Modular System Building with Java™ Packages

Mick Jordan

`Mick.Jordan@Eng.Sun.COM`

Michael L. Van De Vanter

`Michael.VanDeVanter@Eng.Sun.COM`

Sun Microsystems Laboratories
2550 Garcia Avenue
Mountain View, CA 94043 USA

Abstract

As part of a project to develop scalable development techniques for systems written in the Java programming language, we are investigating the suitability of the package construct in Java as a system structuring mechanism. Although the Java package is incomplete in this regard, it represents a good foundation when combined with an advanced programming environment inspired by Vesta. The few ways in which the Java package is unsuitable appear to be correctable with careful programming conventions and support from the environment.

At the center of the proposed approach is the notion of a hierarchical namespace based on Internet Domain Names populated by reusable, independently versioned packages, each of which encapsulates a parameterized build script. This concept unifies several important aspects of software development and permits the design of tools that simplify the development process. A laboratory prototype environment, based on persistent Java objects, is being constructed and now supports its own development.

1. Introduction

The Java programming language [7], widely thought to be designed for portable mini-applications (browser applets), is in fact broadly applicable in traditional domains. The development environment, initially populated with a small set of class libraries (the Java Development Kit or JDK™), is rapidly accumulating third-party libraries for a variety of application areas. Little attention has yet been paid to how programming in Java will scale to large systems, but this will change with growing pressure for sophisticated applets and traditional applications written in Java.

An experimental programming environment for Java being developed at Sun Microsystems Laboratories addresses problems of scale, some of which are familiar and some of which become especially difficult in the decentralized approach to development that is emerging in the Java community. Compatibility is an explicit goal: we wish to avoid language changes and to conform to current Java programming practice. A key part of our strategy is to support the language with an aggregation mechanism that enables system construction by modular composition. For this to succeed, problems related to naming, storage, configuration management, and location-independent building must be solved.

Rather than introduce an entirely new mechanism, we augment an existing one. The Java package mechanism represents a good start toward modular system building, but it is too weak to scale suitably. We supply the missing properties with an advanced software development environment (described more completely elsewhere [9]) that strengthens the package concept and helps developers cope with a few characteristics of packages that are problematic. We also mention small language changes that would have great benefit in this area.

The key idea (an adaptation and extension of the approach taken in Vesta [10]) is the notion of a uniquely named, reusable independently *versioned package*, each version of which contains a *build script*: a parameterized program for building it. This approach uses packages to unify important aspects of large system development: system structure, storage management, building, and configuration management. Tools exploit this unity to simplify the developer's task, even while providing stronger guarantees of reliable and repeatable system builds than is now common.

A prototype is under construction and has been supporting its own development for some time. It is written using an experimental implementation of persistent Java objects [3].

Section 2 of this paper discusses challenges facing

1

developers of large Java systems, pointing out how the Java package mechanism helps with a few of them. Section 3 describes how the Vesta approach offers promise for some others. Section 4 presents our proposed approach to scalable development in Java, and Section 5 summarizes the implementation strategy for our prototype. Section 6 discusses related work, and Section 7 concludes with a discussion of our experience to date.

2. Challenges for large scale Java development

Developers of large Java systems face many technical obstacles, both old and new. A common theme is the need to reliably assemble, build, and deploy systems that are composed out of parts, with the process crossing organizational and geographical boundaries. The tools we have now are unreliable, non-portable, excessively complex, and ultimately don't permit development at this scale to be done effectively.

2.1. System structure and naming

Large systems must be assembled from modules or subsystems that are to some degree independently developed. At the center of all mechanisms for doing this is the matter of naming, and the disarray of our current development tools in this regard accounts for a great deal of the complexity a developer faces.

Even reduced to its simplest terms, software development involves four naming disciplines that are usually independent because of the separate evolution of tools:

1. A programming language supports naming of variables, classes, methods, etc.;
2. A storage system provides naming of files and directories;
3. A build system names the subsystems that are to be assembled; and
4. A versioning system affects how many of these names get bound.

Developers depend on ad hoc relationships among these naming schemes, for example aligning subsystems with file system directories. This has two costs as systems scale up: manual maintenance of many such relationships, and a lack of abstraction and portability.

Naming systems must permit composition of independently developed subsystems without fear of conflicting language names, for example class and procedure names. A similar problem confronts subsystem naming, especially where Java developers expect to combine independently developed libraries and packages.

