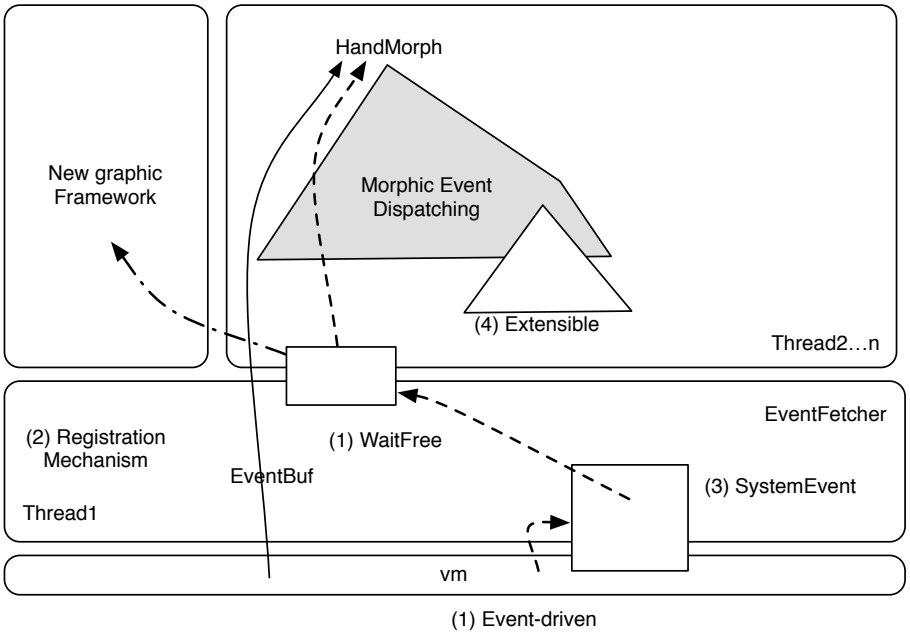

Figure 4.2: Current and future Pharo effort.

fully encapsulate the logic and behavior of low level events (avoid complex logic spread in HandMorph, duplicated code, ...) are then passed to the handler queue. Then HandMorph manipulates such objects and can decide to republish them.

What is left to do.

- We are rewriting the HandMorph logic to not use hardcoded case statements anymore. In fact there are at least two case statements (one for building morphic events from raw buffer, and one for defining the way morphic works). We have nearly fixed the first one using double dispatch.
- We should migrate HostMenus processing to an adequate place, the pasteUpMorph should be responsible to publish its menu. Now we may simply deprecate the host menu facilities because of development resource limitations.
- We should probably make sure that the libraries code that has interaction do it via Morph and not direct input sensor. For example, Rectangle fromUser should be packaged with Morph and use HandMorph API and not directly invoke input sensor.
- The second case statement logic should be rethought to support a smoother extensibility.

- We should validate the extensibility by evaluating how easy it is to introduce multitouch events and Genie in the system.

Open questions. Fernando Olivero proposed a new way to manage event and devices at the level of Morphic. Basically it acknowledge that HandMorph has too much responsibilities that the logic of the event processing is complex, brittle and difficult to extend. The model proposed by Fernando is based on decomposing responsibilities and using state pattern to identify double-click and other state-oriented situations. One question is do we propose a state-machine that can be integrated in HandMorph in a compatible way or is it better to define a separate infrastructure and after a moment to remove HandMorph. Frankly we do not have the answer (yes no magic ball here). Fernando already integrated his model with the refactoring we are performing. A possible path is to integrate our refactorings, finish the first two steps of the previous sections and evaluate how the solution of Fernando is a good alternative.

Business Value. Touch Event, MultiTouch and others like new user interface development can radically benefit from our effort. We should be able to plug easily multiple touch devices.

4.2 Rewrite of Filesystem/Streams

Context. Current file support is rather old, cumbersome and counter intuitive. Pharo as a scripting language should have a good and easy to use file library.

Possible Solutions. We want to use the FileSystem library. However, the state of FileSystem is not totally at the level required to support a replacement of the existing one.

