

*Software Configuration Management in  
an Object Oriented Database*

*Mick Jordan*

*June 1995*

# *Introduction*

---

- **Background**

- software development environment (SDE) for large systems
- multiple programmer teams
- heterogeneous language/host/target environments

- **Characteristics**

- change is the rule - stability is an illusion
- derived information explosion
- demand for *global* information

- **Current practice**

- weak configuration management
- old tools stretched beyond their original design limits
- new tools limited by inadequate infrastructure
- ad hoc “databases”

## *Outline*

---

- **Why a Database?**
- **ObjectStore®**
- **Overview of Forest**
- **Database Storage Management**
- **Performance Measurements**
- **Conclusions**

ObjectStore and Object Design Inc. are registered trademarks of Object Design Inc.

## *Why a Database?*

---

- **SDE data characteristics**
  - small atomic objects, large structured objects
  - highly interconnected
- **File system approach**
  - directories for large scale structure
  - file-formats or *pickling* for small-scale structure
  - file locking for concurrency control
  - ad hoc techniques for transactions
  - => difficult programming interface, problems of scale
- **Object-oriented database (OODB) approach**
  - objects for everything
  - transactional store
  - concurrency control
  - designed to scale

# *ObjectStore*

---

- **Persistent C++**
  - persistence independent of type
  - objects allocated by overloaded variants of operator `new`
  - access to data takes place inside a transaction, using C++ pointers
  - persistent memory mapped into virtual memory transparently
  - use references (smart pointers) to point at data between transactions
- **Client-Server model**
  - single server manages databases on one machine
  - clients communicate with the server to access data
    - most processing takes place in the client
    - client-side caching used to reduce communication overhead
    - clients can access multiple databases on multiple servers
- **Other features**
  - querying, collections libraries, SQL gateway

## *Overview of Forest*

---

- **Forest integrates:**
  - configuration management
  - editing
  - system building
- **Forest adopts the *Vesta* approach**
  - A repository of immutable components divided into two groups
    - *source* and *derived*.
  - A modular organization of systems into versioned *packages*.
  - A system builder that works by evaluating a *system model* expressed as a functional program.
- **Software development is organized around *projects* and *components***

## *Projects*

---

- **support a group of collaborating programmers**
- **contain structured collections of components**
- **provide efficient sharing of components**
- **may refer to other projects via *link* components**
- **correspond one to one with an ObjectStore database**

