Understanding package management by analogy to programming languages

March 12, 2015

1 Introduction

Package management is an area that lies somewhere in the border between programming languages, operating systems and system administration. For this reason, it seems to be overlooked by all three fields as an implementation issue. In the meantime, package management keeps growing in complexity. New languages, new deployment models and new portability requirements, all give rise to new package management systems. Further, this is not simply a matter of competing implementations: modern complex environments often require several package managers to be used in tandem.

For example, when writing JavaScript web applications on a Mac environment, a developer may require using Bower [1], a package manager for client-side JavaScript components. Bower is installed using `npm` [5], a package manager for `node.js` [4], a JavaScript environment. On a Mac system, the typical way to install command-line tools such as `npm` is via either Homebrew [2] or MacPorts [3], the two most popular general-purpose package managers for Mac OSX. This is not a deliberately contrived example; it is the regular way to install development modules for a popular language in a modern platform.

In [7] we have another example of a typical software stack, where a deployment and management scenario for Ruby on Rails applications is described combining a number of tools. It uses Vagrant [8] for virtual machine management; Puppet [6] for editing system configuration files and driving the system-wide package manager on servers; Capistrano for deploying the Ruby on Rails application, including installing Ruby scripts and migrating database tables, driving RubyGems, the language-specific package manager for Ruby modules (with Bundler to mitigate module version conflicts); and RVM for managing conflicting versions of Ruby itself. It is interesting to note the number of different tools being used on top of each other to manage containment and compatibility issues on various layers; and again, this is a typical, realistic scenario.

The combinations of package managers change as we move to a different operating system or use a different language. Learning one's way through a new language or system, nowadays, includes learning one or more packaging environments. As a developer of modules, this includes not only using package managers but also learning to deploy code using them, which includes syntaxes for package specification formats, dependency and versioning rules and deployment conventions. Simply ignoring these environments and managing modules and dependencies by hand is tempting, but the complexity of heterogenous environments and keeping track of dependency updates can become burdensome — all these package managers were created to solve pratical problems which the developer would have to otherwise directly handle, after all. Another alternative that is often proposed, especially by users of operating systems that feature a system-provided package manager (as is the case of most Linux distributions), is to avoid using multiple package managers and use a single general-purpose package manager. This is, of course, as much as a solution as trying to make everyone agree on a single programming language, and this is the first of various analogies between package management and programming languages that we will make throughout this paper. The result is that the ecosystem is not getting any simpler, and at first glance it seems that package management is indeed a largely unsolved problem.

However, maybe the statement "package management is an unsolved problem" simply does not make sense, and is akin to saying that "programming languages are an unsolved problem". In the programming languages world we accept that the multitude of languages is a given. Beyond that, we understand that there are families of languages with different paradigms, with well-known tradeoffs. We also accept that there is room for domain-specific languages and for general-purpose languages. Most importantly, we know how to set boundaries for each language and how to make DSLs and general-purpose languages interact.

We argue that all of these observations can be made with regard to package management as well. In this paper, we will discuss how these observations map from the world of programming languages to that of package managers. Most existing package management systems, however, are still oblivious to the fact that they exist as part of a larger ecosystem, with parts of it handled by other package managers. By discussing how programming languages deal with these issues, we point to directions on how package manegers could follow their example, drawing on our experiences developing both a system-wide package manager [10, 9] and a language-specific package manager [11].

Paradigms of package management

An analogy between programming language paradigms and package management paradigms

We went through filesystem vs. database in Gobo, but this appears everywhere.

- filesystem paradigm vs. database paradigm
- trade-offs: how runtime lookup of files happens
  - databases involved in runtime lookup: gtk icon cache, Haskell db updates, etc.
  - pkg-config, how does it fit?
  - filesystem paradigm and runtime lookup: environment variables, index directories with symlinks, symlinks in lib (libfoo.so, libfoo.so.1, etc.)
- versioning in db vs filesystem. libfoo.so vs. libfoo.so.1, versioning in gobo, /usr/include/python2.7/ etc.
- changes in filesystems to better support the filesystem paradigm? (union mounts, etc)
- related work. outside of "classic" Unix: appdirs, Windows, Plan 9.
- filesystem paradigm in "classic" Unix: /opt
  - parts of the FS hierarchy which use the filesystem paradigm: /usr/share/appname
  - parts which don't: /usr/share/icons
    - effect this had on share/ in gobo
- Makefiles with install and uninstall rules in ports: early example of database paradigm?
- other instances of filesystem vs database paradigm:
  - sysv and BSD init versus systemd
  - Mac OSX equivalent
  - /etc text-based config files vs. gconf
  - Windows registry.

Language-oriented vs. language-agnostic package managers

An analogy between PL space with DSLs and general purpose languages and pkgmgr space with language-oriented and language-agnostic managers

- My paper on LuaRocks makes a case for language-oriented package managers:
  - scalability (30,000 Ubuntu packages in total, 50,000+ Ruby gems alone)
  - portability (language managers are used in several OSes)

Integration between languages vs. integration between package managers

An analogy between PL interaction (APIs and FFIs) and pkgmgr interaction (virtually none??)

