9th Intl. Symposium on System Configuration Management (SCM-9), 5-7 Sept. 1999 © Springer-Verlag

Global Names: Support for Managing Software in a World of Virtual Organizations

Michael L. Van De Vanter[1] and Tobias Murer[2]

[1]Sun Microsystems Laboratories, 901 San Antonio Road
Palo Alto, CA 94303 USA
Michael.VanDeVanter@Eng.Sun.COM
[2]TIK (Computer Engineering and Networks Laboratory)
ETH Zürich, 8092 Zürich, Switzerland
murer@acm.org

Abstract. Emerging technologies such as the Internet, the World Wide Web, Java™ technology, and software components are accelerating product life cycles and encouraging collaboration across organizational boundaries. The familiar coordination problems of large scale software development reappear in a context where tools used by collaborators must be less tightly coupled to one another than before. To the traditional notion of scale, based on the size of software systems, must be added a new dimension of scale: organizational complexity. Designing configuration management systems that scale well over both dimensions requires difficult trade-offs between reliability and flexibility. At the heart of these trade-offs is the aggregate information shared by collaborators: how it is represented, maintained, and understood by the people and tools using it. While designing a prototype development environment intended to scale in both dimensions, we have revisited the role played by naming. A proposed extension to the prototype's naming system addresses issues such as which objects should be named and how the shared naming system is constructed.

1 Introduction

Software product life cycles are accelerating and increasingly take place within so-called virtual organizations that require cooperation across a variety of organizational boundaries [4]. This trend is supported and encouraged by emerging technologies: the World Wide Web, software components, and Java™ technology [6]. Consequently ever more software is shared across organizational boundaries and assembled in increasingly dynamic and varied ways, such as with components and plug-ins.

Highly evolved tools for configuration management, which address familiar problems of scale, have become indispensable. However, these tools, even when reworked for the Internet, often fail to address the diversity of collaborating organizations. To the traditional notion of software size, which we characterize as *compositional complexity,* must be added a new dimension of scale: *organizational complexity.* Tools must now address both dimensions.

The Forest project at Sun Microsystems Laboratories has been developing JP, a prototype environment designed for reliable development of compositionally complex systems written in the Java programming language. Complementary issues of organi-

zational complexity and the broader software life cycle are being addressed, in collaboration with the Virtual Software House project at ETH Zürich, by the *application web*. The application web supports the software life cycle across organizational boundaries, striking a balance between autonomy and collaboration [13].

It became clear that the application web depends critically upon global naming of shared information. JP supports a simple naming system that operates both in the small (editing based on names for individual sources [15]) and in the large (configurations of arbitrary size and complexity shared across multiple JP repositories [9]). Sharing data among multiple organizations and tools, however, demands a richer naming system.

It also became clear that design decisions for such a naming system encounter trade-offs between reliability and flexibility: for example what to name at what granularity and how to support reliable bindings. These decisions require careful thought about the roles played by naming.

A design strategy was adopted in which names are only given to objects as needed, and whose bindings are as reliable as possible. The results of this strategy reflect goals for different parts of the system:

- core configuration management and build systems are designed for utmost reliability, and rely on object structures, not names;
- development tools use a simple, global naming system for JP environments, designed to make information intelligible; and
- the application web uses names to support dynamic and flexible information sharing across organizational boundaries.

The system is cast as an instance of a general framework for naming, the Java Naming and Directory Interface™ (JNDI) standard [8].

This paper discusses the design issues that arose in the development of this strategy, as well as the resulting concrete proposal. Section 2 begins with background on scaling issues, in particular the different demands of compositional vs. organizational scale. Section 3 takes a closer look at specific design issues concerning names in JP and the application web. Section 4 describes the proposed extension to JP naming, followed by a discussion of related work and conclusions.

2 Technologies for Scale in Software Development

The JP programming environment and the application web are designed to solve problems of scale for tools supporting the software life cycle, but along fundamentally different dimensions.

2.1 Compositional vs. Organizational Scale

"Scale" for programs was once measured in lines of code, but the real issue here is complexity. Configuration management systems (including JP) address the *compositional complexity* of systems: the number of modules, versions, variants, platforms, and languages that it takes to construct them. *Organizational complexity*, on the other hand, arises in the presence of virtual organizations (dynamic networks of organizations that cooperate for mutual benefit [4]) and involves the whole life cycle of prod-

ucts. Autonomy drives organizational complexity; single software development organizations don't face the differences of culture, infrastructure, methods, tool preferences, and skills one finds at organizational boundaries in virtual organizations. The application web addresses these aspects of scale.