- We started an effort to add comments in the code.
- We improved the FileSystem library.
- We started to write a documentation in a form of a chapter for the Pharo by Example volume two book.

What is left to do.

- Evaluate if the complete API is implemented (for example rename was missing).
- Migrate existing code to use the FileSystem library.

- Improve or add support for the Stream related API (newFileNamed, oldFileNamed:, readOnly...). A possible solution is to use the Xstream library but it should be discussed on the mailing-list.
- Finish to write a solid documentation to be part of Pharo by Example 2.

Potential Problems. Streams may be a problem or require a larger effort. It may be wise to proceed in a two step process: (1) first focus on the replacement of the file directory implementation - implement some backward compatible streams API using the existing stream library. (2) Assess whether Xtreams is the way to go. (3) Decide and migrate to Xtreams if decided.

It should be noted that going back to pure ANSI-streams as provided by FSFileStream is not really an option. FSFileStreams are pure ansi-protocol compliant streams. No buffering, no optimization, strange concoction of the underlying primitives (always pairing seek and read), no "built-in" way to translate using encodings,... Creating such streams from the FSPath stream creation API makes little to no sense, lest the goal is to provide a simple API for accessing streams in a cross-platform manner.

Since it's a clean break from FileDirectory though, it might be the perfect time to introduce XTreams; -or it might be too much hassle to have to change both parts in existing code at once.

A good compromise would be to make it pluggable, using the existing Standard/MultiByteFileStreams by default, but making XStreams pluggable (on a use–by-use basis, so you can write new code with XStreams while maintaining old code where you have only had time to rewrite the path handling).

4.3 Announcements and Ephemerons

The general question is: How can we get better weak structure and also announcement frameworks? In particular since infrastructure of the system will be based on Announcement, it is important that: (1) error in user code cannot break the dispatch of system announcement, (2) announcement registration does not impact garbage collector reclaiming of objects.

Context. In some cases announcement users need to pass a block that will be executed. Such block may refer to objects and as such it may be a problem because this blocks the garbage collector to reclaim such objects. This is why Announcements introduced a weak form but this is just one way to provide a possible solution. This approach has the disadvantage that the user needs to decide upfront the kind of registration he wants to perform.

Possible Solutions. An Ephemeron is an interesting data structure: it is a pair whose key holds strongly an element and a value which holds weakly an element [**?**]. When the value is reclaimed by the garbage collector, the key reference is turned into a weak reference. Using ephemerons we can then simplify the Announcement and make sure that clients do not have to worry about the type of subscription (action blocks or message sends) you are allowed to use when registering weakly.

In addition, ephemerons open an interesting range of applications to implement isolation as Mark Miller with Caja in JavaScript are demonstrating it. Finally it is worth to note that few languages support ephemerons: JavaScript, Lua, Haskell and other Smalltalk implementations.

4.4 UI Canvas for Zoomable Interfaces

Building decent graphical engines is now difficult in Pharo. Javascript often offers better rendering. How can we get decent zoomable canvas that can support 2010 UI technologies like openGL and Cairo?

Context. The current UI canvas is a bit old and shows its age. In addition the class hierarchy starts to get complex and messy.

Possible Solutions. Design a new canvas that is designed to talk to vectorial frameworks and have a default back-end as fallback. Migrate all the system to use the new API and canvas.

What have been done so far. Igor Stasenko did the following:

- Designed and implemented a new canvas API and all the necessary abstractions to generate adequate code for different backends.
- Deep integration to be able to inject frames into existing VM structure (Igor can you confirm this?)
- Designed and implemented a new textMorph, paragraph and other classes.
- Defined a default back-end.
- Defined a Cairo back-end.
- Defined an openGL back-end (using an OpenGL calling framework).

What is left to do.

- Finish the Cairo back-end for the canvas API.
- Finish the openGL back-end for the canvas API.
- Rewrite the existing Morph drawOn: methods to use the new API.

- Rewrite the existing call to the old API to the use the new API.
- Deprecate the old Canvas api.
- Document the design architecture so that other people can use it and extend it without breaking it.

Business Value. Applications such as Moose, Roassal, Mondrian can really benefit from it. Tablets application can also benefit from the zoomable and vectorial aspects of it.

