Sun Microsystems Laboratories

SML 98-0485

Versioned Java™ Packages in the JP Programming Environment

Michael L. Van De Vanter

The Forest Project

http://www.sunlabs.com/research/forest/

Bay Area Round Table, 11 September 1998

Outline

- **Background: Forest Project & Large Scale SW Development**
- **Versioned Java™ Packages in JP**
  - **Modularity & Versioning**
  - **Namespace**
  - **Federated Repositories**
  - **Configuration Management**
  - **Building**
  - **Tool Integration**
  - **Persistent Object Storage**
- **Implementation Example**
- **Project Status**

Background: The Forest Project

- **Mick Jordan, Principal Investigator**
- **JP Programming Environment**
  - **scalable development model for Java programming**
  - **enhance code reuse by JP-compliant environments**
  - **“write once, build anywhere, run anywhere”**
- **Orthogonal Persistence for Java (OPJ)**
  - **extend Java programming model to storage**
  - **make object lifetime independent of type, code, etc.**
  - **“write once, run anywhere” for data-centric applications**

Background: Challenge of Large Scale Software Development

- **Requirements**
  - **team development – isolation, collaboration**
  - **geographic separation – inter-project, intra-project**
  - **reliable, repeatable, system building**
  - **finding and reusing software components**
  - **extensible toolsets**
- **Solutions**
  - **versioning, configuration management**
  - **file systems, databases, object stores**
  - **builders, build languages, e.g. *make***
  - **naming systems**
  - **software distribution**
  - **tool integration frameworks**

Background: Problem with Current Approaches

Some current systems:

- **Teamware, Make**
- **ClearCase, ClearMake**
- **Continuus/CM, ObjectMake**

The essential problems:

- **solutions are stuck in an inertial time-warp (1970's)**
- **lowest common denominator**
- **denial that system building is programming**
- **inherent scaling problems**
- **no abstraction**
- **most of the current focus is on single-user IDEs**
  - **IBM's Visual Age is a notable exception**

Background: A Better Approach – JP

- **JP is based on the Vesta Approach**
  - **http://www.research.digital.com/SRC/vesta**
- **Vesta provides:**
  - **versioned directories (packages) of flat files**
  - **integrated versioning, configuration management and building**
  - **functional system building language**
  - **scalability through modularity**
- **JP modifies and extends Vesta with:**
  - **persistent object store**
  - **integrated framework for a growing collection of tools**
    - **authoring tools (editors), browsers, analyzers**
  - **a special focus on the Java language**
  - **using the Java language for system building**

Modularity & Versioning: Goals

- **Modular system building is crucial**
  - **it must scale to very large systems**
  - **it must be compatible with current practice**
- **The Java *package mechanism* is a good start**
  - **most language properties are desirable**
  - **the rest can be supplied by an Integrated Development Environment**
- **Complexity must be reduced**
  - **abstract over inessential differences**
  - **automate much of what is now done manually**
- **Reliability must be increased**
  - **builds must be repeatable**
  - **information derived by tools must not be lost**

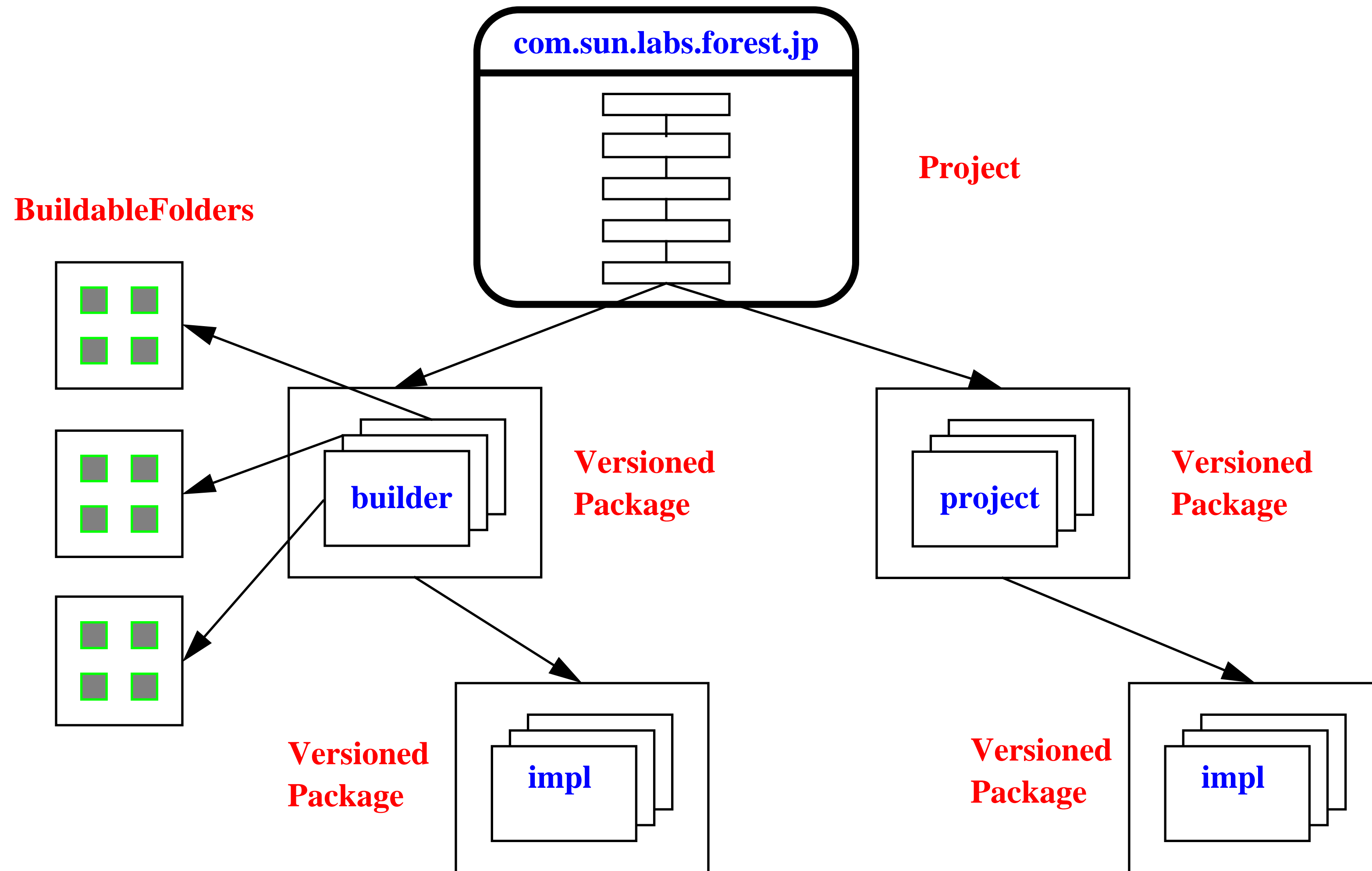
Modularity & Versioning: Versioned Java Packages