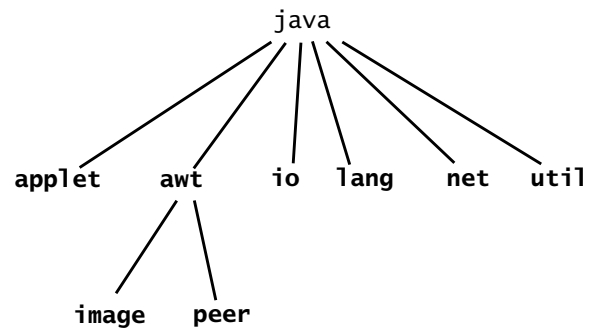Java makes progress on these two points with the *package* language construct. Figure 1 shows the hierarchical

Figure 1. Namespace for core Java packages

namespace for the core Java packages; in this diagram names appear in boldface at nodes where classes (leaf nodes that are not shown) are present. A fully qualified class name, for example `java.util.Stack`, consists of a class name (`Stack`) prefixed by its dot-separated hierarchical package name (`java.util`).

Java packages play several useful roles. Packages provide language-level *access control* to class methods and fields. As a *convention for naming storage objects*, packages help bring some alignment between language and storage names (more about this in Section 2.2).

As a *name control mechanism*, packages help ensure unique identifier names in local contexts because of the tight mapping between storage names and package names: you can't have multiple directories with the same name in a file system. Unfortunately, this doesn't help when more than one storage system is involved. Using the package namespace effectively requires widely adopted naming conventions (some approximation of a "one world" naming scheme) that make available unique prefixes within the hierarchy of names. The best current approach, suggested by Java designers [7] though not widely followed, is to begin global package names with the Internet Domain Name of the development organization.

As a *system structuring mechanism*, packages support the Java `import` statement, making visible the public classes from specified other packages. Unfortunately packages are not first-class language entities; this means that packages are not fully represented at run time and cannot be named or otherwise manipulated (in contrast with classes and variables, for example). As a consequence, the Java package suffers from two weaknesses as a system structuring mechanism. First, the relationship between a package name and a true subsystem is not enforced. Unlike Modula-3 modules [12] and Ada 95 packages [1], Java packages play no part in the dynamic semantics of the language. Initialization order, in particular, is defined in terms of classes not packages. A Java program may thus contain

any collection of classes, independent of package namespace structure. In fact, because a Java virtual machine loads classes dynamically on demand, and because the name of a loaded class might be computed at run-time, the exact set of classes that a program may load cannot be determined before it runs. In practice, convention mandates a relationship between package names and subsystems: programs commonly contain complete subtrees of the namespace, for example all classes in `java.lang`.

A second weakness is that Java packages cannot be imported: a Java `import` may specify only a single class or all classes from a package. Thus the code in Figure 2,

```
import COM.sun.labs.jvb as jvb;

class SpecialStack extends jvb.util.Stack
{ ... }
```

Figure 2. Illegal import/rename of a Java package

which attempts to use the `COM.sun.labs.jvb` package as a language object that permits shortened names, is illegal for two reasons: packages can not be imported, and there is no renaming. This limitation makes the use of long Internet Domain-based package names awkward.

2.2. Storage

A second issue in the development of large systems is management of persistent storage, typically in an independent namespace, for both *source* (human created) and *derived* (machine generated) information. Common practice is to construct a hierarchical file namespace and to maintain manually some alignment between this namespace and the others, for example subsystems with directories.

Current Java tools strongly encourage certain alignments between the storage and language namespaces, for example using one public class per *compilation unit* (source file), with the file name matching the tail of the class name. Java tools work best when compilation units are stored in a directory whose name is strongly related to the package name. For example, the compilation unit for class `Stack` in package `COM.sun.labs.jvb.util` would be stored in the UNIX™ file named `someprefix/ COM/sun/labs/jvb/util/Stack.java`.

This alignment helps, but two distinct namespaces must still be maintained manually. Furthermore, the Java compiler creates considerable confusion by exposing storage names for derived information (class files) and requiring manual management of search paths that include both source and derived storage locations.

2.3. Versions and configurations

Essential to development of large systems is management of information that developers create: this includes permitting concurrent, non-interfering development by multiple individuals and groups, recording history, building variants for different tasks and platforms, and knowing what to deploy as finished products. Terminology is confusing and many tools in everyday use provide little or no support for the kind of configuration management that is needed.

