Coordinated Editing of Versioned Packages in the JP Programming Environment

Michael L. Van De Vanter

Sun Microsystems Laboratories
901 San Antonio Road
Palo Alto, CA 94303 USA
Michael.VanDeVanter@Eng.Sun.COM

Abstract. As part of an investigation of scalable development techniques for systems written in the Java™ programming language, the Forest Project is building JP, a prototype distributed programming environment. For extensibility and usability, a mechanism is required to coordinate the activity of multiple editor programs (each specializing in particular source types) with the JP versioning system. The JP architecture makes it possible, using a very simple framework, to coordinate loosely coupled Java-implemented editors that share no data representations with one another or with the versioning system. This framework also supports a streamlined user model for editing that keeps users' version awareness to an absolute minimum during routine development tasks. This architecture relies on two key technologies: orthogonally persistent object storage, and orthogonal versioning of hierarchical, immutable, source objects.

1 Introduction

Constructing and maintaining large software systems demands aggressive use of versioning and configuration management for source artifacts such as program code and documentation. This requirement conflicts in practice with another important requirement, namely that developers can view, modify, and build sources with as little distraction and delay as possible. The difficulty increases when the requirement is added that source objects be edited and versioned as hierarchical aggregations of parts, and that it be possible for new types of sources (along with appropriate editors) to be added during the lifetime of a development environment. A new coordination framework, developed as part of the JP programming environment by the Forest Project at Sun Microsystems Laboratories, addresses all of these requirements.

The JP programming environment specifically addresses problems of scale in software development: large systems are constructed from parts, usually in multiple configurations, developed by teams, possibly at diverse locations. Central to the JP approach is the notion of a uniquely named, reusable, independently versioned package. Each package contains, along with a hierarchical aggregation of sources, information describing how to build it. JP packages 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.

The first and most important set of tools includes those which permit developers to view and modify sources and to create new source versions to be built. Editing and versioning are coordinated with a framework that is simple, easily extended to new type-specific editor types, and supportive of a streamlined user model that minimizes the need for version awareness by developers. Simplicity in the framework is made possible in large part by orthogonality in the JP architecture:

– orthogonal object persistence removes storage concerns from the framework; and
– a versioning model is orthogonal to version content.

Section 2 of this paper introduces the JP environment and points out the particular problem that is solved by the editor coordination framework. Section 3 walks through this framework, discussing the implication of each part at the level of editor implementation and visible user model. Section 4 compares this approach with other work. Section 5 summarizes current project status, followed by acknowledgments and conclusions based on the work so far.

2 JP Overview

JP is an integrated development environment being developed to explore fundamental solutions to the problems of scale that plague development of large software systems [10]. Although the JP approach is general, the current prototype targets systems written in the Java™ programming language [9] and encourages a development style that will result in simpler and more reliable construction of large Java systems [11].

This section reviews the JP approach and describes requirements that motivate the framework for coordinated editing presented in Section 3.

2.1 Package-Centric Development

Central to JP's design is replacement of one of the weakest links in current development environments: the *make* [8] program for system building. JP's adaptation of the Vesta approach [14], a fundamentally more sound and scalable technology, unifies four roles, described below, more commonly supported by disconnected services.

The JP approach is based on independently *versioned Java packages*: collections of classes (as well as other resources) that exist in stable, immutable, versioned configurations. Each version includes a build script: a system model that is also a parameterized program for building the package. The JP 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 four roles played by JP packages.

System Structure. Packages play the role of software *modules*: the developer assembles large systems by constructing packages whose build scripts import other packages. Each import specifies a particular version of an imported package, and the builder's import mechanism abstracts away conventional distinctions between source (uncompiled) and binary (compiled) reuse. Build script imports do not affect the semantics of the Java language (in particular the Java `import` statement).

Storage. The developer manages sources in terms of the package namespace, with the addition of package-level versioning; tools 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 evaluating its associated build script using a special interpreter; imported packages are incorporated by recursive script invocation. The developer gains access to derived objects (for example executables) 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 remotely located) without perturbing package sources. Build script results contain extensive information that is both precise and complete, enabling straightforward access by tools that extract, analyze, and display particular kinds of information to developers. Build results are guaranteed repeatable, as discussed in the Vesta literature [14], enabling derived information to be managed as a cache within the build interpreter; further discussion of build scripts is beyond the scope of this paper.