4.5 Bootstrap of the Core

Context. The core of Pharo has not been build from a list of statements since far too long. We made progress recently and restarted to exercise a lot of code but being able to bootstrap a core will give us an extra level of agility and robustness.

- While we can produce a new image from an existing one, we have to apply all the sequences of modifications one after the other one. In addition, it may be difficult to get the system to a specific state (e.g., processes) before applying certain up- date.
- Certain classes have not been initialized since years. Code may rot because not systematically exercised. For example, in old versions of Squeak some initializing methods where referring to fonts stored on hard drive. Such a situation clearly showed that the system was not initialized from its own description and that these initialization methods were absolutely not executed since a couple of years.
- Since the system acts as a living system, it is not simple to evolve the kernel for introducing new abstractions. We have to pay attention and migrate existing objects (changing class representation for example). Some radical changes (e.g., changing the header size of objects) cannot be achieved by simple object changes (because objects with modified object format cannot co-exist in the same image) but require to use disc storage to create a new modified version of the system.
- Since the system is not rebuilt using a process that does not have to execute all the modification stream, it is hard to produce a system with only wanted parts. Current implementations often rely on image shrinkers which remove unnecessary parts of the systems. However, this process is tedious because of the dynamic nature of Smalltalk and the use of reflective features which breaks static analysis

Possible Solutions. We are working on a bootstrap of the core. Doing so we are cleaning dependencies, rot code, design problem leading to unwanted dependencies (a typical example was the old Preferences mechanism - now solved by the excellent and modular Setting framework described in Pharo by example two). It will bring the following properties:

- *Agile and explicit process.* Having a bootstrap is important to be sure that the system can always be built from the ground. It makes sure that initialization of key parts of the system (character, classes, ...) is exercised each time the system is built. It limits broken code; this is especially important in Smalltalk since classes are initialized at load time, but can evolve afterwards. It also makes sure that there is no hidden dependency. This supports the idea of software construction by selecting elements.
- *Warranty of initial state.* Currently, the evolution of a Smalltalk image is done by mutating objects in sequence: a stream of changes can bring an image from a state A to a state B. It is not always possible that a change bringing the system to a state C can be applied from state A. It may happen that B has to be reached, and then only C can be applied. Some changes may not be interchangeable and it may be difficult to identify the exact state of the system (for example in terms of running processes). Using a bootstrap process to initialize the kernel, we get the warranty to have a consistent initial state.
- *Explicit malleability and evolution support.* Having an explicit and operational machine executable process to build a kernel is also important to support evolution of the kernel. The evolution can be performed by defining new entities and their relation with existing ones. There is no need to build transition paths from an existing system to the next version. This is particularly important for radical changes where migration would be too complex.
- *Minimal self reference.* The self referential aspect of a bootstrap supports the identification of the minimal subset necessary to express itself. It forces hidden information to be made explicit (format of objects...). From that perspective, it supports better portability of the code basis to other systems.

4.6 Fully parametrized compiler tool chain

Context. The current compiler chain is dated. It is difficult to extend it. For example, we need to be able to support the following typical use cases:

- Compiling towards another byte code sets: for example compiling specific programs to legoOS.

- Supporting a bootstrappable compiler: we should be able to recompile the existing compiler with itself to be able to modify it.
- To support the creation of new kernels, we should be able to specify that the compiler should take certain environments as input and compile classes in an another one.

Possible Solutions. The compiler and the tools such as the refactoring engine, tests, system navigation should all be parametrizable by environments.

4.7 Packages as real objects

Context. PackageInfo relies on string matching to identify packages. It makes the system slows and brittle even after a couple of optimization passes. PackageOrganizer is filled up with incorrect information. For example, we would like to have senders and implementors showing always the package in which the method or class is defined to support better understanding. Some tools like OB have their own cache but there is no synergy with the other tools. It is not a good idea that each tool build its own cache.

Possible Solutions. Get a real object maintaining explicit information about the class and methods that the package contains. This solution avoids to scan all the method categories and class but it requires to do a bit more of bookkeeping.

