Towards a new package dependency model

Christophe Demarey
RMoD Project-Team, Inria Lille–Nord Europe
christophe.demarey@inria.fr

Damien Cassou
RMoD Project-Team, Lifl, Université de Lille 1, Inria Lille–Nord Europe
damien.cassou@univ-lille1.fr

Stéphane Ducasse
RMoD Project-Team, Inria Lille–Nord Europe
stephane.ducase@inria.fr

Abstract

Smalltalk originally did not have a package manager. Each Smalltalk implementation defined its own with more or less functionalities. Since 2010, Monticello/Metacello[Hen09] one package manager is available for open-source Smalltalks. It allows one to load source code packages with their dependencies. This package manager does not have all features we can find in well-known package managers like those used for the Linux operating system. This paper tries to identify the missing features and proposes solution to reach a full-featured package manager. A part of this solution is to represent packages and dependencies as first-class objects, leading to the definition of a new dependency model.

Keywords package management system, package manager, dependency

1. Introduction

In this presentation, the *package* term will be used to depict a shippable piece of software (something that you can deliver) and will not be related to system packages nor source code Version Control Systems (*e.g.*, Monticello), two commonly used package meanings in Smalltalk. Smalltalk (Gemstone, Pharo, Squeak) did not have a package manager for years. Indeed, with the image paradigm, the need of a package manager was not seen as a first need. With time, the need of a package manager comes up to have minimal images where you can load only packages you need. In such situations, a package manager is really important to be able to load all required packages (including transitive dependencies). Loading transitive dependencies implies that packages describe their dependencies.

Since 2010, Metacello provides a solution to manage packages and their dependencies [BCDL13]. It was a huge effort towards a modular system where you can load additional application, libraries and their transitive dependencies. Metacello also evolved year after year, adding new features. At this time, we have a better comprehension of what is working with the Metacello package manager and what is missing or should be improved. We can also compare this package manager with other languages. The goal of this paper is to describe improvements, new functionalities we would like to have for the next generation package manager. First, we describe missing functionalities and then propose a solution for each one. To conclude, we depict future work in this area.

IWST'14, August 18-22, 2014, Cambridge, England.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

2. Problem Description

> "A package management system, also called package manager, is a collection of software tools to automate the process of installing, upgrading, configuring, and removing software packages [...] in a consistent manner. It typically maintains a database of software dependencies and version information to prevent software mismatches and missing prerequisites.", *Wikipedia*

This explanation of a package management system is more operating-system oriented but also applies quite well to a programming language. From a language point of view, what can you expect from a package manager? Here is a non-exhaustive list of wished functionalities for a package manager:

- ensuring system coherence: install, update, remove packages without breaking installed packages. If a conflict occurs, the package manager should explain the conflict and give the user the choice on how to solve it,
- first-class dependencies: packages and their dependencies should be first-class objects, easily accessible from other tools,
- synthetic package descriptions: provide a quick way to create new packages and express their dependencies through short and easily understandable descriptions,
- automated load order computation: automated computation of the load order from the dependency graph.
- knowledge on the system state: a user should be able to query on installed packages in the system,
- knowledge on available packages: a user should be able to query on available packages for the system,
- solving review: allow the user to review what will be installed before actually install packages,
- reproducible loading: once the solving done, the user may want to reuse this solving to load the same set of packages on identical systems (e.g, on the N production servers),
- conditional loading support: filter out packages that cannot be installed in the system (e.g. requires a specific version of the system),
- complex constraints support: declaring a dependency to a fixed version is not enough. A package manager has to allow more constraints like a version range (between version 1.2 and version 1.6, greater than 2.0) or boolean expressions,

- support of update strategies: these strategies will enables automatic updates of compatible packages (e.g., security or bug fixes),
- independence from Version Control Systems: Dependency descriptions should refers to versionned packages and not source code artifacts.

We can see that functionalities expected from a package management system for a programming language are quite the same as functionalities expected for an operating system.

To come back to Metacello, we present an overview of the functionalities offered by package management systems for open-source Smalltalk and some other well-known package management systems.

