Package Universes: Which Components Are Real Candidates?

Alexander Spoon (lex@lexspoon.org)
Swiss Federal Institute of Technology, Lausanne (EPFL)

July 2006

Abstract

Package universes is a component-distribution architecture based on explicitly managing the set of components visible at assembly time. The architecture is usable as is for a number of common organizations and software-engineering processes. Also, by providing a context of use for components, it allows several simplifications in the underlying component system. Experience is reported about two prototype implementations and user groups.

1 Overview

To achieve a reliable assembly of components, one must select good components and reject poor ones. Component systems alone are not enough: as an extreme example, randomized components are typically useless but can be built to match any interface specification. Some mechanism of component distribution is necessary to reject poor components and focus attention on better candidates for assembly.

Such distribution systems are useful in addition to necessary. They lower the requirements and help evaluate the underlying component systems. The lowering of requirements occurs because there is now an alternative mechanism for coping with faulty components: one can optionally use the distribution system instead of the component system to remove the fault. While demanding applications doubtless require a more refined component system, the *package* in package universes emphasizes that the mechanism has applications even when the "component" system is too unrestrictive to deserve the name.

The clarification of a package-distribution mechanism comes from having a context of use of the component system. Without some such a context, it is impossible to weigh the number of errors avoided by the component system against the additional work needed to build the well-formed components. A context of use allows one to convert numbers of errors into equivalent amounts of human work and thus compare quantities in these two different units. Indeed, some component systems, such as those based on type checking [7, 9], add new assembly-time errors with the rationale that worse errors are removed than added. Evaluating such an argument requires comparing not only errors to work, but also comparing the work associated with different kinds of errors. A context of use enables such comparisons.

Package universes is a general architecture for package-distribution systems. It is designed to allow loosely coupled developers to develop a set of mutually compatible components. Thus, it is collaborative software [13, 2] to support a global software process based on components [15].

The general viewpoint of package universes is that developers work within some large but limited *universe* of components. To achieve reliable assembly, one must arrange the software-engineering processes so that the set of components in a universe evolves in desirable directions. Within the general architecture, universes based on explicit lists of components appear practical for a number of development communities. Such universes rely heavily on index servers which hold a list of components. The indexes are then developed, with packages being added, updated, and removed, according to software engineering processes appropriate to the particular development community.

This paper defines package universes in general as well as the specific class of universes based on explicit index servers. It describes potential usage scenarios and plausible simplifications supported by the architecture. It then evaluates those potentials in light of experience with a major Linux distribution and with two implementations of package universes.

2 Package universes architecture

The package universes approach is to focus on the set of components visible to an assembly tool. The set of packages is the visible *universe* of candidate components. Assembly tools always operate within some package universe.

A *package* or *component* is any portion of an assembled program or computer system, for example a RedHat Package Manager (RPM) file [6] or a Java Archive (JAR) file. This paper often writes *package* to emphasize that the distributed components may have very weak declared dependencies. An *assembly* is a combination of components to form a whole. An *assembly tool* manages locating and combining components into an assembly.

Packages can *depend* on other packages. A package assembly tool should only assemble a set of components if all of their dependencies are met by each other. The assembly tool can aid dependency resolution by suggesting components to add to the assembly, but it must only select components available within the developer's current package universe.

A package universe can be defined very broadly. For example, "every package accessible through Google" is a package universe. However, such large universes provide little help in forming reliable assemblies. A useful package universe must have some form of control on what it includes.

One particularly interesting class of universes are those designed around explicit lists of available packages. An *index-server* universe is one whose packages are those stored explicitly on an *index server*. The contents of the index server change over time as authorized developers add, update, and remove packages. An index server corresponds closely to a repository in many Linux distributions.

Universes can be profitably combined in several ways. A *name-based union* combines the packages from an ordered list of universes. Packages in later universes in the list override all same-named packages from earlier in the list, including packages with the same name but a different version. A *pure union* does not override same-named, different-version packages, but such pure unions appear less useful in practice than name-based unions.