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. Completeness and precision are crucial to the repeatable build guarantee. A configuration manager permits concurrent non-interfering work by multiple developers.

2.2 Strong Object-Oriented Abstractions

The JP architecture achieves simplicity and robustness through object-oriented design as well as a strong separation of concerns among subsystems. An important technique for decoupling subsystems in JP is reliance on Java *interfaces*. A Java interface defines a new reference type without implementation, but which can be implemented by otherwise unrelated classes that provide implementations for the interface's methods [9]. The editor coordination framework described in Section 3 is based on a Java interface.

Interfaces isolate implementation choices, and even permit multiple implementation choices to coexist. JP subsystems such as the builder, the versioning system, and source viewing and editing are quite independent. Multiple versioning mechanisms could coexist in a single JP store.

2.3 Orthogonal Persistence

Crucial to the independence of subsystems is JP's replacement of a second weak link in current development environments: reliance on simple file systems to store complex, long-lived application data. An implementation of orthogonal persistence for Java [3] permits all JP objects to live as long as needed. Most importantly, it does so nearly transparently, without degrading JP subsystem boundaries.

Objects persist by reachability from a privileged named root, for example a table of versioned packages that reside in a particular store. Object persistence is independent of object type and independent of how objects are created [2]. New object types, added after construction of a programming environment, can persist as well, without any special modifications. This makes all JP objects potentially persistent in a uniform way, much as Java's automatic memory management reclaims storage from unused objects in a uniform, transparent way.

As a consequence, the objects that populate a JP store (for example those representing Java sources) have not been complicated (nor have their interfaces been twisted) by the need to store them as anything other than the objects they are. A second consequence of making objects first class (and not some external file-based representation of them) is that tools such as editors normally run as Java programs on a simple objects-in, objects-out basis. A third consequence is the difficulty inherent in revising the code that implements an installed JP store; this is an instance of the *schema evolution problem,* whose discussion is beyond the scope of this paper.

2.4 Simple Versioning and Configuration Management

Although versioning and configuration management are at the heart of the JP environment, as described in Section 2.1, the JP versioning/CM system is relatively simple. It can be thought of as a monotonically growing map that permanently binds names and immutable content.

Uniform Versioning. All versioning takes place at the granularity of packages. JP configurations, which aggregate package versions, are themselves versioned packages and can recursively represent systems of any size with precision.

Orthogonal Content. JP versioning is orthogonal to content, as suggested by Conradi and Westfechtel [6]. Objects of new types, added after original programming environment construction, can be versioned as well.

Immutable Content. Objects to be versioned must implement a special interface (`Mutability`) that permits the versioning system to ensure, before creating a new version, that the transitive closure of its proposed content is immutable.

Version Accretion. A developer commits changes (prerequisite to building) by creating new versions. JP's versioning model[1] permits versions to be added subject to rules based on named branches with numbered entries. For example, a developer who wishes to work on version `3` of package named `com.sun.pkg` must first invoke a *Checkout* operation on version `com.sun.pkg.3`. Unlike conventional systems, which treat checked out data as mutable, JP creates a new branch, for example beginning with version `com.sun.pkg.3.checkout-mlvdv.0` (whose content is identical to that of `sun.pkg.3`). The developer may add successively numbered versions, typically one for each attempted build,[2] with an *Advance* operation. This work

1. This duplicates the Vesta model, although other models could be added.
2. The justification for this apparently profligate storage policy is beyond the scope of this paper, but experience with both Vesta and JP support it.

might conclude with version `com.sun.pkg.3.checkout-mlvdv.14`, at which the developer would invoke a *`Checkin`* operation to create `com.sun.pkg.4`. JP also supports a *`Branch`* operation; this could be used to create a branch beginning with version `com.sun.pkg.4.mytest.0`. Experiments might be performed on this branch by a series of Checkout, Advance, and Checkin operations, and a merging tool would help migrate changes back to a main branch.