Tools such as SCCS support only the simplistic notion that a logically single object may exist as a family of related *versions*. This makes it generally impossible to reconstruct earlier builds, since the granularity of what is versioned (individual source files) does not match what the developer actually builds (collections of source files). ClearCase [4] addresses this by versioning directories in addition to files, but of course directories are a storage system concept, not a building concept.

A *configuration* is some collection of developed objects intended to have utility outside its immediate context, and *configuration management* is the process of specifying which versions of which objects to include in a such a collection. Even when supported by tools, however, configurations are typically named and managed as independent, often proprietary, non-linguistic entities. One symptom is a discontinuity at project boundaries. For example, developers often use configuration management and version control systems inside projects, but use ad hoc mechanisms outside projects. The JDK is managed internally by Teamware [15] and SCCS, but external clients see trees of unversioned files in which the identity (and meaning) of configurations is lost, along with sharing among architectural variants and any realistic potential for shared, incremental evolution.

2.4. Building

The standard tool for building in Java and most conventional environments is currently make [6]. Some weaknesses of make-based system have already been mentioned, for example inability to name the subsystems out of which large systems are built. Other problems include the invisibility of many actual dependencies, failure to record enough information about the derivation of objects in order to reuse them safely, and extremely poor support for the composition of makefiles into large systems.

2.5. Reuse of system components

As systems scale up, especially in the emerging Java development model, code reuse is crucial to success. We

define *reuse* here as the ability for a *subscriber* to include into a Java application (either applet or stand-alone program) one or more collections of Java classes (which we'll call *components* for the moment) developed independently by *publishers,* across organizations and via the Internet.

Significant issues, such as techniques for designing reusable code and motivations for doing do, are beyond the scope of this work. However, we note in passing that, unless appropriate use is made of abstract classes and interfaces, Java components quite easily become highly interconnected, difficult to evolve and very difficult to build with traditional tools. This is much more problematic in Java than in languages that enforce abstraction (e.g. Ada [1] and Modula-3 [12]), or in those that provide at least ad hoc separation of declaration and definition (e.g. C or C++ [5]).

Many technical obstacles to reuse, however, can be addressed. The first, discussed in Section 2.1, is the need to combine separately developed components without name clashes. Other issues include how access is managed, and by what abstraction subscribers can use components.

Of the possible approaches to implementing publish and subscribe, *copying* and *sharing*, only sharing can scale up effectively. Copying obviously costs system resources and programmer effort, but the crucial disadvantage is that a subscriber effectively *assumes ownership* of copied code. Copying compounds, for example when one subscriber makes a copy and publishes code that depends on the copy, and a second subscriber makes a copy of that and publishes more code that depends on the second copy, and so on. Unless it becomes possible to revert to a sharing model at the time of publishing, scaling problems will defeat extended reuse.

For sharing to work, however, the advantages of copying must be supplied by other mechanisms. For example, there must be version management and access control to ensure orderly progress. There must be transparent local caching when performance is an issue. Finally, versions and configurations of shared code must not change, that is, they must be *stable*.

It is important that the abstraction by which subscribers use components hides unimportant details but retains necessary degrees of freedom. Neither of the standard approaches (binary and source reuse) does so.

Binary components can be reused only when compatible with the subscriber's environment. Binary compatibility problems plague environments for languages such as C and C++, typically implemented with relatively tight bindings. One symptom is the "fragile superclass problem" in C++.

The situation for Java class files isn't much better. A common misperception is that Java's more dynamic binding (symbolic binding of the class file format and dynamic class file loading) inherently solves compatibility prob-

lems. In fact Java's binary compatibility rules, defined by the Java virtual machine [11], permit only relatively minor variations. Further, the binary compatibility rules actually permit combinations of classes that would fail to compile together from sources. Finally, there are no guarantees at all concerning the behavior of modified code, since compatibility is defined only in terms of class signatures.

Reusing source components preserves useful degrees of freedom, for example permitting the subscriber to build multiple variants with respect to compilation options such as debugging flags, or even choice of compiler. The traditional cost is the overhead of integrating sources into the subscriber's environment and the management of the software in multiple forms.