Functionnality	Metacello	Maven	apt
ensuring system coherence	Partial[1]	Partial[2]	Yes
first-class dependencies	No	No	No
synthetic package descriptions	No	Partial[3]	Yes
automated load order computation	No	Yes	Yes
knowledge on the system state	No	Yes	Yes
knowledge on available packages	No	Partial[4]	Yes
solving review	Yes	Yes	Yes
reproducible loading	Partial [5]	Partial[6]	Yes
conditional loading support	Yes	N/A	No[7]
complex constraints support	No[8]	Yes	Yes
support of update strategies	No[9]	Yes	Yes
independance from VCS	No	Yes	Yes

Table 1. Package management systems functionalities

In the following subsections, we focus on functionalities not yet covered by Metacello, the only tool available for open-source Smalltalks.

2.1 Ensuring system coherence

When installing a new library (or updating an already loaded library) into an image, there is no guarentee that already installed libraries will continue to work. There is no record of which configurations are already installed in the image. Configurations can be collected in the image but it is impossible to know if a particular configuration has been loaded and which version was loaded. The record of such information is very important to allow tools and users to query about installed software. This information is also primordial to install new libraries.

Let's take an example: the package A is already installed in the system and depends on B v1.1. We want to install the package D that has a dependency to B v1.2. There is a conflict: we cannot have both B v1.1 and B v2.2 in the image. At the present time, if we request to install D, B v1.2 will be loaded in the image and A will be broken! This should not happen. Installing a new package into the system should take into account what is currently loaded and what may break! The same problem applies when trying to update an installed package into the system.

[1] Metacello provides hooks to execute code on upgrade or downgrade of an installed package

[2] Maven takes decisions that may lead to an inconsistent system. However, the user can force this decisions.

[3] XML is verbose

[4] Maven has a central repository but does not use it to propose packages.

[5] The use of symbolic versions leads to unreproducible loadings

[6] not possible by using SNAPSHOT dependencies

[7] There is one specific central repository per platform

[8] Only fixed versions supported

[9] Symbolic versions may be used for updates but without warranty on the backward compatibility.

2.2 First-class dependencies

Packages and their dependencies should be first-class objects. The package management system as well as other tools need a quick access to these information. First-class dependencies will facilitate the work needed to go to a modular system, open doors to new tools (e.g. automatic update of dependencies information from the source code). In the current package manager used by open-sources Smalltalk, three kind of dependencies [10] are used to express software dependencies: dependency referring to a project (piece of code with a dedicated description/configuration), to a package and to a group that is a collection of projects, packages or groups. These notions are very close and it is not always easy to understand the subtle differences between these concepts. It will increase the readability to merge them in a common concept. First-class dependencies will also enable the knowledge on the system state (installed packages). With first-class dependencies, we can also add more information on packages, i.e., more meta-data. Currently available meta-data, stored in Configurations, are:

- the package version author,
- the package version description,
- the package version timestamp,
- and the package version blessing: the tag used to manage symbolic versions (development, release).

We would like to add useful information such as the project license, a brief project description, the project website url, the project inception year, a link to the issue management system, a link to the mailing lists, the list of developers/contributors, etc. Other tools (or users) can use such information to choose the package fitting their needs. For example, Maven pom files[11] provide a lot of meta-data for each project.
All this information on packages needs to be loadable easily without loading the package itself and without installing any new code in the system. Indeed, it is very strange to modify the image state by loading new classes (Configurations) to only read some information on packages. It may also be dangerous if the loaded code overrides some existing code in the image. Storing meta-data in methods of a class allows versioning of meta-data only if you use Monticello as Version Control System, and not with any other VCS. VCS are able to version any kind of data: source code, images, text, binaries. Moreover, you can keep each version of your meta-data easily by publishing them to a central package repository, even with Monticello.

2.3 Synthetic package descriptions

Package descriptions are currently cluttered with a lot of specific dependencies hard to handle: platform specific packages and test packages.

2.3.1 Management of platform specific packages

Currently platform-specific code goes into a dedicated package. Then, you still need to tell Metacello which package to load according to the targeted platform. It is done with the `for:do:` message as shown in the following snippet.