A *filter* universe includes a subset of the packages in another universe. The subset is defined by a regular expression which is matched against the package names. Such universes are useful for selecting a few packages—possibly just one package—from an index server.

The *empty* universe has no packages. It is a special case of a union of zero universes. The empty universe is a useful boundary case in theory and in implementation of package universes.

A *literal* universe includes an explicit list of packages in the universe description. Literal universes are practical in two scenarios: test suites for package-universe implementations, and small universes for personal use.

Finally, an *indirect* universe holds a URL that references another universe's description. Indirect universes are useful for communities whose package universes have a complex internal structure. Instead of community members installing complex universe descriptions on their machines, they can refer to a description located on a central community web site.

3 When a package is not available

Regardless of how large a package universe is, developers working within it will eventually want a package that is not available in that universe. Better universes will more frequently have what a particular developer needs, but no practical universe can have everything.

There are two choices whenever an unavailable package is needed. First, the developer can find the package elsewhere, repackage it to be compatible with the present universe, and then add that to the present universe. If it is added by means of an index server, then any other developer using packages from the same server can also use the new package. In Linux-distribution communities, repackaging is the role of a package maintainer. For a private organization, this is part of the role of a program librarian [12].

Alternatively, the developer can locate an entire universe, possibly a tiny one, that includes the package and is maintained by someone else. If there is reason to believe that this universe is compatible with the developer's present universe, the developer can expand his local universe to include the alternate one. In Linux-distribution communities, there are typically a number of such satellite universes. The satellites are maintained independently, but their maintainers commit to maintaining compatibility with some larger communal universe.

4 Sample organizations

Package universes is a general architecture that supports a variety of community organizations and software-engineering processes. This section describes how to instantiate package universes in several useful ways. It assumes, without going into detail, that access control is available on the index servers to restrict what modifications are allowed.

No restrictions. Some communities, especially new ones, may elect to have no restrictions on the shared content. Wikis [11] have shown that such policies are effective for some communities. This policy is trivial to implement: have one index server and disable all access control.

Full access, but only to community members. Many open-source projects have an organization where community status gives broad permission to modify the shared repository. Debian's "Debian Developers" and FreeBSD's "committers" are examples. This policy can be implemented by letting the general public read from the community server, but only allowing updates by community members with sufficient status.

Private in-house development. Individual groups can form their own universe for private development without needing to coordinate with any central organization. They simply create an index server accessible only to themselves.

Public libraries plus private development. The above organization can be refined by allowing developers to use publicly available packages even as they develop packages for private use. Developers in the organization post packages to the local index server, and they then operate within a name-based union of the public universe along with the local index-server universe.

Stable versus unstable streams. Many projects distinguish between stable and unstable streams of development. The stable streams include packages that are heavily tested and deemed to be reliable, while the unstable streams include packages that are more current have more features but are not as reliable. A project can implement this policy by having separate universes for each development stream.

Freezing new stable distributions. A common process for generating stable distributions of code is to take an unstable stream, start testing it, and disallow any patches except for bug fixes. After some point, the distribution is *frozen* and considered a stable release. Such processes can be supported straightforwardly with index-server package universes. For the testing phase, change the development process so that only bug fixes are uploaded. Some teams may want to employ moderation for uploads. The freeze itself is implemented by revoking all update privileges to the index server. Finally, the frozen universe can be duplicated, with one fork hosting the new development stream while the other becomes designated as a stable release.

In general, the above examples show that package universes leaves room for many update policies. Centralized control is not required, but it may be applied to the extent it is important to a particular community. Both extremes of control are supported: centrally planned communities can employ heavy access controls to the index servers, while ad-hoc groups can use virtually no access control.

Particularly interesting is that local communities can build new package universes based on existing carefully managed universes, thus permitting new communities and package universes to be organized in a bottom-up fashion. That is, it is possible to treat existing universes as components of new universes.

5 Plausibly not needed

Communities using package universes to distribute content can plausibly use simpler components without sacrificing the ability to achieve robust assemblies of components. While the ultimate proof is in the experience, this section proposes a few simplifications that are plausible.