Content Hierarchy. Package contents are typically hierarchical aggregates of objects, analogous to folders and text files in simple cases, but possibly more like compound documents in other cases. Such objects, called *parts* in JP, are by definition immutable and by convention constructed to be independent of context; this permits parts to be treated as pure values that can be safely shared among versions.

Lightweight Versions. Version creation, being little more than extension of a naming data structure, is very light weight. The coordination framework, described in Section 3, takes care to share parts among versions when possible.

Limitations. This simple but robust approach to versioning omits some commonly supported mechanisms, for example dynamically bound names such as "`latest`". Such functionality is supplied in JP by tools. For example, a tool might help create new configurations according to higher level intentions, such as "update all imports to latest versions." Other policy-oriented tools might control visibility and access.

2.5 The Coordinated Editing Problem

A developer makes progress in JP by routinely creating and building new versions. This involves creation of new source objects, based on recent edits, as well as new folder-like containers that represent changed contents. Editing in JP is supported by Java-implemented editors that specialize in particular types of source objects, including containers. Starting with an initial part, a JP editor allows the developer to make changes (as in the mutable *buffer* of a text editor) and eventually to create a new object of the same type. The problem is how to coordinate editors working on various parts, and how to progress smoothly as the developer advances through successive versions.

The coordinated editing framework described in the next section addresses this problem. The intended effect is that developers have the freedom to edit what needs to be edited, to build when desired, and to be able to understand the versioning status of a package on those occasions (preferably few) when it is important. This approximates the kind of freedom offered by single-user integrated development environments, but which is difficult to achieve in a scalable development environment based on strong versioning and configuration management. To summarize, this means that in JP:

– new versions of packages must be created quickly and unobtrusively;
– what's being edited is a hierarchy of parts, in which parts may contain other parts of possibly different type, requiring services of type-specific editors;
– contained types are generally opaque, whose implementation is known only to an open-ended collection of editors that collaborate through generic interfaces; and

– developers must be able to ascertain at a glance the relationship among editors and their version-related status.

3 The Coordinated Editing Framework

Coordinated editing in JP is based on the Java interface `PartHandler`.[1] This section walks through the interface, describing how coordination works and discussing implications for implementations and for developers working in JP.

```
public interface PartHandler {
  // Context and Coordination methods
  void initialize(VersionHandler vh);
  boolean setPart(Mutability part);
  void edit();
  PartHandler getPartHandler(Path name);
  // Versioning methods
  Mutability advance();
  void revert();
  // Usability methods
  void setEditable(boolean isEditable);
  void setModified(boolean isModified);
  void setVersionName(String name);
  void setPartName(String name);
}
```

3.1 Context and Coordination

A part handler (an object implementing the `PartHandler` interface) is not an editor itself, but a coordinator. It collaborates with four types of objects:

– a single version handler,
– a single part within the package version being edited,
– possibly views/editors for that part, and
– possibly part handlers for contained subparts.

Figure 1 shows an example of these relationships; here rectangles represent object instances and arrows represent object references. In the example a developer is viewing and preparing to edit a part named `b.d` in version `3` of package `com.sun.pkg`. The leftmost part of this diagram represents a JP versioned store, where mutation is narrowly constrained: versions may be added, but once created never change; parts, once included in a version, likewise never change. All access to and mutation of the versioned store (in the form of new version creation) is managed by the handlers in the middle part of the diagram. All of the relationships shown in this figure will be discussed in this section.

1. The interface presented here has been simplified for exposition; it omits issues of concurrency, event management, and editor start-up/shutdown.