- **A versioned package (in the style of Vesta)**
  - **is a conventional package in the Java package namespace**
  - **is a first-class entity in the JP development environment**
  - **contains a *build script*: a parameterized program for building it**
  - **uses other package versions via `import` statements in the build script**
- **This concept simplifies development by unifying:**
  - **modular system construction**
  - **storage management**
  - **system building**
  - **configuration management**

Modularity & Versioning: Java Packages (cont.)

- **The Java package mechanism offers some useful features**
  - **systematic, (potentially) global name management**
  - **access control within the language**
- **The Java package mechanism is weak for system structuring**
  - **name discipline not enforced**
  - **a package is not first-class (you can't import a package, only its classes)**
  - **a package has no other semantics for compilation or runtime class loading**
- **Versioned package are first-class entities in JP**
  - **The unit of versioning/configurations, storage, and building**
  - **The unit of code sharing and replication across sites**

Modularity & Versioning: Example of JP Package Structure



Modularity & Versioning: Evolution of a JP Versioned Package

mjj_checkout
test
main
mjj_checkout

0 1 5
checkin
checkout
0 1
merge
branch
5 6 7
checkout
checkin
0 1 8

edit
advance

Naming: Versioned Packages and Naming Systems

- **Current tools require management of too many names**
  - **language entities**
  - **files and directories**
  - **build modules**
  - **versions and configurations**
- **Java packages *align* language (class) and storage (file) names**
- **JP packages unify them all**