Tools for compositional complexity must be reliable; tools for organizational complexity must be flexible. Tools that address both aspects of scale (organizational complexity often implies compositional complexity too) face design trade-offs. For reliability, information is best managed in a tightly coupled fashion, as if in a single global data structure with complete referential integrity and type safety. Flexibility, on the other hand, requires information that is less tightly coupled and more open [13] so that information may be created and managed by autonomous organizations, selectively shared, and structured simply for intelligibility. General naming systems, for example URLs, achieve flexibility at the cost of reliability.

The challenge is to find a useful balance. JP was designed to scale with compositional complexity; the application web addresses the additional requirements of organizational complexity.

2.2 The JP Programming Environment

JP is a prototype programming environment for the collaborative, reliable development of compositionally complex systems written in the Java programming language. It is based on close coupling among federated JP repositories, and tool integration via object-oriented interfaces. Implementation simplicity and reliability derive from architectural orthogonality among core services, functional programming, and aggressive use of abstract object interfaces.

Central to the JP approach is the notion of uniquely named, reusable, independently versioned packages [9]. JP packages play many roles:

- *System Structure.* Package versions act as software modules: they contain human-created artifacts such as source code and can use specific versions of other packages by *importing* them.
- *Storage Management.* Package versions and *derived* objects built from them, as well as tools such as compilers and editors, reside in repositories of orthogonally persistent objects [1].
- *Building.* A package version is built by interpreting its *build script*: a functional program that recursively invokes the scripts of imported packages. Previously derived objects are reliably shared via function caching, a mechanism that is largely orthogonal to the type of objects and the tools that create them.
- *Versioning.* Versions of a package are managed by a simple version system that is orthogonal to version content.
- *Configuration Management.* JP configurations are implemented as packages whose role is to import particular versions of other packages, including other configurations. Each version of a configuration specifies an immutable, arbitrarily large, aggregation of packages.
- *Analysis.* Tools for analyzing and visualizing software need not be separately configured, since they have direct access to configuration contents and derived results.

2.3 The Application Web

The *application web* complements JP by addressing problems of organizational scale [13]. Organizational boundaries discourage tool-based collaboration, in the absence of which information must be copied and shared manually. The application web fills the gap between the informality of copying and the tight coupling of conventional tools.

The application web provides an infrastructure for sharing information and supporting collaboration across boundaries within virtual organizations. This brings the character of the WWW into the software life cycle, where information can be autonomously maintained at its origin, but can also be shared through simple protocols. A software application can consist of parts originating at many sites.

The application web provides rich *connectivity* that spans the life cycle of software products, beginning with construction and continuing through deployment and ongoing management. For example a deployed, possibly running, application can be queried for complete, precise, and relevant information about its configuration, available via links back to the organizations in which the parts were created.

Autonomy is just as important. Shared data models are as simple as possible, following the lead of the World Wide Web, enabling dynamic and loosely coupled collaboration.

Flexible collaboration is supported by multiple layers of access. Closely related organizations might construct software jointly, using distributed authoring tools or JP's federated build system. Loosely related organizations might have limited HTTP access to the others' repositories. In between might be application loading services, bug reporting and tracking, querying for component interoperability, and reference information in support of debugging.

3 Naming Systems: Roles and Requirements

Naming is central to the shared information upon which the application web is built. The design strategy adopted for the naming system reflects the importance of a balance between competing requirements for reliability and flexibility. This section discusses general issues, as well as choices made for different parts of the system.

3.1 Design Issues for Naming

The first question when providing name-based access to complex data is whether to do it at all.

Closely coupled tools access data via *object references*, whose referential integrity and type safety derive from modern programming languages. The persistent object system used by JP makes object references suitable for reliable, long-term data storage, but such references are not available outside system boundaries.

In contrast, a *naming system* typically offers access to data from "outside" system boundaries. Names are legible to people (names often encode contextual information and may be redundant) and portable (they can be written down and emailed), but this flexibility comes at the cost of decreased reliability when compared to object references.

At one extreme, every object might be named, mirroring an object reference structure; for example, every element in a hierarchical file system is named. Conversely, only a small number of objects might be named, as with "persistent root" objects in object-oriented databases, requiring that further access be structural. These choices depend on the information clients need and what they know about the structure of the data. Other design issues include the lifetime and mutability of name bindings as well as lookup mechanisms and accessibility guarantees.