[10] See DeepIntoPharo (http://deepintopharo.com), Managing projects with Metacello chapter

[11] http://maven.apache.org/pom.html

```
spec
  group: 'Core'
  with: #('CoolBrowser-Core' 'CoolBrowser-Platform')
spec
  for: #gemstone
  do: [ spec
        package: 'CoolBrowser-Platform'
        with: 'CoolBrowser-PlatformGemstone' ]
spec
  for: #pharo
  do: [ spec
        package: 'CoolBrowser-Platform'
        with: 'CoolBrowser-PlatformPharo' ]
```

Listing 1. Platform packages management example

This information is redundant and clutters the package dependencies description. It should be simplified. You should just say that CoolBrowser-Core requires a platform specific package. The package manager should be smart enough to choose a package fitting the platform requirements.

2.3.2 Management of test packages

Tests are also often put in a dedicated package to allow the loading of a library with or without tests. Metacello does not provide an option to load (or not) test packages. It implies that the developer has to provide a way to load the code with or without tests by himself in the configuration. Here, again, the package dependencies description will be cluttered by groups defined to load tests or not. To summarize, current package dependencies description is far too verbose and should be simplified.

2.4 Automated load order computation

In current package descriptions, the developer needs to explicitly specify which packages should be loaded before the described package. Indeed, the whole dependency graph may contains cycles. With cycles, it is difficult to know which package to load before the others. Most cycles are introduced by the proliferation of specific packages: tests packages, platform-specific packages. The definition of the packages load order is delegated to the developer. It is more work to maintain a consistent load order and also more error prone. It would be good that the package management system gets this preoccupation to free the developer mind and to lighten the package descriptions.

2.5 Complex constraints and update strategies support

Metacello is a first step towards a better modularity. It enables to load quite easily packages and their dependencies. To achieve that, you need to specify which version of a package you need. Actually, Metacello supports only one kind of constraint: exact version match and allows only one constraint per package. It means you are locked to a specific package version. If you want to use a more recent version, you need to update your configuration. It is not totally true because Metacello allows the use of symbolic versions like #development, #stable or #release. These symbolic versions are useful to easily get the latest stable version of a package or to get the development version. The use of symbolic versions is not restricted to these use cases. For example, you can specify that your package relies on the latest stable version of another package. It introduces some flexibility but not enough. We need to express more constraints on package dependencies like: =1.0, =1.0 or 1.1, <2.0, >2.0 and <2.5, etc. More constraints and more constraint kinds will offer new possibilities but there is a price to pay for that: constraints solving will become more complex and time consuming. With only one possible path to follow (each constraint is solved to a specific version and only this one), the current solving can use an ad-hoc algorithm and be quite efficient. With the introduction of non-fixed dependency version constraints, the number of possible pathes explodes and we need an efficient algorithm. Solving a constraint satisfaction problem on a finite domain is an NP-complete problem in general. It implies to use a dedicated solver for this problem. To introduce automated update strategies, we need to know which versions are only bug fixes versions, which versions are backward compatible and which one are not backward compatible. This information is currently not available for packages.

2.6 Reproducible loading

A package management system has several responsibilities:

- allow the user to express requests (installation, update, removal) on packages,
- find a solution (if any) to the user request,
- and apply this solution.

The solution to a user request, e.g. a package installation, is called a dependency resolution. This dependency resolution should be serializable and reusable. Indeed, the solving can be made one time and the solving result can be used many times in possibly many images. That way, we ensure that the exact same set of packages will be loaded into different images. It is very convenient, for example, to ensure that packages installed in the production image(s) will be exactly the same packages deployed in the development image. Such a solution also allows a decoupling between the solving part and the loading part. You can imagine a minimal image with no solver but able to load already solved dependency resolutions.

2.7 Independence from Version Control Systems

A big drawback with the current description of dependencies, is that descriptions are coupled with the legacy Smalltalk Version Control System: Monticello[12]. Indeed, to express dependencies, you need to reference Monticello zip files (mcz files).

```
spec
    package: 'CoolBrowser-Core'
    with: 'CoolBrowser-Core-BobJones.20'
```

Listing 2. Example of explicit reference to a VCS

In the previous example, CoolBrowser-Core-BobJones.20 refers to the CoolBrowser-Core-BobJones.20.mcz Monticello file. With the emergence of the git[13] Version Control System (VCS), a new way to express dependencies comes up allowing the developer to declare dependencies without specifying a specific version of these dependencies. In fact, the default dependency version is the head of the Version Control System but it can be set by specifying a specific repository URL like bellow:

```
github://demarey/metacello-work:1c8c138a7be...
```