Naming: Example – Versioned Java Package Names

- **A versioned package**

```
com.sun.labs.jvb.util
```

- **A package version**

```
com.sun.labs.jvb.util.7
```

- **Source code for a class**

```
com.sun.labs.jvb.util.7.Stack
```

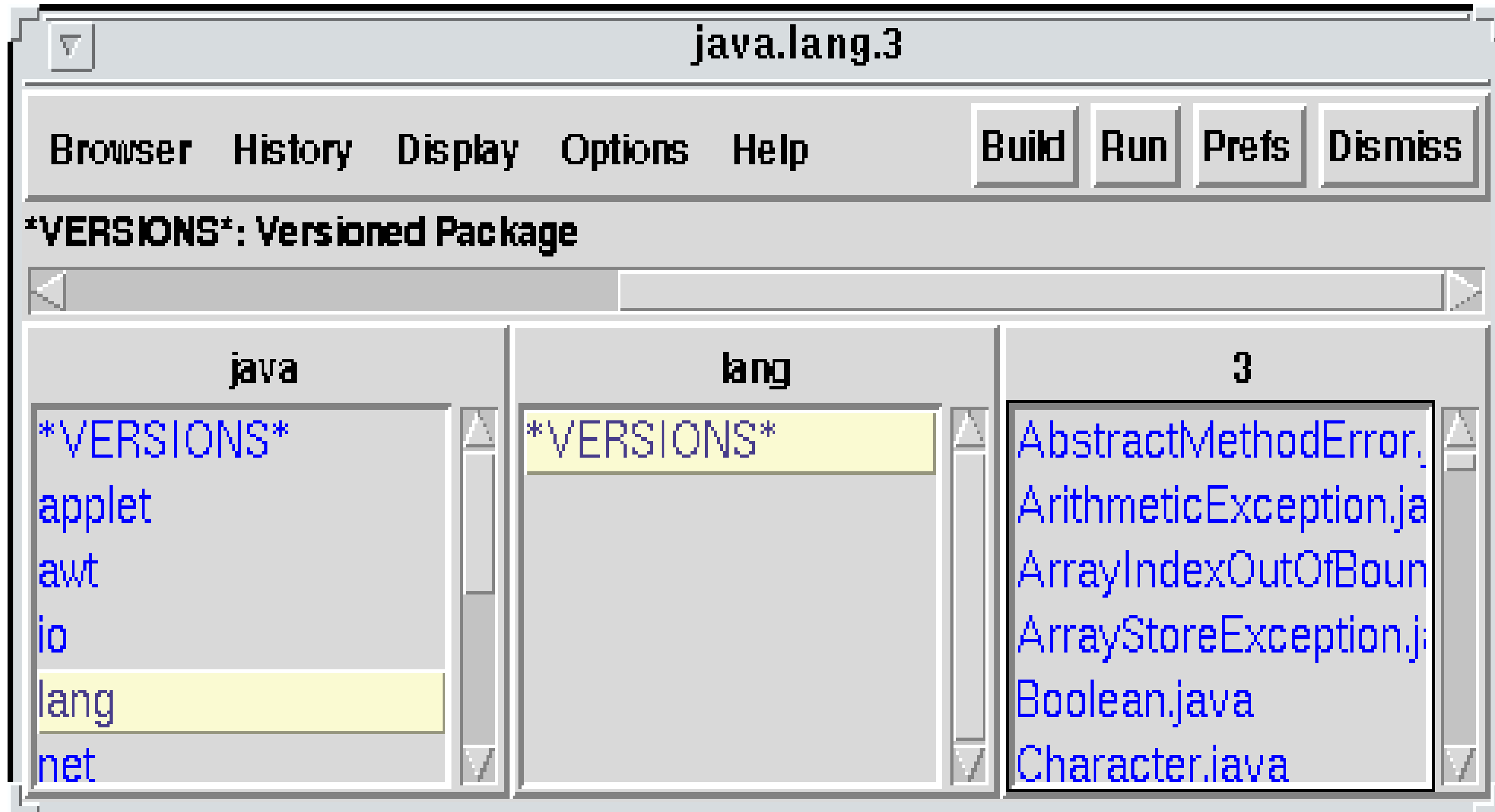
- **Using the class in Java source code**

```
import com.sun.labs.jvb.util.Stack
```

- **Using the package version in a JP build script**

```
import com.sun.labs.jvb.util.7
```

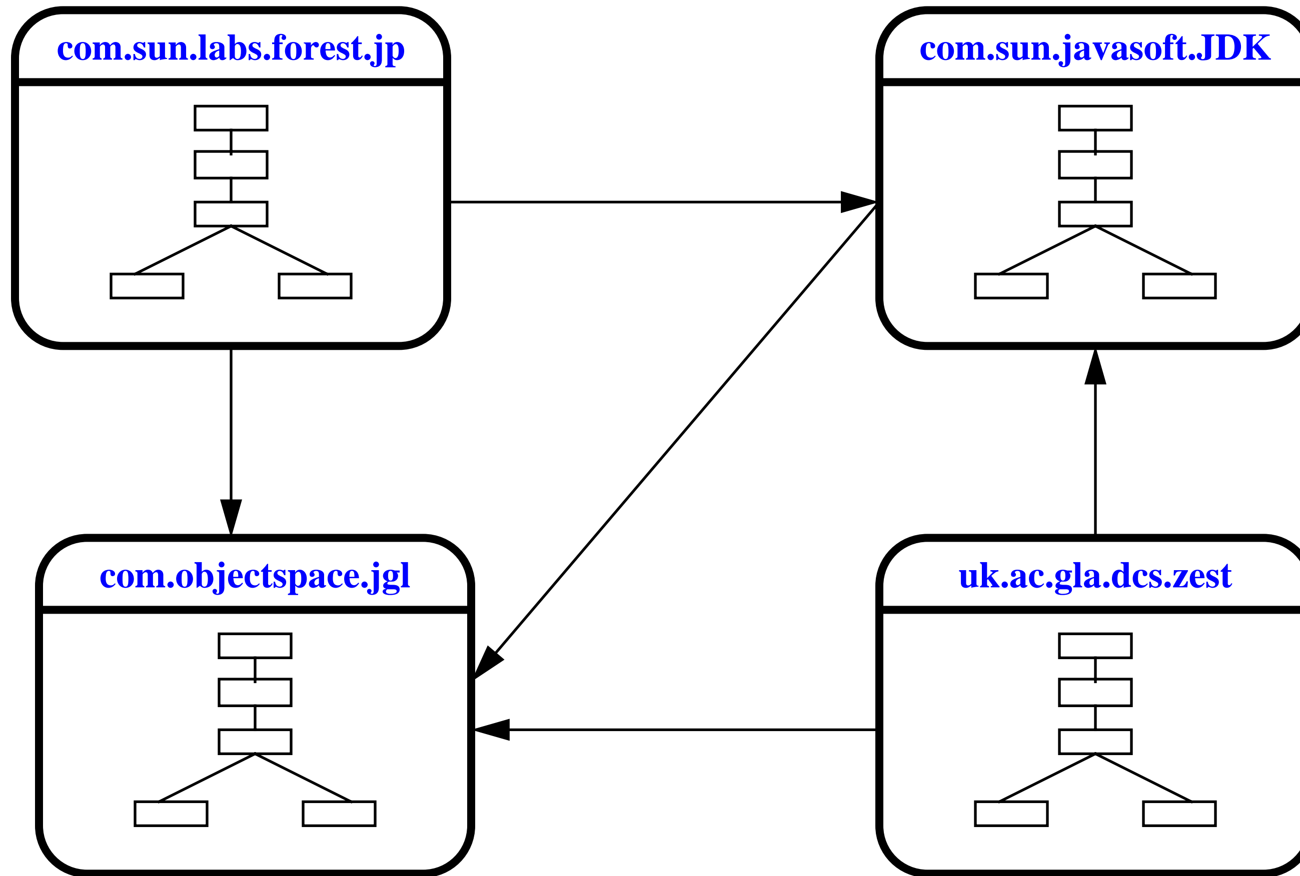
Naming: Example – Abstract Naming in JP (JP-lite browser)