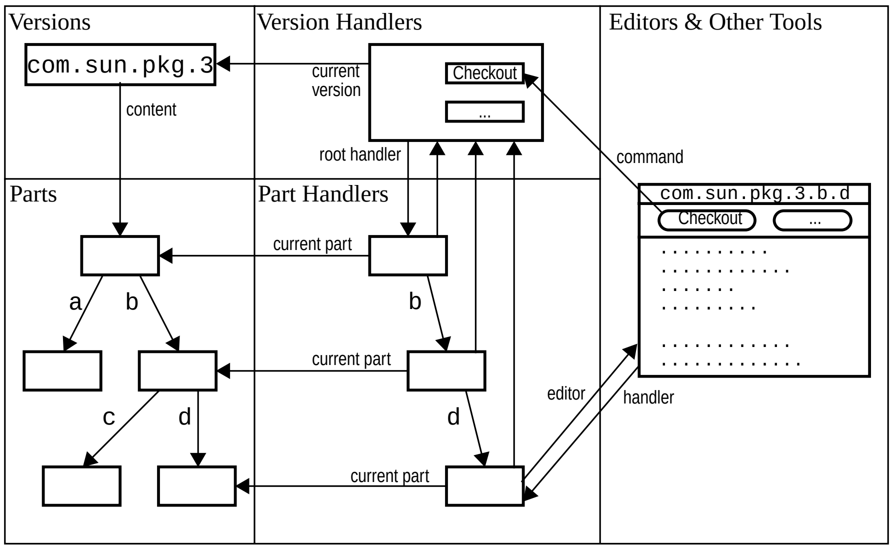

Fig. 1. Example Object Instance Relationships

Four methods in the `PartHandler` interface establish contextual relationships. The first creates a permanent relationship with an object of type `VersionHandler`:

```
void initialize(VersionHandler vh);
```

A version handler plays several, central roles. It:

- is the point of access into the JP versioned store for tools in the environment;
- provides access to its *current* package version;
- implements command objects, which use the versioning system to create new versions (e.g. Checkout, Advance);
- reassociates itself with any newly created version, thus tracking successive versions of the developer's work; and
- notifies its part handlers of version-related state changes.

These roles will become more clear in the following discussion. It is significant that versioning operations are not implemented in any part handler or editor. This is a direct consequence of orthogonality in both the versioning and storage systems. Developers can invoke versioning commands (for example the Checkout command in the figure) from any editor, where they might appear as buttons or menu items, but such command objects are opaque to the editors. Although not strictly required (a separate tool could make versioning commands available), direct access to the commands from every editor reduces distraction and eliminates confusion concerning which editors are associated with which versions.

The next method establishes the fundamental relationship between a handler and a part in the versioned store:

```
        boolean setPart(Mutability part);
```

This returns `false` if the part is not of the implementation type for which the handler is specialized (this is a system error). Although this method is normally invoked only during part handler creation, the part associated with a handler changes during the handler's lifetime as new parts are created during Advance operations. The part represents the version content a developer wants to see when browsing; when the user is editing, the current part represents the most recently versioned instance of the named part.

Each part handler implementation is responsible for creating an editor appropriate to the implementation type of its associated part (for simplicity we assume that all editors appear as top level windows). In a simple realization of the framework, this is done by a call to a method on the part handler:

```
    void edit();
```

An editor communicates only with its associated part handler via a comparably simple interface that can be private to the pair. In the most flexible realization of the framework, using the Model-View-Controller approach, the part handler holds the (possibly mutable) *data model* for the associated part, and editors are *views*. Other editor implementations may manage their own mutable buffers, into which the data content of the current part is initially copied. In any case, the division of responsibility between part handler and editor is private to the two implementation types, hidden by the `PartHandler` interface. Finally, a part handler is responsible for passing to its editor the set of command objects associated with its version handler. The editor can make these available to the developer, for example as buttons or menu items.

A fourth relationship is with handlers for parts that are children of the current part. Tools in the environment gain access to parts by first locating a version handler associated with the desired package version, and then requesting a part handler for each part of interest. A version handler supplies part handlers by recursive calls down the part handler hierarchy of the following method, based on the name of the part:

```
    PartHandler getPartHandler(Path name);
```

Typical implementations create handlers for subparts only when needed, as suggested in Figure 1 where not all potential handlers appear. The implementation types of subparts need not be known to a container or its part handler. Part handlers are created by a shared abstract factory object that embodies the desired binding (possibly configured by individual developers) between part types and handler types (which in turn implies editor types in the simple case).

The net effect of this collaboration is that each version of a package being accessed by tools in the environment is associated with a single version handler; the version handler manages a tree of part handlers which partially mirror the version's content. A developer can conveniently and safely view multiple versions of a package at the same time, *even in situations where a part is contained in multiple versions.* Such sharing is common, since unchanged parts are typically shared by successive versions (see below). Each version's handler group maintains context for viewing parts, which are immutable and independent of the contexts in which they are included.