A more appropriate abstraction mechanism would support reuse in more relevant terms, for example permitting a subscriber to specify "version 5 of package `COM.sun.labs.jvb.util` for platform SPARC™/ Solaris-2™ with debugging code included."

3. Vesta

The Vesta configuration management system was developed at the Digital Equipment Corporation's Systems Research Center (SRC) to address the needs of organizations developing and releasing large, complex software systems [10]. Weaknesses in current UNIX tools led to the basic requirements for Vesta:

- The amount of developer effort needed to propagate a change are commensurate with the conceptual size of the change, not the amount of affected code.
- System-building descriptions are concise and scale with the size and complexity of the code under development, not with the size of the entire system.
- System-building descriptions are complete and self-contained, so that any built object can be reconstructed reliably when needed.

Vesta uses an integrated storage system (repository) and system builder in which the unit of versioning is the same as the unit of building, addressing one of the problems mentioned in Section 2.3. Many of the problems mentioned in Section 2.4 are addressed by the Vesta build system, which is based on composable, parameterized evaluation of functional programs (instead of pattern-based rule invocation) and on generation of detailed information about the result of each build.

The approach was validated by using Vesta to manage "about 4400 modules comprising 1.4 million lines of source code, mostly written in Modula-2+ [14], but with some C and assembly language. The code included many different kinds of subsystems, such as a micro-kernel operating system, a comprehensive set of user libraries, soft-

ware development tools, and a substantial collection of application programs. New versions of these components were being produced frequently by the 25 full-time developers, and changes in the interfaces between major components occurred often, necessitating frequent integration. The development tools (e.g., compilers) were evolving concurrently with the software that was built with them. Nearly every component was targeted to run on multiple operating systems and machine architectures." The authors concluded that their goals had been met [10].

4. A scalable approach to Java development

We propose an approach to developing large systems in Java that addresses problems of scale discussed in Section 2. We start with the Vesta approach, which unifies versioning, configuration management, and building at the granularity of what Vesta calls "packages." We apply this notion to the Java language by identifying packages in the development environment (the reusable "components" of Section 2.5) with packages in the Java namespace, working out a number of name-related issues that this raises. Finally, we design an integrated environment that fills in the missing pieces and exploits the simplicity made possible by this approach.

Central to the proposed development model are independently *versioned Java packages*: collections of classes (as well as other resources) that exist in stable (immutable), versioned configurations. Each version contains a build script: a parameterized program for building the package. The environment abstracts away details of package location, contents, and construction, affording developers the luxury of a single package naming scheme in most situations. The following overview describes the roles played by packages in this approach.

- *System Structure*. The developer assembles large applications by constructing packages whose build scripts import other packages. Each import specifies a particular version of the imported package, and the builder's import mechanism abstracts away conventional distinctions between source (uncompiled) and binary (compiled) reuse.
- *Storage*. The developer manages sources in terms of the package namespace, with package-level versioning added, using tools that abstract away details of underlying storage and distribution. The build system invisibly manages derived objects (class files and other objects created by invoking build scripts), thereby reducing name clutter and eliminating confusion among objects derived in alternate variants.
- *System Building*. The developer builds a package by invoking its associated build script (called its *system*

model in Vesta), which incorporates imported packages by recursive invocation of their scripts. The developer gains access to derived objects (for example an executable) by invoking tools that abstract away the complexity of the script's result. The developer can supply parameters that cause variants to be built (even of imported packages, possibly located in distant, read-only locations) without perturbing package sources. Build script results contain extensive information that is both precise and complete, enabling straightforward integration of tools that extract, analyze, and display particular kinds of information to developers.

- *Configuration Management*. The developer creates configurations as packages whose role is to import particular versions of other packages, including other configurations. Each version of such a package specifies transitively and immutably an arbitrarily large, buildable aggregation of packages comprising some version of an application, applet, library of classes or other deliverable. A configuration manager permits concurrent non-interfering work by multiple developers.

We intend that this approach be simple to use and implement, as exemplified by our prototype environment, described in more detail elsewhere [9]. The remainder of this paper will focus on the use of Java packages in the proposed approach.

4.1. Packages and system structure

We use Java packages as the building blocks for large systems. However, as mentioned in Section 2.1, there is no enforced notion of subsystem (or module) behind the Java package namespace. We add this notion in the development environment, following Vesta, by making the package the unit of storage, the unit of versioning and configurations, and the unit of building. We thus promote the package as the basic unit of code sharing and evolution (more about this in Section 4.3). We ensure that package versions can be named independently of location, thus making them the unit of sharing and replication across multiple sites (Section 4.2).