Naming: Versioned Packages and Global Names

- **Global, location-independent, package names are important**
  - **replication**
  - **build anywhere**
- **Java names are potentially global**
  - **DNS-based names are officially encouraged, e.g.** `com.sun.labs.jvb.util`
  - **global naming is not always followed**
- **Federation of Repositories (Projects)**
  - **each project owns a portion of the logical namespace**

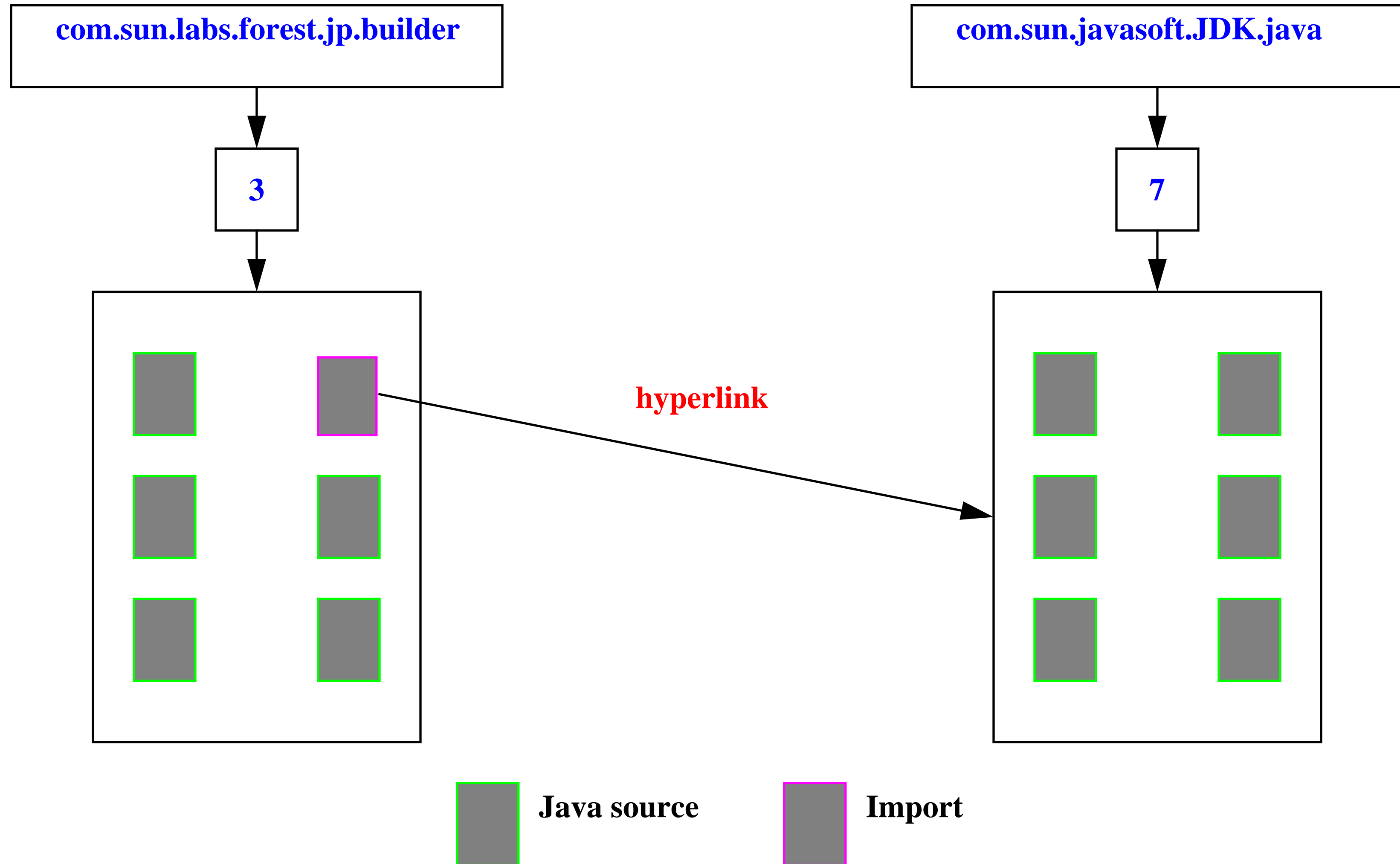
Naming: Example – Federated JP Repositories