Listing 3. URL used to refer to a specific version in a git repository

Despite the fact that several VCS are supported (git, Monticello, flat files), current package descriptions are closely tied to VCS: in numbered versions, you have to explicitly reference a particular artefact of the supported VCS, in general an mcz file name. We need to find a way to decouple the dependency description, and overall packages distribution from the VCS. We can also ask ourselves Why is it coupled? Pharo does not deliver binary packages (even if it is the

[12] http://www.wiresong.ca/monticello/

[13] http://git-scm.com/

case in professional environments such as VisualWorks [MLW05]) but rather packages with the source code. It can explain why there is a coupling between the package distribution and the Version Control System, but it should not be coupled! Source code versions are not the same concept as deliverable package versions.

3. Proposed Solution

This section will expose solutions for each problem exposed above. Even if each problem is seen as an individual case, all solutions put together describe a coherent approach.

3.1 Ensuring system coherence with installed package information

The solution to avoid broken libraries after installing/updating a software is very simple: we need to keep information about installed packages / software into the image. With a good object model of these dependencies, other tools will be able to use this information to ensure the coherence of the system. The simple proposition is to create a Package Registry with the responsibility to register all packages loaded into the image, and of course all meta-information on these packages. This registry will be used by other tools to ask for installed software, but also to get input for dependency solving.

For example, we can imagine different strategies to solve a software installation. Indeed, a sofware installation request can be translated to a constraint satisfaction problem. One strategy to solve a software installation could be to *minimize the number of updated packages and the number of new packages to install*. To implement this strategy, you need to know what is already installed in your system. It takes more importance when you need to install a software without breaking those already installed. If we use the example exposed below: *package A is already installed in the system and depends on B v1.1. I want to install the package D that has a dependency to B v1.2.*, if A can only use the version 1.1 of the package B, we need to add this constraint before starting to solve dependencies. The package registry will help to find the constraints we need to add to the dependency solving to keep the system in a coherent state, i.e., with all packages / software working.

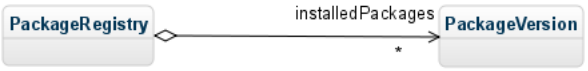

Figure 1. Package registry

We can imagine that such package information can be stored in the package Manifest (a package manifest is data class storing information about rule false positives) and extracted on demand to be published on package catalog and other external package description systems.

3.2 First-class dependencies

Having first-class dependencies in the image implies to extract the core concepts manipulated by software dependencies. Is a dependency to a package of your project the same kind of a dependency to a package outside your project? A package represents a piece of software you want to distribute or use. This piece of software may be something you developed or something coming from outside your project. There is not really a difference. Then, how to represent a group of pieces of software, i.e. a group of packages? A group is just a meta-package: a meta-package does not contain actual software, it is an empty package that simply depends on other packages, thus forming a group. With packages and meta-packages, we have an uniform representation of dependencies.

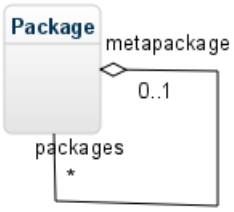

Figure 2. MetaPackage representation

Instances of the model introduced below describe packages with information on dependencies but also other meta-data. Package descriptions will be represented with objects in the image but we also need to store them with the source code. To achieve that, we need to serialize and also, deserialize these objects. Package meta-data needs to be easily accessible without being obliged to load the package itself or to install new source code in the image. To reach this goal, we need to define a serialization format for these meta-data. This format should be easily loadable and saved into/from the image and, if possible, easily human-readable. A good solution is to serialize packages meta-data with STON[14], a Smalltalk variant of the well-known JSON standard. STON is quite readable, close to a standard and provides automatic serialization/deserialization of objects.