Package names can be short. Package names do not need to include DNS domain names or otherwise attempt to be globally unique. Names can be short because they are all used within the context of one universe, and each universe has a presumably cooperative set of users (communities whose users attack each other have larger problems than their infrastructure!).

Versions can be totally ordered. Branching versions are not required, as contrasted with systems like Conary [4]. Developers contributing packages to a universe are assumed to cooperate and to want the same update policy for their packages. Thus, all users of the universe can update their packages in tandem, instead of some users operating on different branches of development from others. Instead of branching via versions, branching is accomplished at a larger granularity via the use of multiple universes.

Dependencies can be based simply on names. For example, "A depends on B" is sufficient. Versioned dependencies, such as "A 2.5 depends on B newer than version 1.4," are not needed, because users tend to update to the newest version of all packages anyway. Rich interface dependencies, *e.g.* those based on semantic behavior [21], are not strictly necessary, because in-universe incompatibilities between components can be treated and resolved in the same way as bugs within individual components. The simple dependencies are plausible because all dependencies are resolved within a single universe of packages that have been developed to be mutually compatible. The limited scope increases the power of those working within it.

It is true that this simple versioning approach does not directly support package updates that intentionally break compatibility. Instead, users must use package names to represent incompatible components. For example, if "libc version 5.100" and "libc version 6.101" break compatibility, then they could be considered separate packages by naming them "libc5 version 100" and "libc6 version 101."

In general, any dependency problems that arise can be worked around by modifying the contents of the package universe. That is, instead of requiring the dependency system to handle an arbitrary set of packages, the package universes approach is to let humans engineer the set of packages so that they have simple dependencies. Instead of solving harder problems, community members can work together to make the dependency problems easier.

6 Debian's experience

Debian [14, 5] uses a package-universes architecture to develop a Linux distribution. It employs roughly 1,000 volunteer developers and distributes over 10,000 packages of content. Thus, the project has significant experience with distributing packages using index-server package universes. This section briefly evaluates package universes in light of this experience.

Debian limits access to the index servers to "Debian developers." It uses separate index servers to separate its package streams into a number of "stable" releases and a single "unstable" stream undergoing active development. The project freezes new stable releases using the process described just above.

Most but not all of plausible simplifications described above are supported by Debian. Despite having over 10,000 packages, the package names in Debian are short, and in particular do not include any systematic disambiguation scheme such as DNS domain names. Similarly, the version numbers are totally ordered with no internal branching. Instead, it relies on occasional package renaming, *e.g.* `libc5` versus `libc6`.

Debian's dependencies, on the other hand, are more complex than proposed in the previous section. The dependencies include: depending on alternatives, multiple provided package names per package, depending on specific version ranges, and three gradations of dependency strengths, namely "depends," "suggests," and "recommends." While this dependency system is simpler than

those of many component systems, they are significantly more complicated than the simple dependencies described in the previous section. This suggests, but does not establish, that in practice a richer dependency system may be necessary than this paper conjectures.

Finally, Debian has developed both large communal index servers and small satellite universes. The communal index servers, as mentioned above, include over 10,000 packages. At the same time, `apt-get.org` [1] lists thousands of satellite universes that can be combined with one or another main Debian universe.

7 Implementation experience

Two principled implementations of package universes have been tried.

- **Package Universes in Squeak.** Package universes was first implemented for Squeak [8, 20]. Many components formats were supported, ranging from completely unstructured change sets and binary project dumps, up to highly structured Monticello packages [3]. Two major universes were tried: a development universe with little access control and many packages, and the Stable 3.7 universe with stricter access control and a more limited set of relatively robust packages. The development universe includes 500 packages, while the Stable 3.7 release includes 200 packages. The implementation requires 2,133 lines of non-blank, non-comment Smalltalk code, ignoring class definitions.[1]

- **Scala Bazaars.** Scala Bazaars [19] is a recent package-universes implementation used to support the open-source community for Scala [16, 18]. The distributed packages are simply zip files. The implementation is 5,026 lines of Scala code.[2] The one widely used index server includes 20 packages.