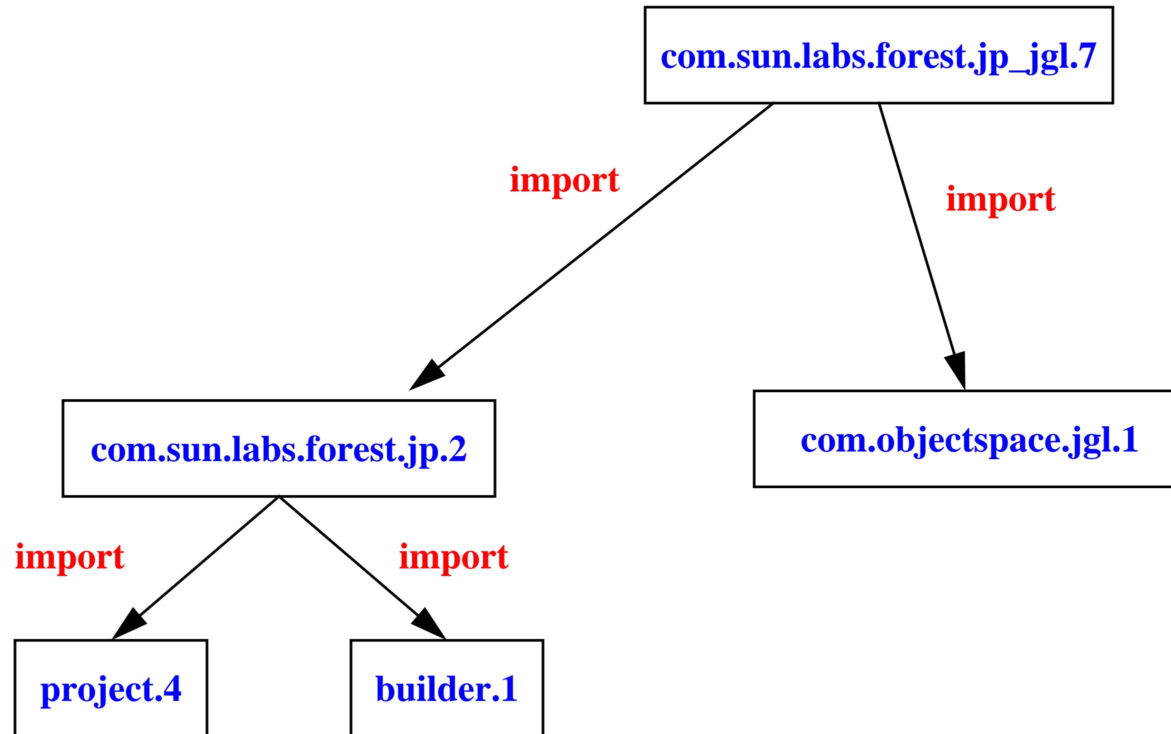
CM: Versioned Packages and Configuration Management

- ***umbrella*** **packages are used for aggregation**
  - **use other package versions via `import` in the build script**
  - **the build script `import` always refers to a precise, immutable version**
- **umbrellas typically contain no source code directly**
- **umbrellas are versioned**
- **umbrellas may recursively describe arbitrarily large systems**