Unfortunately developers routinely suffer *too many naming systems* that are badly suited for the task: one for the target language (e.g. class names), another for storage (location dependent file names), possibly a third for modules (often modeled weakly as directories), and others for versioning and configuration. Not only must developers understand all these naming systems, they must keep them arranged in complex relationships just to keep the tools working. The problem is made worse by *naming too many things*, for example by cluttering the name spaces for sources with derived objects.

The proposed strategy is to name as few objects as possible, depending on specific requirements. The rest of this section describes the roles played by names in different parts of the system. Whereas the JP build system uses no names at all, the JP tool interface layer uses a unified naming system that spans multiple JP repositories. The former approach permits reliable building for configurations of arbitrary complexity, and the latter provides a comprehensible user model that abstracts away inessential information. Extending JP into the application web, requires a third approach to naming, one that will serve also as a bridge to non-JP tools.

3.2 Naming in the JP Build System

At the heart of JP is a build system that provides stronger guarantees of reliable and repeatable builds than is now common. Several technologies support these guarantees, but naming is not among them.

In order to be built, a *package version* must be committed to the repository, and it must completely specify its build computation. In its build script (a functional program) source objects (those created by humans using tools such as editors) appear as literal data values of the scripting language in declarations equivalent to:

```
JavaSource myStack = <the text written by a developer>;
```

JP source objects, known as *parts*, are implemented with *functional objects*: they are immutable and can be treated as pure values whose object identity plays no role [2]. JP parts are context-free and not intrinsically named and can thus be safely shared by many contexts.

Perhaps surprisingly, it is also necessary that package versions themselves be represented as functional objects; they participate in build scripts as literal values in import declaration equivalent to:

```
import <reference to contents of an existing package ver.>;
```

To the extent that parts do have names in the build system, it is only for the internal purposes of the computation, for example as in the declaration of **`myStack`** in the first

example. These local bindings, within the computation engendered by each JP package version, are isolated from the rest of the environment and from other versions.

Objects created during building are likewise bound for the duration of the computation. Such bindings persist only to the extent that they are captured in a returned *build result*. The result is generally not named; tools invoke the **`build()`** method on package versions and operate structurally on the returned value.

3.3 Local Names in the JP Environment

It is one thing to exploit the power of a purely functional build system computing over a repository of context-free objects treated as values; it is quite another to help developers create, understand, and manage such objects. People and their work are inherently contextual and must be able to understand data in their own context.

Tools in a JP repository use a simple, unified name system for versioned packages and their contents [9], as shown in Example 1 and described further in Section 4.1.

Example 1. JP Names

`com.sun.labs.forest.jp.util`	a JP package
`com.sun.labs.forest.jp.util/7`	a JP package version
`com.sun.labs.forest.jp.util/7#Stack`	a source ("compilation unit") in a JP package version

These are the only names a developer normally sees, based on the design decision that names do not need to carry any other information, and that no other objects need names.

Unlike many naming systems, these names are reliable: name bindings, once created, are eternal, immutable, and always accessible. This makes names redundant, strictly speaking, but they help make a crucial connection between abstract, buildable values and the work being done by developers. This separation between internal representation and user-visible names permits the design of each to be optimized for its own purposes. For example, the package name space is aligned with names in the Java programming language, eliminating any distinction between storage and language names. The version name space in JP supports a simple branching and numbering model, but any name space would do; version names can therefore be aligned with local software development processes, once again eliminating name distinctions.

Names are used to good advantage by JP's framework for editor coordination [15]. This framework allows JP developers to commit new package versions, which can then be built if desired. Constructing a version involves creation of new source objects, based on editing activity, as well as new folder-like containers that represent changed contents. The framework makes this process nearly transparent, even in the presence of multiple editors that are not version-aware. It also arranges that unedited parts be shared by successive versions. Although the framework operates structurally, editors appear to be operating in the JP name space. Other tools, for example for navigating, searching, and creating new objects, operate similarly.

3.4 Implementation Challenges

Supporting both structural and name-based access to information has its costs. JP exploits the implementation of parts as immutable values and permits a value to be bound to many names, for example in successive versions of a package in which the part has not been touched. Consequently, any human view of a part must supply appropriate context.