What have been done so far. RPackage has been developed and rewritten three times. Therefore we learned and improved the design and implementation. It is now fully tested. RPackage is used by Moose and Nautilus since a year without problem.

We faced the problems of handling system event and this is why we built SystemAnnouncement. Now SystemAnnouncement is fully working. It needs to replace SystemChangeNotifier.

What is left to do.

- We should verify that loading a new package creates a correct RPackage. Normally this is done because the announcements are raised at the right place.
- We should revisit the API of RPackageOrganizer and PackageInfo and make it converges. We should deprecate part of the PackageInfo API related to category since there is no logic of category in RPackage.
- The migration from PackageInfo to RPackage was slow down because external tools such as Refactoring Browser used the API of PackageInfo. Since it was not clear how we can manage change in Refactoring Browser we started to

load it in the image but we did not get the energy to adapt RB to the change needed in RPackage.

- In addition the integration inside the system was slow down because of the current state of the PackageOrganizer and its logic (sometimes packages are not registered but appear later). Monticello relies on PackageOrganizer so care should be taken.
- A pass on the integration is needed to make sure that the Announcements are fully managed.
- The main problem that blocked the future development of RPackage was the fact that mapping category to packages will increase the number of packages. It should be noted that getting too many negative feedback ruins energy, especially when the work is done on free time.

Potential Problems. We will get more packages if we map category to packages. This means that configurations would have to be extended. RPackage supports the notion of tags. A class can belong to any number of tags. With tags we can freely represent classes sorting at the UI level inside a package, but tools should be adapted to reflect this.

Tudor Girba suggested:

Stage 1: Get RPackage in the image and pretend that it acts like a smart cache for categories. Everything is expressed in terms of categories (using MC *-notation) and mirrored in RPackage. Tools can now be built on top of RPackage. At this stage, no modification is needed and the new tools will work next to the old ones.

Stage 2: Leave the logic of Monticello loading as it is now (n-to-1 Categories-to-MonticelloPackage), but modify Monticello publishing to rely on 1-to-1 RPackage-to-MonticelloPackage. This will basically force people to start changing the configurations.

Stage 3: Change the Monticello loading to only allow 1-to-1 RPackage-to-MonticelloPackage. At this point, we can safely remove the Categories.

One pending issue is the loading and saving of packages once we got rid of PackageInfo. Indeed right now the system gets notified and still rely on * notation for loading and saving packages. For compatibility such notation should be kept but the saving and loading of package should handle the transformation from and into RPackages.

4.8 Package Meta-Data

Context. Since we want to validate package using specific rules and tests. We need a mechanism to describe false positives. No developer will use a tool that report endlessly the same errors. Currently SmallLint relies on adding pragma to method body but this solution does not scale. Nobody want methods with 15 pragmas about rules. In addition, we need a way to store package documentation and other meta-data.

Possible Solutions. We are currently experimenting with the introduction of a per package manifesto class.

Steps.

- We added a manifesto class per package.
- Using this manifesto we record the rules that should not be run on a package and we record also the rules that are false positive.
- We are building a user interface to tag false positive and manage rules.

Once a first validation on real cases will be performed we will announce it and gather feedback.

4.9 Less Model Clutter and Duplication

Context. We should stop to have models for program elements each time duplicated all the time. Example are: PseudoClass, FilePackage, MC* packages, MethodReference, RBEnvironment elements.

Possible Solutions. We designed Ring (a source code meta model) to serve as common basis for a large set of tools [**?**]. Ring is used instead of the MethodReference and CommentReference. It supports a code model that is polymorphic with Smalltalk run-time objects. Therefore a browser such as Nautilus is able to manipulate runtime classes and compiled methods as well as Ring objects (representing classes remotely accessing, on a disc...). It means that we should be able to use the same browser to remotely browse classes, browse files, and living objects on the system. No need for there different browsers.

Ring is an important effort. Veronica Uquillas-Gomez spent several months designing, implementing and testing it. This work has a real value for the community.

What is left to do.

- Analyze the current models uses and learn how they can be replaced by Ring. We did such analysis for pseudo class. Monticello seems to use PseudoClass and FilePackage for some of the code model representation.
- Convert use of models into ring ones.