This alignment extends to build scripts, where Java developers specify what to compile and assemble into a system. When a package depends on other packages, its build script must explicitly *import* the desired versions of each. The import statement in build scripts (Section 4.4) thus serves as the glue for assembling large systems.

A second weakness of the Java package construct concerns naming inside programs. Global name management requires inconveniently long names, for which the best solution would be a language extension that legitimizes the code fragment shown in Figure 2. Meanwhile, we advocate

a programming style in which unqualified class names appear in code, accompanied by explicit per-class Java import statements; long global names will then be needed only to resolve name clashes, which are readily detected by the compiler in the presence of explicit imports. The environment adds some support by providing a mechanism for mapping between class names within the language and longer, globally-unique storage names that have additional package prefixes.

4.2. Packages in the storage system

Package naming independent of storage location is crucial, both locally and when distributed development requires federated storage.

The environment extends the alignment between the language and storage namespaces beyond what is encouraged by current Java tools, to the point where developers seldom need to distinguish between them at all (following Vesta, derived objects are managed in a persistent cache and do not appear in the namespace at all.). The name `COM.sun.labs.jvb` can mean both the set of classes declared with a "`package COM.sun.labs.jvb`" statement, and the set of stored compilation units that implements the classes. Tools for locating and browsing Java code elide information related to local context, for example allowing access to sources for class `Stack` purely via its language name `COM.sun.labs.jvb.util.Stack`. Other source objects, such as related HTML documents, can be named analogously and can reside in storage system packages.

One aspect of the language confounds this approach. Classes have privileged access to certain fields and methods of other classes declared in the same package. New classes might declare themselves to be "in" an existing package developed by a different organization, violating the proposed namespace alignment. We discourage the practice, but when necessary we recommended that the storage system apply a prefix to the Java package name, as mentioned in the previous section. Even if the names do not match exactly, some benefit will accrue from sharing a structural similarity.

We propose to abstract over existing mechanisms for managing storage distribution (for example network file systems, http, and distributed databases), both for portability and to provide a uniform and simple model for developers. This requires adoption of the Java global naming discipline [7] that derives package name prefixes from reversed internet domain names, for example the package prefix `COM.sun.labs` from the domain name `labs.sun.COM`. Even when code uses shorter names internally, such as `jvb.util`, we encourage globally unique names such as `COM.sun.labs.jvb.util` in storage system packages.

Projects are storage units that occupy distinct subtrees of the global namespace. Storage is ultimately located by a URL such as `file://projects/jvb`, but user-interaction takes place purely in terms of package names. For projects located by the http protocol, we expect but do not require that the server component of the URL be related to the project's package namespace, for example `http://labs.sun.COM`.

Although the package namespace provides the illusion of a *one-world* model, individual projects are necessarily separately administered, forming a *federated* database. In particular, it is not practical to guarantee eternal referential integrity for inter-project references in a WAN. This concession to practicality permits individual projects to be garbage collected independently. Inter-project references can only be created under the control of the system, thus allowing inter-project negotiation over link stability.

4.3. Packages and configuration management

Packages also serve as configurations, which developers assemble using two mechanisms:
* *Direct Inclusion*: placing sources into a package version, thereby assuring that any use of the package version will include them.
* *Importation*: placing into a package's build script an import statement that references a specific version of another package.

These form a simple and powerful discipline for assembling arbitrarily large Java systems in the context of the Java package namespace. It is also possible to describe inter-version construction (systems that contain or use more than one version of a package); this generality is essential to describe systems that host their own evolution. Our prototype environment is an example of such a *self-evolving system*.

The proposed discipline leads to three stereotypical source organizations of Java sources that reflect different intentions about how they will be built and reused: versioned packages, nested packages, and umbrella packages.

A *versioned package* is a collection of sources (and possibly nested packages) that exists in stable, named versions (for example `COM.sun.labs.jvb.util.7`), even as the package evolves. It may import other versioned packages, although this may not be externally visible. It can be built in isolation, producing derived objects (typically a set of class files and possibly an applet or program) that are said to be the result of the build. Versioned packages are the reusable modules of scalable Java development.