For example, from the perspective of the build system an import is bound to the *content* (value) of a package version, not a name for the content. A person examining a build script, on the other hand, expects to see the human *intention* behind the import, which is best captured by the name used when the import was created. Thus, the internal representation of a build script import must carry this extra contextual information, however inessential to the builder.

Likewise, build errors cannot be reported usefully when a compiler sees source code only as byte arrays embedded in a graph of functional objects. Each invocation of the build system must be given a way to "re-contextualize" any parts mentioned in communication with humans, for example so that programming errors can be located and corrected.

JP editors, like the builder and compilers, traffic only in object values: old values are copied into buffers, and new parts are created when needed. The editor coordination framework maintains context that explains the meaning of data in each editor buffer. The framework informs each editor of the current name for each buffer, which routinely changes as versioning progresses, even though its only purpose is to give the developer contextual information.

Bridging different areas of JP's architecture has deeper implications in the underlying implementation, since a JP repository must be able to manage parts created by different versions of editors and derive results using different versions of compilers. This requires, in effect, a separate type system for each configuration, the consequences of which are beyond the scope of this paper. These issues do not arise, of course, in systems where tools and data are not strongly typed.

3.5 Global Names in the JP Environment

In order to support development of large systems, JP must permit collaboration among developers using multiple, federated JP repositories. The challenge is to approximate as closely as possible the guarantee of reliable, repeatable builds made by JP in single repositories.

The organizational aspect of the problem requires a global naming policy. JP aligns package names with the emerging *global name space* for Java packages, which begins with reverse DNS names. This grants organizations exclusive authority to create names in owned domains, which they can subdivide as needed. JP makes the aggressive presumption that a name, once created, has constant meaning, viewed from any JP environment, anywhere in the world, for all time.

JP imports are permitted to cross repository boundaries, which presents implementation choices for the representation of a non-local import. Two solutions are being explored, functionally similar but with very different architectural implications:

– At the platform level, an experimental mechanism supports persistent remote references to persistent objects.
– At the application level, package version names can, when combined with a locator service, implement the same binding.

Although neither of these mechanisms can guarantee that remotely implemented bindings are either eternal or accessible, it is possible to check that bindings are immutable. Object *fingerprints*, upon which JP's function caching is based, can be stored with the representation of an import, making it possible to verify that the retrieved value of a remote import, if available at all, is the intended one.

3.6 Naming Requirements for the Application Web

Whereas JP embeds names in a closed world designed for reliable, federated building, the application web is designed for more flexible forms of collaboration that span organizations, phases of the life cycle, and tools [13]. The application web captures interconnected information associated with software development, and makes it available via a variety of tools throughout the software life cycle. Collaboration requires some common understanding about software systems and their structure, as reflected in a shared naming system. Such a naming system must balance requirements for expressiveness (for effectiveness), reliability (for compositional scale), and flexibility (for organizational scale). The goal of expressiveness suggests that the JP name space be extended by naming objects that otherwise would only be treated structurally. Examples include build results and the internal structure of configurations.

The World Wide Web is an example of a scalable, global name system that meets many of these requirements. However, the application web presents additional requirements such as versioning and configurations, such as those used in JP.

Equally important is the reliability of names in a well-defined life cycle, where names are guaranteed to persist and be immutably bound. Confidence that bindings will be available and will not change encourages effective and efficient communication among people and tools *by reference* to objects. It also permits reliable name-based caching. On the other hand, it must be understood that bindings can be subject to service interruptions and expirations.

Authentication, trust, and access control are also essential, but are beyond the scope of the current work.

4 The Extended JP Naming System

The extended JP naming system addresses the goals of the application web and is based on experience with a simple prototype. This required recasting the original as a composite naming system, making some syntax adjustments for scalability. It also involved extending its scope, and defining more carefully such issues as the life cycle of names.[1]

1. This description uses the terminology of the Java Naming and Directory Interface (JNDI) standard [8]. Name syntax is described on an EBNF notation that includes '['..']' for options and '{'..'}' for zero or arbitrary repetitions.

4.1 Package Versions and Contents

Editing, versioning, system modularity, and building all operate at the granularity of package versions.

- Every package and package version has a *canonical name* that identifies it uniquely in the global package name space.
- Every part within a package version may also have a *canonical name* that identifies it uniquely.
- This *composite naming system* includes separate naming systems for packages, package versions, and parts.