Here is an example of what could be a serialization of a package metadata:

```
Package {
  #name : 'Seaside',
  #version : 3.1.0,
  #description : 'The framework for developing
    sophisticated web applications in Smalltalk.',
  #website : 'http://www.seaside.st',
  #dependencies : {
    'Grease' : 1.1,
    'Seaside-Core' : 3.1.0,
    'Seaside-Canvas' : 3.1.0,
    'Seaside-Session' : 3.1.0,
    'Seaside-Component' : 3.1.0,
    'Seaside-RenderLoop' : 3.1.0,
    'Seaside-Tools-Core' : 3.1.0,
    'Seaside-Flow' : 3.1.0,
    'Seaside-Environment' : 3.1.0,
    'Seaside-Widgets' : 3.1.0
  }
}
```

Listing 4. New serialization example of a package metadata

As package meta-data are outside the image, we need to find a way to store it with Smalltalk source code for legacy VCS (e.g., Monticello). The easiest solution would be to include the STON file into the mcz file that is a zip file but this solution will imply to transfer the whole mcz file to only get the metadata. It may be slow with a low bandwidth. Another option could be the creation of a specific Monticello package to hold these meta-data. The specific mcz will contain nothing but the STON file and Monticello meta-data. This way, the solution is still compatible with Monticello and can retrieve packages meta-data quite efficiently. However having empty package from a programmer point of view can be confusing.

[14] https://github.com/svenvc/ston/blob/master/ston-paper.md

Managing two packages for a package and its description is not optimal.

3.3 Synthetic package descriptions

With first-class dependencies, we have a nice dependency model in the image but we still need to find a way to handle platform-specific packages. The Debian operating system introduced the notion of virtual packages[15] for its package management system.

> A virtual package is a generic name that applies to any one of a group of packages, all of which provide similar basic functionality. For example, both the tin and trn programs are news readers, and should therefore satisfy any dependency of a program that required a news reader on a system, in order to work or to be useful. They are therefore both said to provide the virtual package called news-reader.

A virtual package is some kind of under-specified contract. Indeed, the contract only relies on the virtual package name and has no description. Some packages require this contract and some others provide it. Virtual packages should be used carefully because there is no verification that a package really implements the contract needed. There may also be some naming comflicts if appropriate names are not chosen. Beside that, virtual packages offer great features such as a loose coupling between packages. The package manager can choose the best package providing a virtual package according to the specific user request and environment. This low coupling avoids to predict all potential cases in the package description.
The idea is to use virtual packages to manage platform-specific packages. If a package Foo needs platform-specific packages, then it should declare a dependency to the Foo-Platform virtual package. The package implementor then needs to create platform-specific packages (e.g., Foo-Pharo, Foo-Gemstone), each providing the Foo-Platform virtual package. At the solving time, the package manager will search in the repository for all packages implementing the required virtual package. Of course, a virtual package will also have a version to choose an appropriate version of the virtual package. To work properly, packages (and as a consequence virtual packages) need to define requirements. Those requirements will be checked to see if a package can be installed on a given platform (e.g., Foo-Pharo requires the Pharo platform). Requirements already exist with Metacello and are named platformAttributes. By checking package requirements, the package manager will see that the package Foo-Gemstone cannot be installed on Pharo, and then the Foo-Pharo will be selected. The package description becomes shorter and cleaner.

Here is an example of a legacy description:

```
spec for: #common do: [
  spec
    package: 'Foo' with: 'Foo-Platform';
    group: 'default' with: #('Foo' 'Foo-Platform') ].
spec for: #gemstome do: [
  spec
    package: 'Foo-Platform' with: 'Foo-Gemstone'].
spec for: #pharo do: [
  spec
    package: 'Foo-Platform' with: 'Foo-Pharo'].
```

Listing 5. Legacy description for platform-specific packages

The Foo package becomes:

```
spec
  requires: 'Foo-Platform'.
```

Listing 6. Foo package description

The platform-specific package becomes:

```
spec
  provides: 'Foo-Platform'.
```

Listing 7. Foo-Pharo and Foo-Gemstone package description

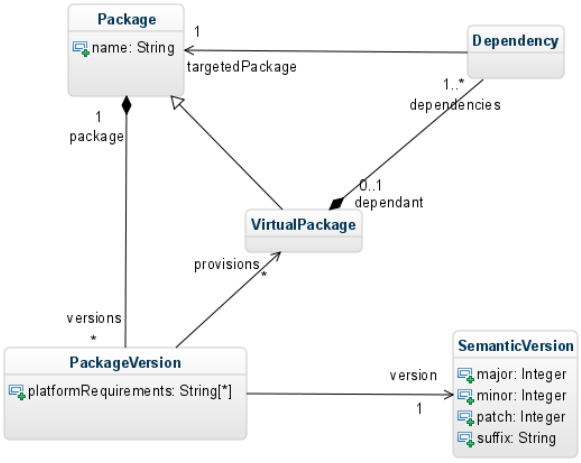

Figure 3. Virtual package modelization

With the unification of dependency description and the introduction of virtual packages, descriptions become smaller and easier to read, write or maintain.