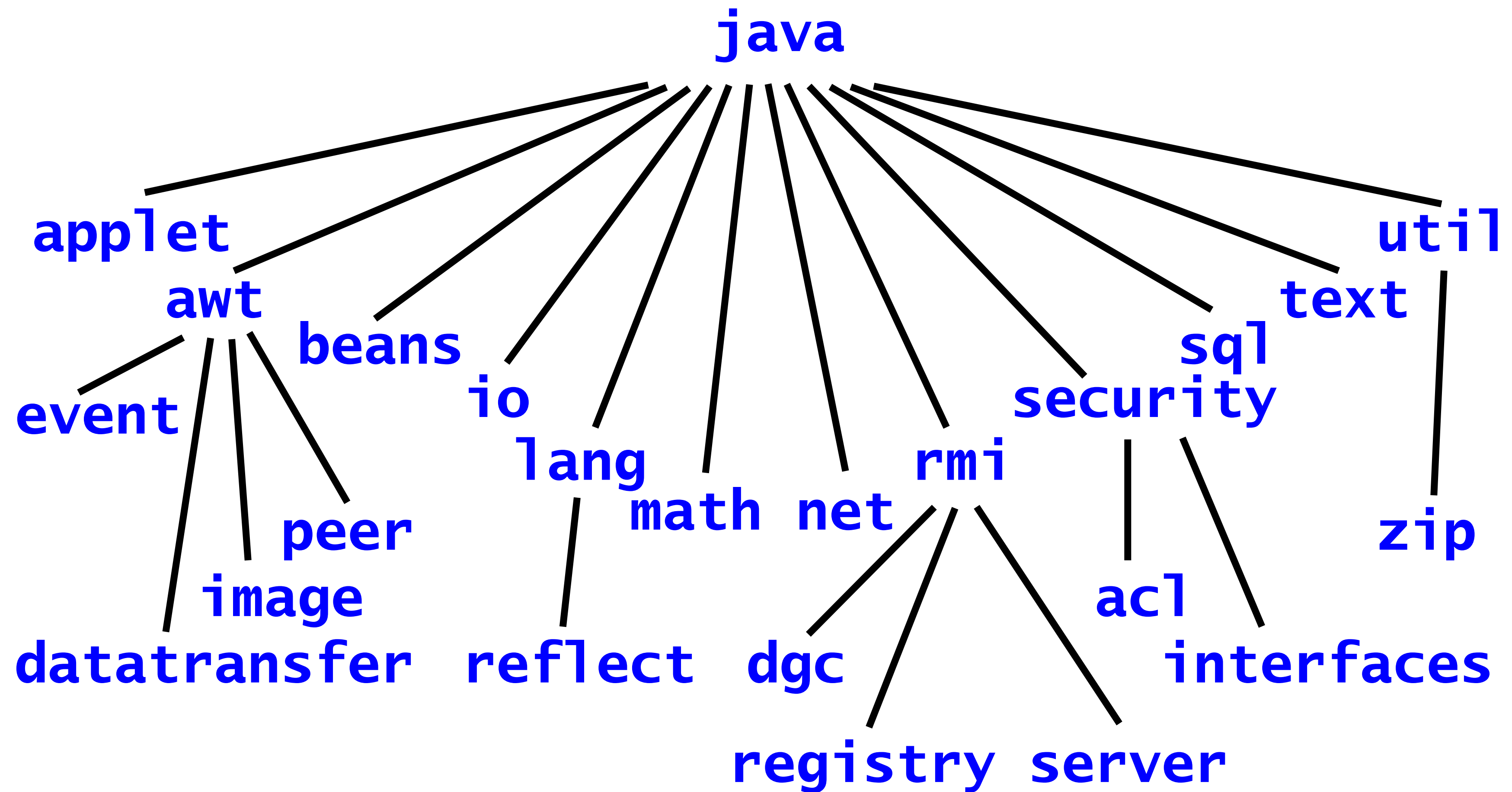
CM: Importing Packages in JP



CM: Aggregating JP Packages by Importing



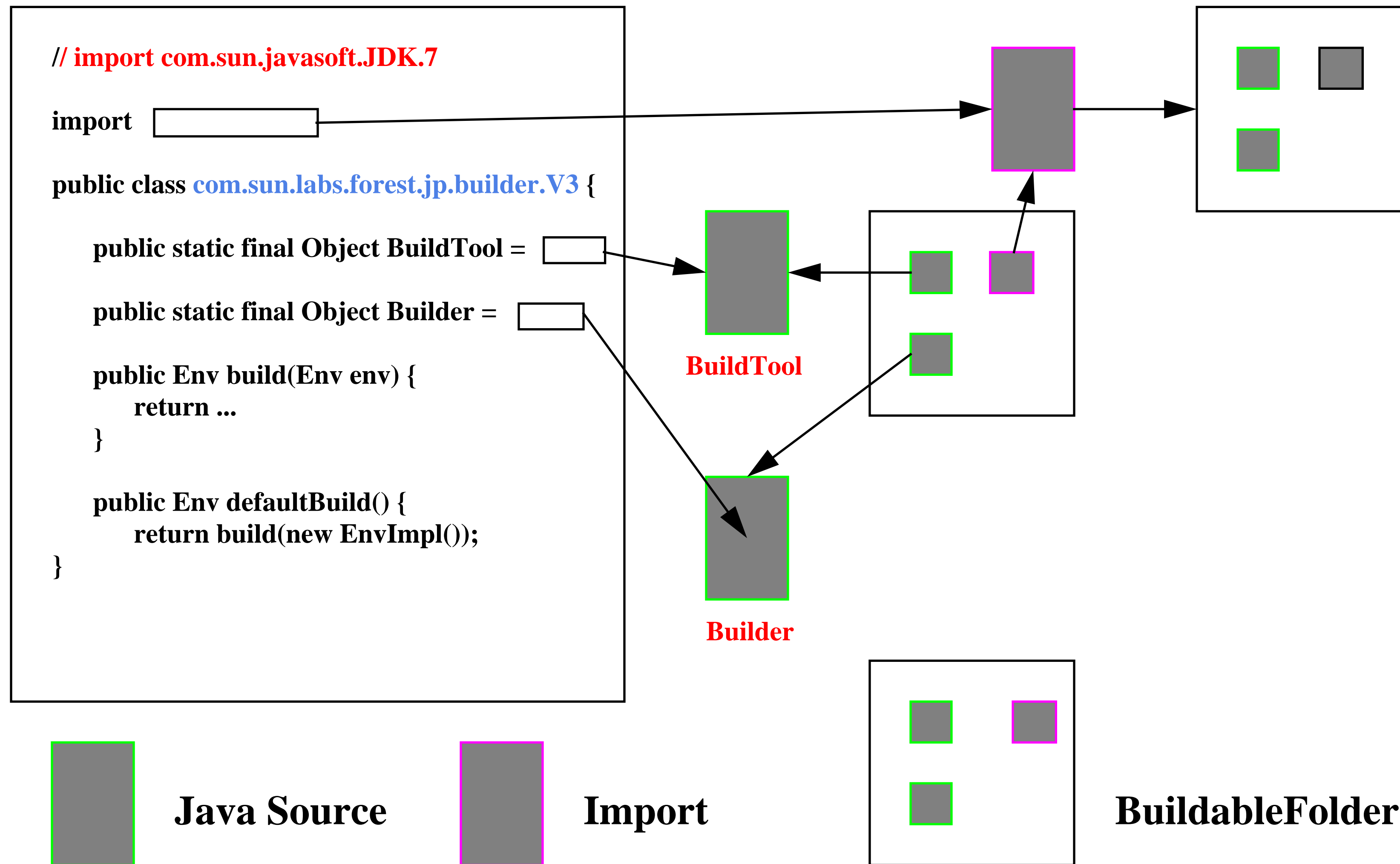
CM: JDK 1.1 Core Class Package Hierarchy