*Canonical_Name = Package_Name [‘**/**’ Version_Name [‘**#**’ Part_Name]]*

The Package Naming System. This naming system mirrors the increasingly accepted global name space for Java packages [6], based on reverse DNS names. The hierarchy implies no inclusion relationships, either in the Java programming language or in JP; for example **a** and **a.b** name unrelated packages.

*Package_Name = Atomic_Name { ‘**.**’ Atomic_Name }*

Example 2. Package Naming System

COM.sun.labs.forest.jp	canonical package name

The Version Naming System. This naming system identifies each version relative to its package. JP version names are hierarchical, with alternating numbers and names, as in Example 3. However, other versioning models can be used to suit particular development processes, as mentioned in Section 3.3. This freedom permits organizations to share information based on names, even when the semantics of particular version names are not shared.

A canonical name includes at most one version name, reflecting the decision to version packages only *in toto*. Packages in which contents are to be versioned independently must be constructed as configurations of other packages.

*Version_Name = Number { ‘**.**’ Atomic_Name ‘**.**’ Number }*

Example 3. Version Naming System

1.murer.4	version name
COM.sun.labs.forest.jp/1.murer.4	canonical package version name

The Part Naming System. This naming system identifies human created source objects within the content of a version. In contrast to package names, hierarchy *does* imply inclusion. Version content is managed as a single compound document, whose root part name is null, and in which embedded parts may or may not be named.

*Part_Name = Atomic_Name { ‘**.**’ Atomic_Name }*

Example 4. Part Naming System

`model.layout`	part name
`COM.sun.labs.forest.jp/1.murer.4#model.layout`	canonical part name

Alternatives. The above syntax is influenced by URLs. Other approaches could be used, so long as package names match the name space for Java packages, and composite names can be parsed unambiguously. An earlier JP naming system used the separator '`.`' exclusively, as in Example 5:

Example 5. Old JP Naming System

`COM.sun.labs.forest.jp.1.murer.4.model.layout`	canonical part name

The simplicity of the old approach is appealing, but it promised to become confusing with increasing richness of the naming system, since boundaries between the individual naming systems are not immediately obvious.

4.2 Configurations

An extension to the JP naming system makes more of the internal structure of configurations visible through naming.

Projections. The global package name space is immense. Developers work in subsets that include just the package versions aggregated into systems under development. These subsets are defined by configurations: package versions that recursively aggregate other package versions, as in Example 6:

Example 6. Configuration Names

`CH.ethz.ee.tik.vsh/8`	configuration version name

`import COM.sun.java.JDK/2.mac.3`
`import COM.sun.labs.forest.jp/4`
`import CH.ethz.ee.tik.vsh.services/9`
	imports of package versions in the configuration

The contents of a configuration version can be treated as a projection of the global name space in which names can be used that have meaning only relative to the particular context of the configuration, as in Example 7. In this example a part appears in a configuration version because its containing package version is explicitly imported; three different names refer to the same part:

- The canonical name, which is independent of any configuration.
- A configuration-relative name, based on an extended syntax as shown. All such names can be unambiguously translated to canonical names.

Example 7. Configuration-relative Names

`CH.ethz.ee.tik.vsh/8`	configuration name
`COM.sun.labs.forest.jp/4#Main`	canonical part name
`CH.ethz.ee.tik.vsh/8/COM.sun.labs.forest.jp#Main`	part name embedded in configuration context
`COM.sun.labs.forest.jp#Main`	part name relative to implied configuration context

– A configuration-sensitive name that only identifies the part uniquely in a context where the configuration version is implied. These shorter names are the ones JP tools might display when a developer is working in the context of an evolving configuration. In situations where even more context is implied, even shorter names might appear, for example **`jp#Main`**.

The particular projection used in Example 7 is only one of many that could be designed to suit various needs for visualizing and working within the context of configurations.

Complete system descriptions. Configurations capture more than sources. For example, they contain a complete prescription for building it, including the precise version of a compiler (which also comes from a JP package version), compiler options, and which version of important libraries it is compiled against. These are all examples of *meta-information* relevant to the configuration. Other kinds of parts might also be present, for example design documents and test case specifications. Some kinds of information might not be naturally represented as parts, in which case the naming system might be extended explicitly.

4.3 Derived Parts

Information created by building a configuration, although guaranteed by JP to be well-defined, is not canonically named and is understood to have no meaning outside of its originating configuration. For the purposes of the application web, any build result should contain enough information to identify its configuration, perhaps by links leading back to the repository in which it was originally created.