3.2 Versioning

Versioning operations include Checkout, Advance, Checkin, and Branch. Each adds a new version to a JP package according to rules of the versioning model. Checkout, Checkin, and Branch operations create new versions with content identical to their predecessors, as in the example of Section 2.4.

A developer's changes are committed by the Advance command, implemented within the version handler. This command creates a new version, makes it the successor to the current version with a call to the versioning system, and then causes the new version to become version handler's current version. The Advance command obtains the content for a new version by calling the following method on the root part handler:

```
    Mutability advance();
```

This returns a part that represents the current "value" of the handler's data model. A newly created part also becomes the handler's current part. In cases where the developer has made no changes, then the handler returns its current part, without creating a new one. Parts can be shared safely among versions when they are designed to be independent of context.

A handler for a container part must implement this method recursively, for example advancing each child handler and creating a new container part if any of its children have produced new parts. This amounts to a distributed implementation of the simple path-copying approach to versioning large objects, for example as used for Exodus "storage objects" [4]. It is not required that handler/editor pairs follow this approach, but doing so effectively constrains storage consumption.

It is worth noting that all this takes places via interfaces that hide part representations, which are generally considered private to handler/editor implementations and possibly some privileged clients of them. This permits alternate implementations of similar parts. For example, editor implementations are free to represent versioned parts as deltas that refer to predecessor parts (whether or not all the predecessors were actually versioned directly), either in the name of compact storage or to implement editor-specific history mechanisms. Such history mechanisms are orthogonal to package-level history maintained in package version histories; history-guided undo can be supported at both levels, but always by advancing to newly created versions, not by removing versions.

This separation of concerns is greatly enhanced by the absence from this interface of any negotiation for persistent storage: a handler (or its editor) simply creates an object, which is then passed to the versioning system where it becomes permanent.

The handler framework itself supports a crude but familiar kind of undo. A Revert action recursively instructs a group of part handlers to discard changes made relative to the current version (i.e. since the most recent Advance).

```
    void revert();
```

3.3 Usability

Only those aspects described above are strictly required for the JP coordination framework to function correctly. For it to function effectively, however, attention must be paid to the look and feel of the system in operation. Four methods in the `PartHandler` interface are used to ensure smooth, unobtrusive interaction between editing and versioning.

A common, frustrating usability failure occurs in many environments when changes made in one or more edit buffers cannot be conveniently committed because of inappropriate versioning state. Each JP editor is made aware whether editing should be permitted.[1] Notification is propagated from version handler to part handlers by calls on the method:

```
void setEditable(boolean isEditable);
```

Editors display this status (ideally using a visual theme shared with other editors) and refuse to make changes when this permission has not been granted. For example, a developer viewing package version `com.sun.pkg.3`, as shown in Figure 1, would not be allowed to make changes without first clicking on the Checkout button, invoking the Checkout operation in the associated version handler. The version handler uses the versioning system to create a checkout branch and then notifies part handlers, which in turn notify editors that editing is permitted. At this point the version handler's current version changes to the newly created version (`3.checkout-mlvdv.0`, for example). Part handler and editors make no changes, however, because the content of the newly created version is identical to its predecessor.

Confusion arises from misunderstanding the status of uncommitted editing changes. In keeping with JP's versioning model, any change to a part is treated as a change to the whole version. Each version handler keeps a “dirty” bit and notifies its part handlers when it changes:

```
void setModified(boolean isModified);
```

Editors display this status, ideally using a visual theme shared with other editors. The coordination framework requires that any editor about to permit a change must check the dirty bit; when the dirty bit is off, the editor must notify its part handler, which in turn notifies the version handler, which then notifies all part handlers that the dirty bit has come on.

It can be confusing to have parts visible from more than one version, so each editors is made aware of the name for its current version. This name changes when versioning actions occur, as in the Checkout example mentioned above, following which the version handler notifies all part handlers with calls on the method:

```
void setVersionName(String name);
```

Editors display this name, for example in the window bar as shown in Figure 1.