Experience to date is far lower than with Debian, but it is enough to merit two comments. First, the experience suggests that simple, unprincipled package systems are remarkably effective. Both implementations use a preponderance of simply formatted packages while still achieving a large number of components that can be mutually installed. While assembly-time errors occur, errors that a component system is ideally suited to prevent, they are less frequent than errors within packages.

Second, simple naming and dependencies appear to be quite practical. As one example, there are four published versions of the Refactoring Browser [17] on SqueakMap [10], an index of all software publicly available for Squeak. These different versions correspond to different load dependencies, *e.g.* "Refactoring Browser for 3.2" and "Refactoring Browser for 3.7". To contrast, the two package-universes index servers simply have "Refactoring Browser."

8 Conclusion

Package universes provide a conceptual framework for discussing the distribution and selection of components. Experience suggests that, more specifically, a focus on index-server package universes works well for open source communities. This experience has implications for the design of future component systems, because the positive results were achieved without needing sophisticated component formats.

[1] Calculated with the built-in `linesOfCode` method.

[2] Calculated with the `wc` Unix command.

9 Acknowledgments

SqueakMap convinced me that how you distribute packages is as important as the form of those packages. Many members of the Squeak and Scala mailing lists helped flesh out the ideas of package universes, including: Hannes Hirzel, Göran Krampe, Craig Latta, Sean McDirmid, Stephan Rudlof, Michel Schinz, Jamie Webb, and many others.

References

[1] apt-get.org web site. `http://www.apt-get.org`.

[2] Liam J. Bannon and Kjeld Schmidt. CSCW: Four characters in search of a context. In *Studies in computer supported cooperative work: Theory, practice and design*, pages 3–16. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 1991.

[3] Avi Bryant and Colin Putney. Monticello web site. `http://www.wiresong.ca/Monticello/`.

[4] Conary web site. `http://wiki.conary.com`.

[5] Debian web site. `http://www.debian.org`.

[6] Eric Foster-Johnson. *Red Hat RPM Guide*. Wiley Publishing, Inc., Indiannapolis, IN, March 2003.

[7] Günter Graw, Peter Herrmann, and Heiko Krumm. Composing object-oriented specifications and verifications with cTLA. In *Proc. of the Workshop on Component-Oriented Programming (WCOP)*, 1999.

[8] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *ACM Conference on Object-Oriented Programming, Systems, Language, and Applications (OOPSLA)*, 1997.

[9] Paola Inverardi, Alexander L. Wolf, and Daniel Yankelevich. Static checking of system behaviors using derived component assumptions. *ACM Transactions on Software Engineering Methodology*, 9(3):239–272, 2000.

[10] Göran Krampe. SqueakMap web site. `http://map.squeak.org`.

[11] Bo Leuf and Ward Cunningham. *The Wiki Way: Quick Collaboration on the Web*. Addison-Wesley, April 2001.

[12] Harlin Mills. *Chief programmer teams: Principles and procedures*. IBM Federal Systems Division, Gaithersburg, US, 1971.

[13] Kevin Mills. Computer-supported cooperative work. In Patrick R. Penland and Aleyamma Mathai, editors, *Encyclopedia of Library and Information Sciences (2nd Edition)*, pages 666–677. Marcel Dekker, New York, May 2003.

[14] Mattia Monga. From bazaar to kibbutz: How freedom deals with coherence in the Debian project. In *Fourth Workshop on Open Source Software Engineering*, Edinburgh, UK, 2004.

[15] Tobias Murer. The challenge of the global software process. In *Proc. of the Workshop on Component-Oriented Programming (WCOP)*, 1997.

[16] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL, 2004.

[17] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[18] Scala web site. `http://scala.epfl.ch`.

[19] Scala Bazaars web site. `http://www.lexspoon.org/sbaz`.

[20] Squeak web site. `http://www.squeak.org`.

[21] Sotirios Terzis and Paddy Nixon. Component trading: The basis for a component-oriented development framework. In *Proc. of the Workshop on Component-Oriented Programming (WCOP)*, 1999.