In GoboLinux we researched the idea of building an "FFI" of sorts into our general-purpose package manager, called Aliens [cite Michael's presentation in linux.conf.au]. The idea was that we could provide a general API for writing shims that interacted with domain-specific package managers (that is, language-specific ones) in an clean way. Ad-hoc integrations between general-purpose and domain-specific package managers proved traumatic to some projects [debian's troubles with rubygems]. Still, they have been successfully realized in others, such as the use of LuaRocks by Buildroot.

language-oriented (domain-specific) vs. language-agnostic (general-purpose) package managers

Practical experience with filesystem-oriented package management: Gobo

- earliest related work: GNU stow, encap. (Our paper from Workshop em Software Livre mentions that, but we didn't really know about them when making Gobo)
- GoboLinux
  - Our extensive use of shell scripting for system management was a sort of attempt to show that the system could work with "just a few scripts".
  - core ideas work (the system is still driven via symlinks) but management scripts became complex over time
- the build system, the tooling for generating packages, is the central piece of a package management system.
  - not clear at first, but looking at package management systems they all integrate the build process. Why?
- Discuss the challenges of GoboLinux through the prism of its build system (?)
  - Tried to make it easy for users to build packages.
  - Took inspiration from systems such as Gentoo, which was itself inspired from BSD Ports.
  - aim was to make the simple cases super-simple (like a 3-line script) and the complex cases possible (leveraging the generality of shell scripts)
  - This worked up to a point. Eventually, started requiring more and more metadata, even for the so-called simple cases.
  - Further, this metadata had to be integrated with the deployed system.
  - The ArchLinux build system seems to be a modern-day successor of this style of build system in the Linux space.
- Modern examples
  - The Nix project has been around almost as long as GoboLinux. NixOS is nowadays a serious contender in the world of server-oriented operating systems.
  - virtual machines: minimalism is making its way back in OS layout design. (There have always been minimalistic Linux distributions, back from the "rescue" distros such as Damn Small Linux and tomsrtbt (which would fit in a floppy!).

        * CoreOS, based on Gentoo, is the current representative of the minimalistic server-oriented distro world.
        * Ubuntu Core was recently announced, and this might be a beginning of a general trend of "core" OSes.
            · Ubuntu Core "snappy" packages strongly resemble GoboLinux!
    - Homebrew, a package manager for Mac OS X, is a successful realization of this idea. One of its original design criteria was to do package management "the GoboLinux way" [in the git history of homebrew's README.md we find them citing gobolinux] (so I guess that's the most widespread legacy of our work)

- What do we mean by filesystem facilities: a more powerful fs would make things better?
    - Back in 200x I wrote [the "clueless" whitepaper] how we needed more low-level tooling from the underlying operating system in order to be able to realize some of the ideas of GoboLinux cleanly. We were asking for more abstraction and isolation in userspace: essentially I wanted union filesystems and possibly some sort of containers. Lacking those, we had to make do with chroot.
    - For a while we hoped that as underlying technology matured, these ideas could come to fruition.
    - Docker seems to be proof of that; a container-based system that greatly simplified application deployment.
    - However, Glauber Costa, one of the developers of the Linux Containers system, described the limitations of that approach exposed the hackery involved [did he write about it somewhere?]. Costa himself moved away from containers and joined the efforts of the OSv project, a minimalistic operating system targeting hypervisor-based architectures.

References

[1] Bower - a package manager for the web. `http://bower.io`, 2015.

[2] Homebrew - the missing package manager for OSX. `http://brew.sh`, 2015.

[3] The MacPorts project. `http://www.macports.org`, 2015.

[4] node.js. `https://nodejs.org/`, 2015.

[5] npm - Node Package Manager. `http://npmjs.com`, 2015.

[6] John Arrundel. *Puppet 3 Cookbook*. Packt Publishing, August 2013.

[7] Anthony Burns and Tom Copeland. *Deploying Rails*. Pragmatic Bookshelf, July 2012.

[8] Mitchell Hashimoto. *Vagrant: Up and Running*. O'Reilly Media, 1 edition, June 2013.

[9] Michael Homer, Hisham Muhammad, and Jonas Karlsson. An updated directory structure for Unix. In *linux.conf.au 2010*, Wellington, New Zealand, 2010.

[10] Hisham Muhammad and André Detsch. An alternative for the Unix directory structure. In *III Workshop Software Livre*, Porto Alegre, Brazil, 2002.

[11] Hisham Muhammad, Fábio Mascarenhas, and Roberto Ierusalimschy. LuaRocks - a declarative and extensible package management system for Lua. *Lecture Notes in Computer Science*, 8129:16–30, 2013.