Hierarchical naming of parts within a package is natural for source objects, although not strictly required. Each editor is made aware of the current name of its associated part. The name of a part can change, for example when the developer

1. In the current versioning model, permission to edit is defined by whether the current version is the latest on a checkout branch.

renames an enclosing folder, in which case the affected part handlers must notify handlers for child parts with calls on the method:

```
void setPartName(String name);
```

3.4 Coordination in Action: The Edit-Build Loop

An important consequence of this framework is the smoothness with which work gets done. Here is a summary of how it appears to a developer.

– A developer opens one or more editors on parts of a package version, for example `com.sun.pkg.3`. Parts might include both folders and source objects.
– The editors visually indicate that editing is not permitted and, of course, that the contents of the version are not "dirty".
– Deciding to make changes, the developer presses the Checkout button on any of the associated editors. Every editor shows the new version name (for example `com.sun.pkg.3.checkout-mlvdv.0`), and every editor indicates that editing is now permitted.
– The developer begins to edit. The first change in any editor causes all editors to show that the dirty bit has turned on.
– The developer decides to try a build by pressing a build button on any editor, which automatically invokes an Advance before the actual build. All editors show that the dirty bit has gone off and that there is a new version name (for example `com.sun.pkg.3.checkout-mlvdv.1`).
– When finished with the session, the developer presses the Checkin button and enters some version comments. The version name changes yet again (for example to `com.sun.pkg.4`), and editors show that editing is no longer permitted.

It is significant that the developer pays no attention to versioning, other than the decision to invoke Checkout and Checkin operations. At the same time, the progression through versions is visible in every editing context, and the developer benefits from a complete history of every build performed, along with the JP guarantee that all build are repeatable.

The developer may ask to view another version, for which a new version handler and associated editors would be created. These editors are managed separately, since they have distinct version handlers, even in situations where the same part might be shared between the two versions in view.

In practice developers often work concurrently on many packages, which in JP are independently versioned. Coordination among version handlers is an important issue that lies beyond the scope of this paper.

4 Related work

4.1 Vesta

The JP environment is based heavily on design principles from the Vesta project [14], which takes the position that configuration management, building, and storage management must be aligned for reliability and scalability. Building takes place in the store in both JP and Vesta; in JP, however, editing also takes place in the store, whereas it does not in Vesta. A Vesta Checkout operation copies the hierarchical content of a package version into a file system tree where it may be edited with conventional editors; a Vesta build requires creation of a new version by copying files back into the store.

Vesta makes no provision for coordination among file-based editors and versioning system, as there is in JP. Furthermore, Vesta package contents are limited by what can be represented as files (Vesta stored sources are byte arrays) and directories, in contrast to JP's flexible use of arbitrary objects.

4.2 POEM

The POEM environment [15], developed concurrently with the first JP prototype and with a similar strong influence from Vesta, shares many goals with JP.

Key implementation strategies differ, however. POEM keeps meta-data in an OODB, leaving storage of parts to a conventional file-based versioning system. In contrast, JP represents all data as persistent objects in a simple object-oriented versioning framework. As in most systems, POEM permits editing by creating mutable files; JP supports editing in the store through direct object interaction in the handler framework (although integration with separate editors can be supported by particular implementations of the `PartHandler` interface). Building in POEM is encapsulated as an operation within software units, whereas the JP approach separates the two more strongly: JP parts are treated as immutable values, and the JP builder is a single interpreter (with generic caching behavior) that computes over a space of values that includes those parts as well as derived objects.

4.3 Compound Document Editing

The JP approach to coordinated editing has much in common with compound document frameworks such as OpenDoc [7]. Compound documents can contain other documents whose types are unknown, the only requirement being adherence to a coordination framework by appropriate editors. JP departs most notably from compound document frameworks with its support for versioning (without requiring any version awareness in editor implementations).

Compound document frameworks deal with other issues as well. For example, parts of their protocols concern GUI-related resources (for example screen real estate and access to a shared menu bar), so that editors for embedded parts can be visually embedded within a containing editor; such support is absent in JP, but could in principle be added. Compound document protocols must also deal with the sticky problem of storage management, a problem solved transparently in JP through its reliance on orthogonally persistent objects.