A *nested package* is a collection of sources that resides in the namespace below a versioned package (for example `COM.sun.labs.jvb.util.test`), and which is both

included in and versioned together with the versioned package. A nested package cannot be built in isolation, but rather depends on its enclosing context. Nested packages typically implement architectural variants or break up a large package (but cf. umbrella packages, below). For maximum reuse, nested packages should be restricted to cases where real nested dependencies exist.

An *umbrella package* (a Vesta term), though technically a versioned package, is distinguished by the intention to aggregate other versioned packages into larger units, doing so by importation rather than direct containment. Umbrellas, as do all versioned packages, exist in versions, and each version represents a choice of which versions of other versioned packages to import. An umbrella can import other umbrellas, enabling stable descriptions of very large systems at varying levels of detail.

The result of building an umbrella will not in general be equivalent to an aggregation of separately built results from its imported packages. Build scripts can override parameters in the build scripts of imported packages, for example by specifying a different compiler or debugging option.

An important kind of Java umbrella imports immediate children in the package namespace. For example, if `java.awt.image` and `java.awt.peer` (see Figure 1) were versioned packages, and if package `java.awt` were an umbrella, `java.awt.1` might import `java.awt.image.5` and `java.awt.peer.6` and `java.awt.2` might import `java.awt.image.9` and `java.awt.peer.8`. The effect is similar to ClearCase versioned directories [4].

A developer organizes the package namespace by choosing nodes in the hierarchy where versioning appears, and thus where the transition between versioned package names and nested package names appears. For example, the name `COM.sun.labs.jvb.builder.5.store` denotes version 5 of a versioned package named `COM.sun.labs.jvb.builder`; the suffix `store` denotes a nested package that has been separated for implementation convenience, but which the system does not permit to be built independently. A nested package can always be promoted to a versioned package at the cost of some restructuring. An umbrella named `COM.sun.labs.jvb` would typically group all its versioned packages, such as `builder,` into a convenient whole.

The goal of aligning the language and storage system namespace interacts subtly with the three proposed classifications for packages. For example, in the core Java packages there are sources at node `java.awt`, meaning that this should be a versioned package. How then should sources at nodes `java.awt.image` and `java.awt.peer` be organized: as nested packages within `java.awt` or as versioned packages themselves? The latter choice provides more flexibility but has the unfortunate potential for a name clash, for example should a nested package named `image` be added to some later version of `java.awt`. This is not particularly easy to check against, and may be harmless in any event. In our current design we do not prevent or warn about this condition.

4.4. Packages and building

The environment produces information derived from a package version (for example executables) by invoking its build script. Invocation may involve specification of some parameters left unbound in a build script, typically those associated with platform variants, or it may not when all such parameters are bound.

Build script invocation produces a *package object* that encapsulates its results. Package objects persist and provide useful reflective information about the package, for example identifying all sources and resulting derived objects. It also references package objects for recursively built imported packages, thereby representing the complete build, no matter how large the system. Vesta-style caching in the build script interpreter, combined with the repeatable build guarantee, ensures build avoidance when possible.

The build system manages an internal namespace of package objects in alignment with the storage and language namespaces. Returning to the `COM.sun.labs.jvb` example, the name `COM.sun.labs.jvb` can also denote the compiled collection of classes and related resources. Construction of applets, along with like-named resources such as HTML files, can be done with straightforward scripting techniques.

5. Implementation strategy

We have argued elsewhere that conventional operating system facilities for persistent storage (directories and files) are inadequate and should be replaced by persistent storage of programming language objects [8]. We emulated this approach with files in an early pre-prototype (written in Tcl/Tk [13]) that was inspired and influenced by m3build [16], the build system for Modula-3. This pre-prototype is in regular use, hosting the Java-based implementation of its successor.

The prototype now under construction is based on an experimental implementation [3] of orthogonal persistence [2] for Java. In fact, this environment is highly characteristic of the *persistent application systems* for which orthogonal persistence is intended.

Persistent Java objects model all aspects of the software development process, from small objects used in compilers and associated tools, through objects used to represent packages and versions, to large objects that represent com-

plete system builds. Navigation and querying is possible for all objects, subject to access control constraints. Versioning is ubiquitous and convenient; editors can be integrated directly with the versioning system and are able to exploit the system to evolve objects incrementally.