Business Value. Using one single code meta model or as few as possible (and polymorphic with runtime objects) will bring less code to maintain, a large opportunity for reuse - for example browsers can be reused for runtime objects, files on disc or remote objects. In addition we will have less to synchronize.

4.10 Building and Reusing UI Logic

Context. Building user interface is tedious. Now existing approaches to offer builders only focus on the declaration (drag and drop) of the UI elements. Code is generated: it is not sure that the builder can be reopened on modified interface. Some UIBuilders rely on specific event mechanisms that would have to be added to Pharo. Then more important the reuse of the logic is not supported. For example, how can we reuse that an input field selects the elements of a list?

Possible Solutions. One interesting solution is to build application model based on valueHolders like in VisualWorks. Benjamin van Ryseghem already started a controlled experiment and it looks promising since he can reuse the logic of a given composition.

In addition, first class objects (a.k.a literal specs) representing the descriptions of the graphical elements like in VisualWorks or Microsoft is an interesting path because

- with specs: one model can propose multiple UI presentations.
- Specs have the advantage that they are not sensitive to instance variable changes as morph serialization is.

The effort made around Glamour is also really interesting but it works at another level. Glamour allows one to specify the flow between UI element: when clicking on a list, how another list is filled up. Glamour is not made to build widgets. However we are interested to see how Glamour can be used to prototype browsers or the basis for toolset.

4.11 New Network Layer

Context. The network support is improving over the years. Zinc, while at another level than the socket layer, is already a good improvement. The effort around Zodiac an https client/server is also important. Pharo since its version 1.3 integrates Zinc. Ocean is an effort to rewrite the Socket layer and to fully test it. We are really interested in such effort.

Steps. Luc Fabresse and Noury Bouraqadi spent a lot of time working on Ocean a new socket layer. We hope to integrate their work as soon as it is possible.

4.12 Serializers

Context. The serializers are a bit outdated, cumbersome and not that fast. Originally we thought that ImageSegment was the solution. We apply an scientific approach and Mariano Martinez-Peck proved to us that it was not. The analysis are described in a scientific publication [**?**].

Possible Solutions. Fuel is a fast and solid serialization framework [**?**]. Its development started in 2010 and the vision was that we need a good serializer to support faster package loading. Now Fuel is an object serializer.

Fuel has the following characteristics:

- Fully tested,
- Several years of engineering,
- Serious benchmarking,
- Clean design

Potential Problems. MC uses old serializers. One solution is to propose a converter between the two systems.

Fuel should handle well migration between different versions.

4.13 SystemChangeNotifier replacement

Context. SystemNotifier is a mechanism that we introduced with Roel Wuyts a while ago. The idea was that each tool requiring to change its state after a change of the system happens, just needs to register its interest to the SystemChangeNotifier. Historical note: before SystemChangeNotifier, tools had to identify and modify all the places in the system to get notify. It was obviously terrible. SystemChangeNotifier is a typical example of why we need a good backbone.

One of the problems we face with SystemChangeNotifier is that the event raised are not objects and the protocol is sometimes incomplete or unclear. In addition it duplicates the behavior of Announcements.

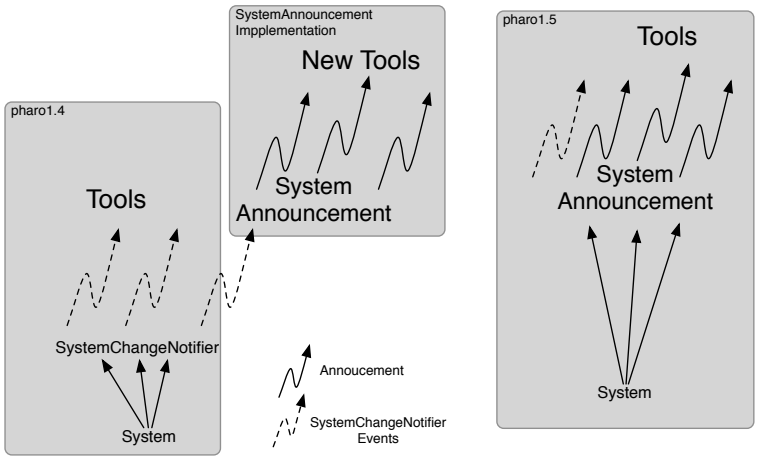

Figure 4.3: You should save your configuration in your repository. Then publish it when ready in the different repositories for validation.