4.4 COOP/Orm

JP's editor coordination has the most in common with work on fine-grained collaborative editing and version control by Magnusson et. al. [16][17]. Both start from the position that a software development environment must be focused on concurrent, collaborative, and distributed development; both projects place version and configuration management mechanisms at the heart of the respective systems.

Differences between the systems reflect different project emphasis:

– *Point of Departure*: JP starts with the requirement for reliable, scalable system building, whereas COOP/Orm suggests that software development is an important special case of collaborative document development.
– *Versioning*: COOP/Orm emphasizes fine-grained versioning, whereas versioning in JP is defined to coincide with the granularity of building (language packages).
– *Editing*: COOP/Orm emphasizes fine-grained collaborative editing, whereas JP approximates file-granularity editing (finer grained editing is a JP goal) with collaboration permitted only by concurrent package checkouts.
– *Distribution*: COOP/Orm emphasizes distributed, collaborative editing, whereas JP emphasizes reliable, distributed building.
– *Editor Integration*: COOP/Orm editors must be strongly versioning-aware and participate in a collective representation scheme, where JP editors have no such requirements. JP editors are free to represent the parts they manage in any way. JP storage is made straightforward by a combination of orthogonal persistence and orthogonally versioned store objects.

None of these differences appear to reflect fundamental incompatibilities between the COOP/Orm and JP approaches.

4.5 ClearCase

Although there is no direct counterpart in ClearCase, a commercial Configuration Management product [5], to the editor coordination framework that is the focus of this paper, the case study of ClearCase by Asklund and Magnusson [1] invites a broader comparison. The dominant distinction is in fundamental technology: ClearCase is designed to work within the semantics of conventional file systems and the *make* [8] program for system building.

The JP approach abandons both of these in favor of technologies believed to be fundamentally more reliable and scalable: persistent object storage combined with Vesta-style functional programming for building. This shift potentially addresses most shortcomings mentioned in the case study, by making needed mechanisms either more reliable, easier to implement, or unnecessary. Specific limitations addressed by the current, limited prototype of the JP approach include:

– Support for versioned sub-systems;
– “Light-weight” branch types;
– Fine-grained “micro-versions”; and
– A more powerful support for configurations than “labels”.

5 Project Status

The architectural principles behind the JP environment were first explored in a prototype implemented in C++ using an object-oriented database [12][13]; that first prototype included editor coordination as described here. The architectural complexity (and accompanying fragility) added by the OODB, although less severe than in file-based implementations, drove us to seek truly orthogonal persistence.

A second, file-based prototype has been in daily use for several years, supporting both its own development and that of its successor. Editor coordination is missing in this second prototype, and it is sorely missed.

A third prototype is being constructed in Java using PJama [3] for object storage, an implementation of orthogonal persistence [2] for Java. PJama is being developed through a collaboration between the Forest Project at Sun Microsystems Laboratories and the Persistence and Distribution Group at the University of Glasgow. The third JP prototype is dedicated specifically to building large systems in Java [11], and the editor coordination scheme reported here is at its core.

Our immediate goal is to put into daily use the PJama-based JP prototype. On that platform we intend to push source editing in several directions:

– Finer granularity: decompose Java source objects into smaller language-oriented parts, for example as is done in COOP/Orm [17].
– Higher-level sources: build or import tools that create higher-level source objects from which Java sources are generated through building. Examples include GUI builders and OO modeling tools. The main difficulty here is adapting file-centric applications to the much simpler “objects-in, objects-out” interface.
– Add a layer for software process tracking and management.
– Add a more rich view architecture, for example supporting editing in place in the style of compound documents.
– Language-based editing: make available to editors the results of recent builds, so that suitably equipped editors can exploit language information to drive high-productivity editing [18].

6 Conclusions and Further Work

Our very limited experience with coordinated editing suggests this approach does indeed enable a smooth edit-build loop for developers, with absolutely minimal need for version awareness until such time as a versioning-specific action such as Checkin is desired. Lack of editor coordination in the second prototype has been both a constant irritation and reminder of its success in the first prototype. We are confident that this framework will be a key component in our efforts to make convenient an otherwise inconvenient environment that creates a completely new versioned configuration with every build.