This version of the environment naturally allows integration with existing Java applications, for example the HotJava™ browser or the Java compiler. Amongst other things, this would enable a more graceful response to version mismatch when loading applets, by automatically generating a new version in the persistent store The properties of the system support incremental construction for all objects, including derived objects generated by compilers, which then enables reliable incremental software upgrades, at an appropriate level of granularity.

6. Related work

The proposed approach is based heavily on the Vesta system [10], as discussed throughout, which takes an innovative position on the relationship among configuration management, building, and storage management. Unlike Vesta, whose package mechanism was independent of its target language (which has no packages), we target a language with an existing package mechanism. Our goal is additional leverage, but without undue change to the language or current practice.

7. Conclusions

We have presented a model supporting large-scale software development in Java that is based on the idea of versioned packages. Conceptual simplicity is achieved by exploiting the package concept at the language level, in the persistent storage system, and in the system build process. Stability is achieved by requiring that published packages exists as stable, immutable versions which encapsulate their dependencies. The versioning model is simple and familiar and provides for variants as well as serial development. Large scale development is supported by establishing a global package namespace, based on Internet Domain Names, that is implemented as a collection of loosely connected federated databases.

An extension to the Java import mechanism would significantly increase the suitability of Java for large-scale programming.

We are constructing and using a prototype development environment that supports this model. Early experiences have been positive, although somewhat confounded by lack of adherence to global class naming conventions. Simplified name management in particular (both the alignment of names, as well as the non-naming of derived objects) has

proven extraordinarily helpful, and we find ourselves encountering far fewer errors while working within the prototype than we do when we must work with the file system.

On the other hand, it has become clear that additional tools are needed to help visualize and manage the package namespace. A necessary consequence of version stability is that the package namespace in a particular store becomes cluttered with obsolete package names and outright naming mistakes that have been corrected by migrating development into other names. We anticipate a context-based view mechanism that helps visualize the relevant parts of the namespace in a store.

We are especially encouraged to see that problems of scale now being discussed by members of the advanced Java programming community are in general those addressed by this environment.

8. Acknowledgments

9. Trademarks

Sun, Sun Microsystems, HotJava, Java, and Solaris are trademarks or registered trademarks of Sun Microsystems Inc. in the United States and other countries. SPARC is a trademark of SPARC International, Inc., licensed exclusively to Sun Microsystems Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open, Ltd.

References

[1]   *Ada 95 Reference Manual*, International Standard ANSI/ ISO/IEC-8652:1995, Jan 1995.

[2]   M.P. Atkinson and R. Morrison, "Orthogonally Persistent Object Systems", *VLDB Journal* **4**,3 August 1995.

[3]   M.P. Atkinson, M.J. Jordan, L.Daynes and S. Spence, "Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system"*, Proceedings of the 7th International Workshop on Persistent Objects Systems*, Cape May, New Jersey, May 1996.

[4]   *ClearCase Concepts Manual*, Atria Software, 1992.

[5]   M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[6]   S. I. Feldman, "Make -- A Program for Maintaining Computer Programs", *Software--Practice & Experience* **9**,3

March 1979, 255-265.

[7]   J. Gosling, W. N. Joy and G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.

[8]   M. Jordan and M. L. Van De Vanter, "Software Configuration Management in an Object-Oriented Database", *USENIX Conference on Object-Oriented Technologies (COOTS)*, Monterey, CA, June 26-29 1995.

[9]   M. Jordan and M. L. Van De Vanter, "Large Scale Software Development in Java", Sun Microsystems Laboratories Technical Report, 1997 (in preparation).

[10]   R. Levin and P. McJones, *The Vesta Approach to Configuration Management*, Research Report 105, Digital Equipment Corporation Systems Research Center, June 1993.

[11]   T. Lindholm and F. Yellin, *The Java Virtual Machine*, Addison Wesley, 1996.

[12]   Greg Nelson (Ed.), *Systems Programming in Modula-3*, Prentice Hall.

[13]   J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, 1994.

[14]   P. Rovner, R. Levin, and J. Wick, *On Extending Modula-2 for Building Large Integrated Systems*", Research Report 3, Digital Equipment Corporation Systems Research Center, January 1985.

[15]   *CodeManager User's Guide*, Sun Microsystems Inc., Part No. 801-2169-11.

[16]   *m3build*, http://www.research.digital.com/SRC/modula-3/html/m3build/m3build.html, Digital Equipment Corporation 1992.