Access to derived information is structural within the JP environment, but access can also be provided by describing each build result as a new name space in the composite naming system, relative to its configuration version. This might name such useful information as compiled classes and Javadoc HTML files. A function applied to a given configuration builds the naming context for a *derived parts naming system*. This is another hierarchical naming system, for example:

*Derived_Part_Name = Atomic_Name { '**.**' Atomic_Name }*

In Example 8 **`compile`** is a distinguished name that refers to the derived name space. The names that follow can be simple, in situations where build scripts explicitly

Example 8. Derived Part Naming System

`util.Connection`	derived part name
`CH.ethz.ee.tik.vsh/8/compile#util.Connection`	derived part name in context of a configuration's build result

define such names, or they might be expressions whose evaluation would provide something approximating structural access to the information. The names are reliable in either case, a considerable advantage when caching derived information.

4.4 The Life Cycle of Names

The use of global names within the application web requires a set of rules about how names are created and managed. These rules may be summarized in terms of the *life cycle* for canonical names; projected and derived names are always well-defined in terms of canonical names, as discussed earlier.

- *Creation.* A new name is explicitly created by an autonomous organization participating in the application web. Intellectual work is recorded in JP by committing a package version to a repository, and this requires that it be named; likewise, information cannot be shared in the application web until it is named.
- *Uniqueness.* A newly created name is presumed never to have been used before. The authority to create valid names is managed at the topmost level by partitioning the global name space into DNS domains that organizations own and can subdivide as needed.
- *Persistent Binding.* A name is bound to a value when created, and this binding may never change. This permits loosely coupled organizations to communicate reliably using names, and permits reliable caching of bindings.
- *Unavailability.* No non-local lookup service can be constantly available. The bound value of a name may not be available in situations where there are system failures and there is no local cache available.
- *Eternal Names.* A name, once created, must live forever. Even if a binding expires at its source (see below) caches may live on; names must be remembered so that they will not be rebound.
- *Binding Expiration.* Although names live forever, the storage of accumulated bindings may not always be practical or desirable. Expiration dates would help organizations negotiate the lifetime of their storage services, much as other artifacts in the software business eventually expire.

The rules of the life cycle for names cannot be strictly enforced in the world for which the application web is designed. The success of names used this way must rely on the motivation of participating organizations to follow the rules for their own interest, combined with end-to-end tests to ensure that bound values never change.

5 Related Work

The Vesta project [10], from which JP's core build technology is derived, implemented

functional building over a repository of immutable versioned packages. Its package name space is flat, local, and not related to language names, and there is no integrated support for integrated editing.

Distributed configuration management systems have been developed, some commercial such as ClearCase Multisite [3]. These generally presume every site is running the same tools. This restriction can be lifted by designing *middleware* to glue together different systems, for example by Kaiser and Dossick [5]. These approaches address compositional complexity, but often neglect the difficulties of organizational complexity.

Noll and Scacchi address much the same goals as ours with a shared distributed CM system that connects to each organization's tools with adapters [14]. Their design emphasizes a shared model to be understood by all organizations, whereas the application web address a much simpler, open ended model; the application web is less expressive, but may have greater potential for organizational scale. The GIPSY project addresses organizational complexity by proposing a simple, unified model that represents software product, process and organization form [12].

Collaborative authoring tools presume close organizational coupling, although WebDAV aims to bring some of this functionality to the more loosely coupled World Wide Web [16]. Although this is necessarily embedded in a less expressive name space than is needed for the application web, WebDAV could serve as an appropriate infrastructure for parts of the application web.

The application web proposes life cycle connectivity of software to its origin; this permits copying to be replaced by reliable caching. These features allow for reliable software deployment as well as other business opportunities within a Virtual Software House. Such opportunities might include consistent, up-to-date, connected software catalogues as well as component seeking and matching. The Software Dock proposes a sophisticated deployable software description format and an agent based deployment engine to support the software deployment life cycle [7]. Castanet, a product of Marimba, offers incremental software deployment services based on channels, mirroring and fingerprinting technology [11]. In contrast to more sophisticated approaches, the application web promotes the simple "web" idea of connectivity where information relevant for the whole software life cycle is directly accessed from its original source.