Building: JP Build Scripts are Functional Programs

- **Build scripts are Vesta-style mostly functional programs**
  - **compute over domain of source objects, class files, and other derived objects**
  - **objects in, objects out**
  - ***stability* by default – builds are *repeatable***
- **The build script interpreter is responsible for build avoidance**
  - **caches (memoizes) function evaluations for minimal rebuild after changes**
  - **cache by *fingerprint*: a highly reliable abstract value**
- **Derived objects are unnamed and invisible unless needed**
- **Each build invocation creates a *package object*:**
  - **describes completely and precisely the sources and resources used**
  - **describes all results**
  - **available for use by tools in the environment**

Building: JP Build Scripts are Hyper-Programs



Building: Making Versioned Java Packages Reusable

- **Package names are location-independent**
  - **assume global naming**
  - **check values abstractly via fingerprints**
- **Build scripts completely describe what must be built**
  - **libraries**
  - **compiler (in a versioned package, of course)**
  - **supports build-anywhere reliably**
- **Build script `import` statements in JP are *abstract***
  - **package versions may stored locally, remotely, or mirrored**
  - **package versions may be available in source, binary, or binary-on-demand**

Tool Integration: Java-based Tools

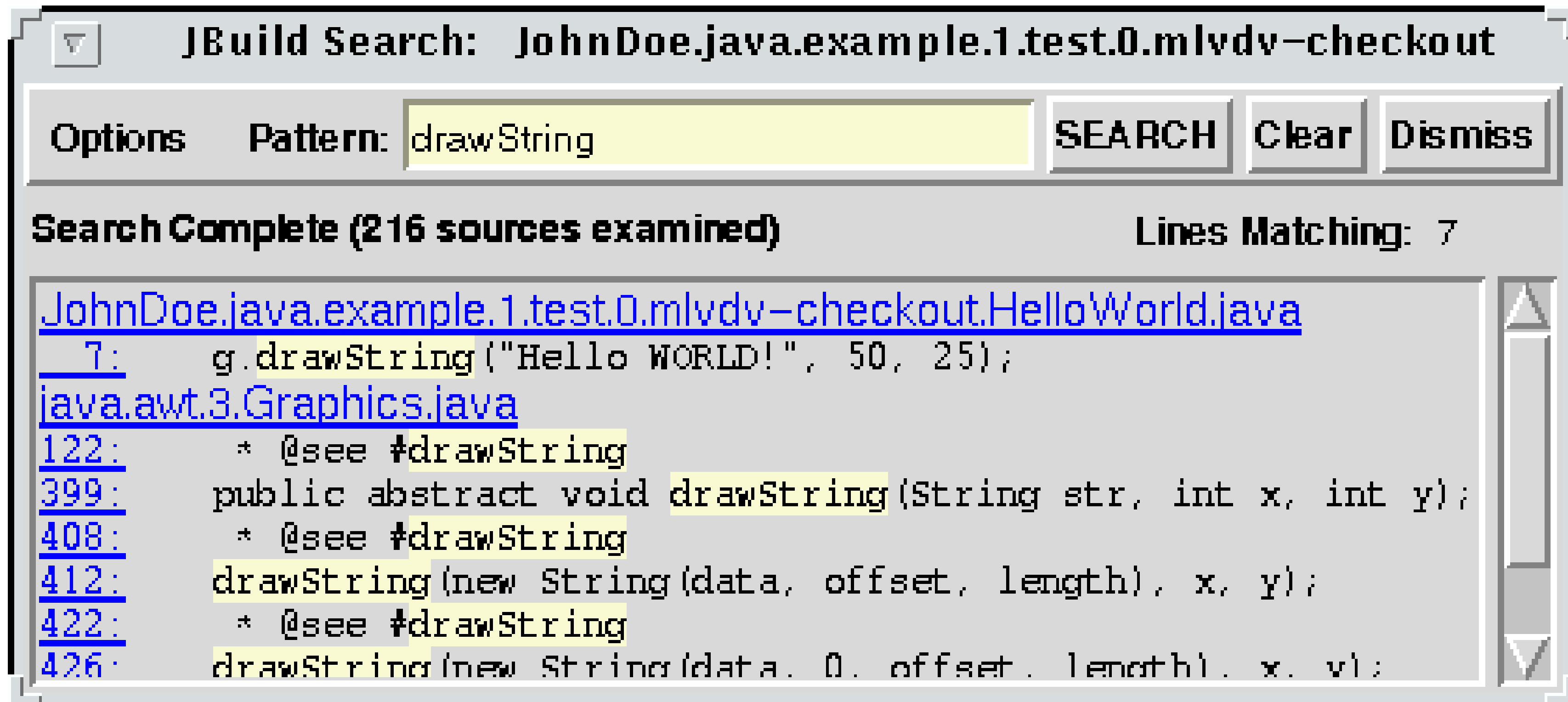
- **Tool interfaces simplified**
  - **objects in, objects out**
  - **need only handle application-specific logical naming**
  - **uncluttered by storage management**
- **Java interfaces to all subsystems**
  - **Build results (package object)**
  - **Versioned Package Repository (coordinated source editing)**
  - **Configurations (browsers and analyzers)**