To handle properly test dependencies (but also other kind of dependencies like development dependencies), we propose to define a scope to dependencies. A dependency may be needed to run tests but not at runtime, another could only be useful to develop the package. A dependency scope is in fact a kind of dependency. We propose to define a core dependency class and specialized versions of this dependency: *runtime dependency*, *test dependency* and *development* dependency.

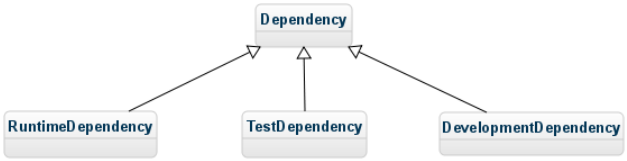

Figure 4. Dependency scopes

3.4 Automated load order

The biggest problem to enable automated load order computing by the package management system is the presence of cycles in the dependency graph. The analysis of several projects such as Seaside [DRS+10] showed that most cycles involve platform-specific packages. If we omit these dependencies, it is hard to find a cycle in a dependency graph. The solution proposed is to consider platform-specific packages as part of the core package.

Let's take an example: there is a Foo package. If we find a Foo-Tests package, we should consider that this package is part

[15] http://www.debian.org/doc/manuals/debian-faq/ch-pkg_basics.en.html

of the Foo package. The platform-specific package is in a different package for technical reasons (selective loading) but conceptually, the platform-specific and the core packages represent the same package. At the loading time, this conceptual representation will be translated into a batch loading of these packages. Indeed, they depend on each other, and then should be loaded at the same time. With this approach cycles should be removed from the dependency graph. If there are still some cycles, it may highlight a design problem.

3.5 Complex constraints solving and update strategies support

To be able to express more sophisticated constraints than *'I depend on the package foo in the exact version 1.1'*, we need to revisit the dependency description format to allow more expressivity. For example, if I need a version of B at least equals to the version 1.1, I will write B >1.1. Such constraints will imply to solve NP-complete problems and thus to use proper solvers. To ease package updates and packages descriptions, we can also take advantage of the semantic versioning[16]. It is nothing else that a convention to follow to number versions of packages. By following these conventions, we will be able to perform automatic updates like bug fixes updates, security issues update because version numbers will give us information on backward compatibility. For example, v1.2.4 will be compatible with v1.2.3, v1.2.1 and v1.2.0 (bug fixes versions) but also with the v1.* versions. On the other side, it will not be compatible with the v2.* versions. More sophisticated constraints and the adoption of a versioning strategy: semantic versioning will open new doors to dependency management. It should lead to less package versions, and at least less package description. It is also important to notice that tools can help to ensure the coherence of the versioning strategy (e.g., forbid the use of minor/patches version if an API change is detected).

3.6 Reproducible loading

To enable reproducible loading, we need to serialize dependency resolution. Such a file is named a lock file in the Composer dependency manager (for the PHP language). Composer writes the list of the exact versions it will install into a composer.lock file. It locks the project (package) to those specific versions. This mechanism is useful to save resolution time and to be sure to install exactly the same set of packages on the same machine or on other machines. It is similar to the Snapshotcello behavior which freezes the versions. It also enables the installation of packages into an image that does not have a solver.