6 Conclusions

Reliable, scalable configuration management is essential for developing the next generation of large software systems. Traditional tools fail to help organizations cooperate in emerging models for virtual organizations. Java technology makes some of this easier, but reliable, scalable tools are still needed. Tool strategies for software require a balance between addressing compositional and organizational complexity; names play an important role in these strategies.

The application web, an extension to a reliable, scalable development environment for Java software, addresses the emerging challenge of organizational complexity. It does this in a simple, scalable, collaboration framework based on global naming that permits connecting a wide variety of services, applicable to many phases of the soft-

ware life cycle. This approach presumes that reliable configuration management and repeatable building are among the core services, but it also conspires to make available a wide variety of related meta-information about software.

Early versions of the JP environment are in use, and a simple prototype of the application web, based on HTTP-coupled tools, has been developed for demonstration purposes. It supports several of the anticipated services, for example loading and running applications directly out of their originating repositories, and navigating via information present in running applications back to the sources and documents in the precise configuration in which they were built.

7 Acknowledgments

This work benefits greatly from the vision of Mick Jordan, Principal Investigator of the Forest Project at Sun Microsystems Laboratories and coauthor of JP. The VSH project is supported by Professor Albert Kündig at ETH Zürich and funded by the Swiss Priority Program of the Swiss National Science Foundation. Yuval Peduel and anonymous reviewers made helpful comments on this paper.

8 Trademarks

References

1. Atkinson, M., Daynès, L., Jordan, M., Printezis, T., Spence, S.: An Orthogonally Persistent Java. ACM SIGMOD Record **25** (1996) 68-75
2. Baker, H.: Equal Rights for Functional Objects or, The More Things Change, The More They Are the Same. ACM OOPS Messenger **4**,4 (October 1993) 2-27
3. ClearCase MultiSite `http://www.rational.com/products/cc_multisite/`
4. Davidow, W., Malone, M.: The Virtual Organization: Structuring and Revitalizing the Corporation for the 21st Century. Burlingame Books (1992)
5. Kaiser, G., Dossick, S.: Workgroup Middleware for Distributed Projects. IEEE Seventh International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (June 1998) 63-68
6. Gosling, J., Joy, W., Steele, G.: The Java™ Language Specification. Addison-Wesley (1996)
7. Hall, R., Heimbigner, D., Wolf, A.: A Cooperative Approach to Support Software Deployment Using the Software Dock. Proceedings of the International Conference on Software Engineering, Los Angeles, CA. (May 1999)

8. JAVA NAMING AND DIRECTORY INTERFACE^{TM}(JNDI), `http://java.sun.com/products/jndi/`, Sun Microsystems, Inc. (1999)

9. Jordan, M., Van De Vanter, M.: Modular System Building With Java Packages. In: Ebert, J., Lewerentz, C. (eds.): Proceedings 8th Conference on Software Engineering Environments. IEEE Computer Society Press, Los Alamitos, CA, USA (1997) 155-163

10. Levin, R., McJones, P.: The Vesta Approach to Configuration Management. Research Report 105. Digital Equipment Corporation Systems Research Center (1993)

11. Marimba Inc. Castanet Product Family. `http://www.marimba.com/` (1998)

12. Murer, T., Scherer, D.: Structural unity of product, process and organization form in the GIPSY process support framework. In: Ebert, J., Lewerentz, C. (eds.): Proceedings 8th Conference on Software Engineering Environments. IEEE Computer Society Press, Los Alamitos, CA, USA (1997) 93-100

13. Murer, T., Van De Vanter, M.: Replacing Copies With Connections: Managing Software across the Virtual Organization. 2nd Workshop on Coordinating Distributed Software Development Projects at IEEE Eighth International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises WETICE-8, Stanford University (June 1999)

14. Noll, J., Scacchi, W.: Supporting Distributed Configuration Management in Virtual Enterprises. Proceedings 7th International Workshop Software Configuration Management (ICSE 97 SCM-7), Lecture Notes in Computer Science, Vol. 1235. Springer-Verlag, Berlin Heidelberg New York (1997) 142-160

15. Van De Vanter, M.: Coordinated editing of versioned packages in the JP programming environment. Proceedings System Configuration Management, ECOOP '98 SCM-8 Symposium. Lecture Notes in Computer Science, Vol. 1439. Springer-Verlag, Berlin Heidelberg New York (1998) 158-173

16. IETF WebDAV Working Group, World Wide Web Distributed Authoring and Versioning, `http://www.webdav.org/`