Possible Solutions. SystemAnnouncement is a new announcement-based system notifier equivalent. As shown in Figure 4.3, SystemAnnouncement currently listens to all the notifications raised by SystemChangeNotifier and republish them as Announcements. RPackage is based on them as well as some Moose tools.

The second large step to be performed is to replace SystemChangeNotifier and use instead SystemAnnouncement.

http://code.google.com/p/pharo/issues/detail?id=5174 propose a list of steps.

- 1a) Internalize state in Announcements, instead of using data from Events.
- 1b) Implement SystemChangeNotifier protocol on SystemAnnouncer, in extension protocol *SystemEvents, so external consumers will register correctly when loaded (but receive polymorphic Announcements instead of Events).
- 1c) Implementing announcement-handling methods on current Event consumers (ChangeSet, SmalltalkImage, RecentMessageList, TestCase, TestRunner, etc.) in Core.
- 2) Migrate current Event-consumers from SystemChangeNotifier to SystemAnnouncer
- 3) Switch SystemChangeNotifier uniqueInstance to SystemAnnouncer.
- 4) Remove all contents in System-Change Notification except SystemChangeNotifier's class method.

- 5a) Make new package from SystemEvents extension protocol.
- 5b) Store SystemEvents package in external repository, remove from Core.

4.14 Cleaning Morphic

Context. Morph is still a complex frameworks with multiple facets and an organic growth that is difficult to tame. In Pharo we already did a large effort, removing a lot of unused, experimental features. There are still work to do.

Possible Solutions. Step by step taking opportunity to clean and improve the coherence of the system. CUIS performed some interesting cleans, now CUIS is not a panacea and some design decisions have to be discussed. Still reducing the system complexity and coherence is important.

Steps. The following actions should be done:

- Clean keyboard binding harcoded logic. The work of Guillermo Polito on keyboard is definitively important and should be integrated.
- Reduce unnecessary options.
- Clean the logic of system event processing. Using a state pattern as suggested and demonstrated by Fernando Olivero in Gaucho is interesting.
- Clarify morphic events triggering.
- Reduce the number of widgets (check the ones that are used/usefull) since there are some duplicates.
- Remove DependentFields from Object. Not every object should have dependent!

Potential Problems. We will have to address the UITheme / UIManager syndrome. Indeed UIManager has the logic of creation of certain widgets such as the debugger, its acts also as a UIBuilder. The relationship between such classes has to be clarified.

Business Value. Building different users interface is important. Morphic is flexible and so far there is no other framework available with this large range.

CHAPTER 5

Important but uncertain points

We are experts in many aspects of Pharo and the community has a strong set of contributors in many topics. Still there are aspects where we are not in full control of the options. A typical case is the status of the Virtual Machines and the C interaction. This is typically for such situations that having a strong consortium can help the community to get a momentum and build its own safer future. Building a more manageable Virtual Machine should be one major goal of the consortium.

5.1 Everybody should be able to compile VMs

Context. Up to last year it was nearly impossible to compile a virtual machine out of the box without a deep knowledge about the VM itself. This was a real problem because it hampered programmers to be autonomous, learn, improve the system. For example, some important fixes (such as input semaphore) took more than 18 months before being integrated in the Windows virtual machine. There was a real need for expanding the knowledge about the virtual machines.

What have been done so far. This year, Igor Stasenko as its part of his job at INRIA, built an infrastructure to let everybody recompile easily all the existing VMs. Lectures were given during the Deep into Smalltalk seminars as well as at ESUG. Following a simple and short list of steps, the participants could compile the virtual machine for their architecture.

This infrastructure is key for us because we want to make sure that people can compile their own VM and learn VM technology. In addition the infrastructure put in place is built on GIT so that people can freely experiment and propose fixes.