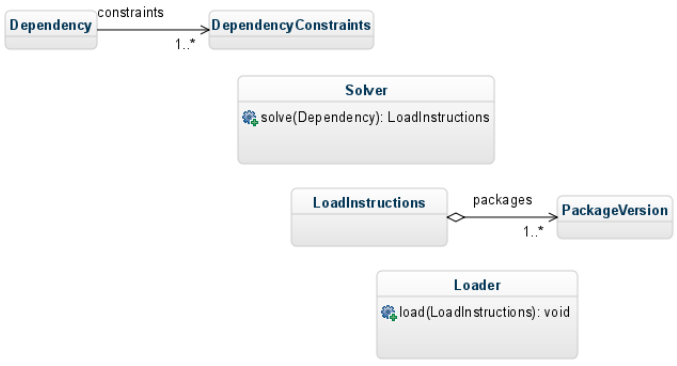

Figure 5. Solver and Loader decoupling

[16] http://semver.org/

As you can see on the class diagram, a Solver will take as input a dependency that is in fact a constraint or a set of constraints on one or more packages. The result of a solving is a LoadInstructions object having an ordered collection of all packages (with their specific versions) to load. This object may be serialized (or not) and then given to the package loader to actually perform the load. We propose to use the STON format to serialize LoadInstructions objects. The STON format will be already used to store package meta-information.

3.7 Independence from Version Control Systems

The best way to decouple package distribution from the source code / VCS is to set up a package repository where packages will be published. Published packages will have an independent version numbering. Many languages adopted this approach Java with the Maven central repository[17], Ruby with RubyGems[18], Perl with CPAN[19], Python with PyPI - the Python Package Index[20], PHP with Packagist[21], JavaScript with Bower[22], . . . This approach allows a great decoupling but also open doors for added-value features:

- a central place for the community to share artifacts,
- a central place to search for existing libraries,
- a central place to find (meta-)information on libraries.

For such a service, a preoccupation may be the duplication of the source code (in the VCS and in the central repository) and the storage needed. Both issues can be solved easily: there is no need to duplicate data. We can simply publish a description of the package version in the package repository. This description will contain all information needed to get sources directly from any VCS.

4. Conclusion

In this paper, we looked at the package management system challenges. We proposed different solutions to address these problems:

- have first-class objects for package dependencies in the system,
- use virtual packages to handle the complexity of platform-specific code,
- set up a repository dedicated to host packages and decoupled from Version Control Systems,
- support complex constraints support,
- adopt the semantic versioning.

The adoption of these propositions will increase the solving complexity[23] but, on another side, will offer a lot of facilities and new functionalities to the developer. Package descriptions will be easier to write and maintain, we will be able to do automatic updates, check the system coherence, and last but not least get a central repository for Pharo/Gemstone libraries. It will enhance the

[17] http://search.maven.org

[18] https://rubygems.org/

[19] https://metacpan.org/

[20] https://pypi.python.org/pypi

[21] https://packagist.org/

[22] http://bower.io/search/

[23] when a user request a package installation or update, the package manager needs to find a solution (a set of package versions to install) fitting expressed constraints (direct dependencies constraints) and transitive constraints (constraints expressed on transitive dependencies, i.e. dependencies of dependencies of the package to install). If constraints are not strong, the numbre of pathes to explore by the solver is huge and it is more complex to find a solution in a reasonnable time.

share and reuse of libraries by giving visibility on these projects. This model is adopted by a wide range of other languages.

Further work will target complex dependencies solving. There are two solutions: implement a new solver, re-using or not existing pieces, or use an existing solver. The last option is tempting because we will benefit for years of experience of specialists. Roberto Di Cosmo (Université de Paris VII) led an European project named Mancoosi[24]. The Mancoosi project defined a common dependency description format for Linux distributions: CUDF[25]. CUDF descriptions can be used as input data for many solvers. It would be a good idea to see if our dependency model could be converted to CUDF and thus, be able to use solvers already used to solve Linux dependencies. This work just has been done by the OCAML community with OPAM[26].

References

[BCDL13] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Square Bracket Associates, 2013.

[DRS+10] Stéphane Ducasse, Lukas Renggli, C. David Shaffer, Rick Zaccone, and Michael Davies. *Dynamic Web Development with Seaside*. Square Bracket Associates, 2010.

[Hen09] Dale Henrich. Metacello, a package management system for smalltalk [software]. https://github.com/dalehenrich/metacello-work, 2009.

[MLW05] Eliot Miranda, David Leibs, and Roel Wuyts. Parcels: a fast and feature-rich binary deployment technology. *Journal of Computer Languages, Systems and Structures*, 31(3-4):165–182, May 2005.

[24] http://www.mancoosi.org/

[25] http://www.mancoosi.org/cudf/

[26] http://opam.ocamlpro.com/