Tool Integration: Example – The JP Version Interface

```
interface Version extends Ancestry, Container, Naming,
    TimeStamped {
  Version checkout(String name, boolean reserve);
  Version advance(Mutability obj);
  Version branch(String name);
  Version checkin();
  Mutability getObject();
  Version addComment(Object comment)

  // other methods proving navigation, reflection, e.g.
  Version getMainBranch();
  String getName();
```

Tool Integration: Example – Build-directed Search (JP-lite)



Storage: Persistent Objects implemented by PJama

- **An experimental Java platform – modified virtual machine**
  - **provides *orthogonal persistence* for Java objects**
- **Collaborative research**
  - **Persistence and Distribution Group at University of Glasgow, Scotland**
- **A practical technology**
  - **extremely simple API – one extra class**
  - **robust – atomic transactions**
  - **scalable – store sizes up to 2Gb now and larger on the way**
  - **incremental – operates with an object cache as small as 6Mb**
- **The next step on the Java journey**
  - **extend the Java object model to persistent storage**
  - **makes objects *real***

Implementation: A Tale of Three JP Implementations

1. C++ and ObjectStore

- **"Software Configuration Management in an Object-Oriented Database", COOTS95.**

2. Tcl/Tk and the Unix file system

- **JP-lite, used internally by group members**
- **supported own development**
- **supporting JP on PJama implementation**

3. PJama

- **100% pure Java implementation – in progress**

Implementing the Version Interface: an Instructive Example

Design Issues

- Object Storage Mechanism
- Concurrency Control
- Performance
- Scalability
- Portability
- Ease of Implementation

Version: Simple File System Implementation

- **Design choices**
  - **map each version node to a directory**
  - **keep additional state in a hidden (dot) file**
  - **lock entire tree when mutating**
- **Problems**
  - **file systems have limited functionality and vary**
    - **e.g. no links to directories, need a portability layer,**
    - **directory scanning and open file calls are expensive**
  - **programming tedious**
    - **pathnames as object references, lots of pathname (string) manipulations**
  - **manual (disk) storage management**
  - **implementation is exposed**
    - **easy to accidently violate system invariants**

Version: More Complex File System Implementation

- **Design Choices**
  - **DIY object storage**
    - **define and implement a file format**
    - **manually map objects to byte sequences**
    - **keep leaf objects as files?**
- **Problems**
  - **tedious, error-prone, manual programming**
  - **concurrency control more difficult**
  - **difficult to evolve the design**
  - **must handle caching explicitly for scalability**
  - **manual disk storage management even harder**

Version: Does Object Serialization Help?

- **Virtues**
  - **easy to use**
  - **can handle complex, fine-grained, object structures**
  - **simple concurrency control, atomic update model**
- **Problems**
  - **inherently not incremental**
    - **"big inhale/exhale" too expensive**
  - **so data must be partitioned**
    - **have to worry about (too much) reachability**
    - **re-introduces concurrency control, atomic update problems**
  - **types that do not implement Serializable**
  - **does not handle static fields (lack of orthogonality)**

Version Implementation: PJama

- **No extra programming**
  - **focus on the application logic**
  - **shortens the development time**
  - **improves reliability**
- **Full power of the Java language available**
  - **arbitrary data structures**
  - **fine-grain concurrency control**
- **Scalability**
  - **transparent, incremental, data transfer**
  - **object caching implicit**
- **(Disk) Garbage Collection**
  - **normal Java language rules**

Project Status

- **Interim version (JP-lite)**
  - **in use for 2+ years**
  - **supported its own development**
  - **experience has been encouraging**
- **Next version (JP on PJama)**
  - **development in progress**
  - **supported by JP-lite**
  - **storage via orthogonal persistence for Java (OPJ) – no files**
  - **add mechanisms for distributed development**
  - **explore collaboration models**

More Information

- **http://www.sunlabs.com/research/forest/**
  - **Research Papers (COOTS, SEE, SCM)**
  - **Slides from public presentations**
  - **Downloadable implementation of PJama (for research & evaluation)**