5.2 VMs identification and regression testing

Virtual machines do not include exactly the same fixes or behavior. For example, the behavior on finalization. Users can experience not adequate behavior when using a VM that is not one coming from our build. Therefore we need

- to identify better the VM and their subtle differences. It will improve the bug tracking.
- to set up an infrastructure for VM regression testing. This implies to build dedicated test cases.

5.3 One Unified FFI framework

Context. Interacting with the external world is an absolute need. We want to be able to use libraries written in C. Now the current state of FFI solution is totally messy. Really few people knows really what is working where and what are the features available: callback, threaded...

Right now we have:

- FFI for call-outs.
- Alien for call-backs
- NativeBoost for in image assembly generation of faster call-outs with threaded calls in option.

Getting FFI supported threaded calls is developed by E. Miranda in Cog but its exact status is unclear to us.

Possible Solutions. We need *one* framework. We would like to have Spock: one FFI library composed out of two systems exhibiting the same API: a better version than FFI and NativeBoost. Indeed, we need one solution where the FFI support is developed in C so that we can handle new CPU. This is important for strange architecture. It is often simpler to recompile a library than to build an assembly generator dealing with alignment and other gorgeous subtleties. In addition, we should take advantage of NativeBoost since NativeBoost generates faster code than FFI, it supports threaded calls and callbacks.

Now it is not clear that the better version of FFI will be available so at least we should have a fully working version of NativeBoost which is central for the OpenGl Cairo new UI Canvas.

Business Value. There is huge business value. Pharo should be able to talk to any C-library.

5.4 64 Bits

The availability of the VM for 64 bits architecture is also an important point. Its impact on FFI functionality should be evaluated.

5.5 New Object Formats

The current object header has only 12 bits for hashes and this is limiting us. E. Miranda mentioned that he wants to propose a new object header and we are really interested in such changes. In addition, we need to be able to play with encodings of new information such as immutability and the current header format does not let us doing that.

CHAPTER 6

And you!

We are working hard to make our vision happening. We are doing mistakes but learning from them. Doing something everyday a bit is the best way to make a better chance to things to happen.

Realize that with 300 little steps you can do something big. Rhythm is important because it gives a beat. This is why we integrate daily fixes in Pharo. Every single day Pharo gets better.

Now you can have an impact: you can help!

- We are setting up a consortium with the help of INRIA. Financially: you will be able to join the Pharo consortium to support the salary of a full time world class engineer: Igor Stasenko (NativeBoost, Athens and tons of Pharo improvements).
- Giving your time:
  - Ask questions in the mailing-lists, there is no such thing as a stupid question, do not hesitate ask,
  - Provide code, join this is fun,
  - Improve our web presence, blogs, tweets are important,
  - Test proposed fixes, knowing that a fix does not solve the problem is a really important information for us,
  - Enhance comments,
  - Fix the english of documentation and books (yes most of us are not native)

A final point, some of you may think that we know everything and that they cannot help because too stupid. This is totally wrong. What you should consider is that we are learning a lot. This is the power of Smalltalk. Often we go over a piece of code several times to get used to it, slowly understanding it, chewing it until we get it. Then we think about what we do not like and slowly fix it. Taking any opportunity to learn is what we do. This is why looking at code fix is the best way to help and learn. If you do not get a bug, pass to the next one. Remember we are doing that since Squeak 3.4 and every day we are learning new things.

Thanks

We want to thank all the people with which we discussed over the years, the committers and pharoers that support us in this vision and effort. Thank you guys!

Special thanks to Sean De Nigris and Ben Coman for capturing our english mistakes. Thanks Philippe Marschall, Henrik Johansen, Milan Mimica, Igor Stasenko, Mariano Martinez-Peck, Serge Stinckwich, Nicolas Cellier, Dimitry Chioupis, Norbert Hartl, Krishnamachari Sudhakar, Wilhelm Schwab, Nick Ager, for feedback on the earlier versions.

INRIA is spending a lot of resources for Pharo. We specifically want to thank INRIA for its constant support and in particular we want to thank the INRIA Lille Nord Europe Center team.