Implementation experience so far confirms our intuition that the need to deal with issues of storage and versioning complicates the construction of editors tremendously; the overhead of creating new editors for particular objects is sufficiently low in JP that it makes sense to construct an editor in this framework for tasks as simple as setting a boolean value (the first prototype indeed had such an editor).

We will be able to evaluate and validate this framework more thoroughly as we pursue the editing-related goals mentioned in the previous section.

7 Acknowledgments

This work benefits greatly from the vision of Mick Jordan, Principal Investigator of the Forest Project at Sun Microsystems Laboratories. The delightful opportunity to use an increasingly practical, highly orthogonal persistent object system owes much to Prof. Malcolm Atkinson and his band of persistent researchers in Glasgow. Helpful comments on drafts of this paper were contributed by Huw Evans, Tobias Murer, Yuval Peduel, and anonymous reviewers.

8 Trademarks

References

1. Asklund, U, Magnusson, B.: A Case-Study of Configuration Management with ClearCase in an Industrial Environment. In: Conradi, R. (ed.): Software Configuration Management, Proceedings of the ICSE '97 SCM-7 Workshop, Boston. Lecture Notes in Computer Science, Vol. 1235. Springer-Verlag, Berlin Heidelberg New York (1997) 201-221
2. Atkinson, M., Morrison, R: Orthogonally Persistent Object Systems. VLDB Journal **4 (**1995)
3. Atkinson, M., Daynès, L., Jordan, M., Printezis, T., Spence, S.: An Orthogonally Persistent Java. In: ACM SIGMOD Record **25** (1996) 68-75
4. Carey, M., DeWitt, D., Richardson, J., Shekita, E., Lochovsky, F.: Storage Management for Objects in EXODUS. In: Kim, W. (ed.): Object-Oriented Concepts, Databases, and Applications. Addison Wesley, Reading, Massachusetts (1989) 341-369
5. ClearCase Concepts Manual. Atria Software (1992). See also `http://www.rational.com/products/clearcase/`
6. Conradi, R., Westfechtel, B.: Towards a Uniform Version Model for Software Configuration Management. In: Conradi, R. (ed.): Software Configuration Management, Proceedings of the ICSE '97 SCM-7 Workshop, Boston. Lecture Notes in Computer Science, Vol. 1235. Springer-Verlag, Berlin Heidelberg New York (1997) 1-17
7. Feiler, J., Meadow, A.: Essential OpenDoc. Addison Wesley, Reading, Massachusetts (1996)
8. Feldman, S.: Make -- A Program for Maintaining Computer Programs. Software--Practice & Experience **9** (1979) 255-265
9. Gosling, J., Joy, W., Steele, G.: The Java Language Specification. Addison-Wesley (1996)
10. Jordan, M., Van De Vanter, M.: Large Scale Software Development in Java. (In Preparation) Sun Microsystems Laboratories Technical Report (1998)

11. 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, Cottbus, Germany (1997) 155-163

12. Jordan, M., Van De Vanter, M.: Software Configuration Management in an Object-Oriented Database. In: USENIX Conference on Object-Oriented Technologies (COOTS), Monterey, CA, June 26-29 (1995)

13. Lamb, C., Orenstein, J., Weinreb, D.: The ObjectStore Database System. Communications of the ACM **4** (1991) 50-63

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

15. Lin, Y., Reiss, S.: Configuration Management in Terms of Modules. In: Estublier, J. (ed.): Software Configuration Management. Lecture Notes in Computer Science, Vol. 1005. Springer-Verlag, Berlin Heidelberg New York (1995) 101-117

16. Magnusson, B., Asklund, U., Minör, S.: Fine-Grained Revision Control for Collaborative Software Development. In: Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering, Los Angeles, California (1993) 33-41

17. Magnusson. B.: Fine-Grained Version Control in COOP/Orm. In: Workshop on Version Control in CSCW Applications at the European Conference on Computer Supported Cooperative Work, Stockholm (1995)

18. Van De Vanter, M.: Practical Language-Based Editing for Software Engineers. In: Taylor, R., Coutaz, J. (Eds.): *S*oftware Engineering and Human-Computer Interaction. Lecture Notes in Computer Science, Vol. 896. Springer-Verlag, Berlin Heidelberg New York (1995)