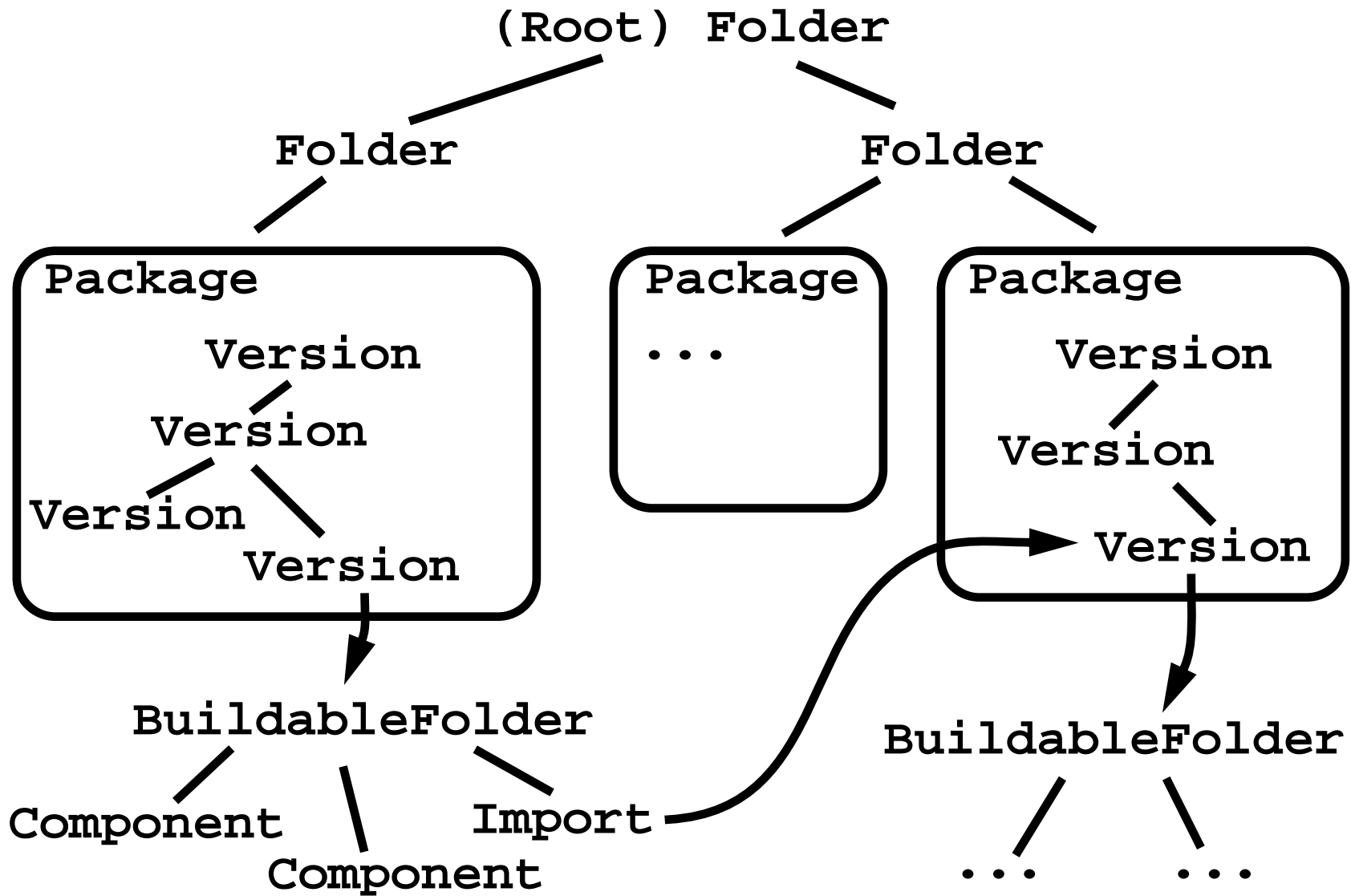
## *Components*

---

- **All objects in a project are subtypes of the abstract base class **Component**.**
- **Component characteristics**
  - abstract
  - uniquely identified (128 bits)
  - mostly immutable
  - reference counted
  - abstract value (identity - represented with a *fingerprint*)
- **Examples**
  - **Folder, Package, Version, BuildableFolder, Link, Import**
  - **Map, Sequence, Set**
  - **Text, TokenSequence, AbstractSyntaxTree**
  - **Library, Program, System**

# Project Structure

---



## *BuildableFolders*

---

- A **BuildableFolder** supports precise configuration management and system building

It can be thought of in three ways:

- A *folder* of source components
  - like a file system directory
- A *compound document*
  - the unit of versioning
- A *program module*
  - evaluated to build derived components

## *Storage Management*

---

- **ObjectStore mechanisms**

- *segments and clusters*
- overloaded operator `new`, placement syntax
- location enquiry functions

- **Forest mechanisms**

- abstract placement data into `DBNewPlace` class
- `Component` class defines a method that returns its `DBNewPlace`

- **Forest policies**

- minimize conflict - maximize concurrency => short transactions
- use `ObjectStore` references to hold results of navigation
- default allocation with same placement as containing component
- separate segment for each package
- separate segment for the derived components

## *Controlling Database Size*

---

- **How much to store persistently?**
  - derived information dominates storage
  - immutability and precise configuration management permit caching and sharing
- **Separation of interface and implementation**
  - Components accessed through abstract interfaces
    - no implementation details revealed
    - multiple implementations possible
    - trade-off persistent/transient data
- **Example: *Annotated Abstract Syntax Tree***
  - **Minimum persistent storage:**
    - source component it was generated from
    - compiler used to generate it
    - ASTs of imported modules
    - target environment (machine, compiler options etc.)

## *Garbage Collection*

---

- **Sharing and concurrency demand garbage collection**
  - **ObjectStore does not provide it**
  - **Components support it by reference counts**
- **Garbage arises in three ways:**
  - **temporary components generated during system building**
  - **derived components can be discarded at any time**
    - **because they can be regenerated reliably at will**
  - **source components can be discarded by removing old package versions**
    - **probably unnecessary, little impact on overall space requirements**
- **Distributed garbage collection**
  - **LinkComponent denotes an inter-database references**
  - **component UID identifies the database**
    - **=> can collect a group of connected databases**

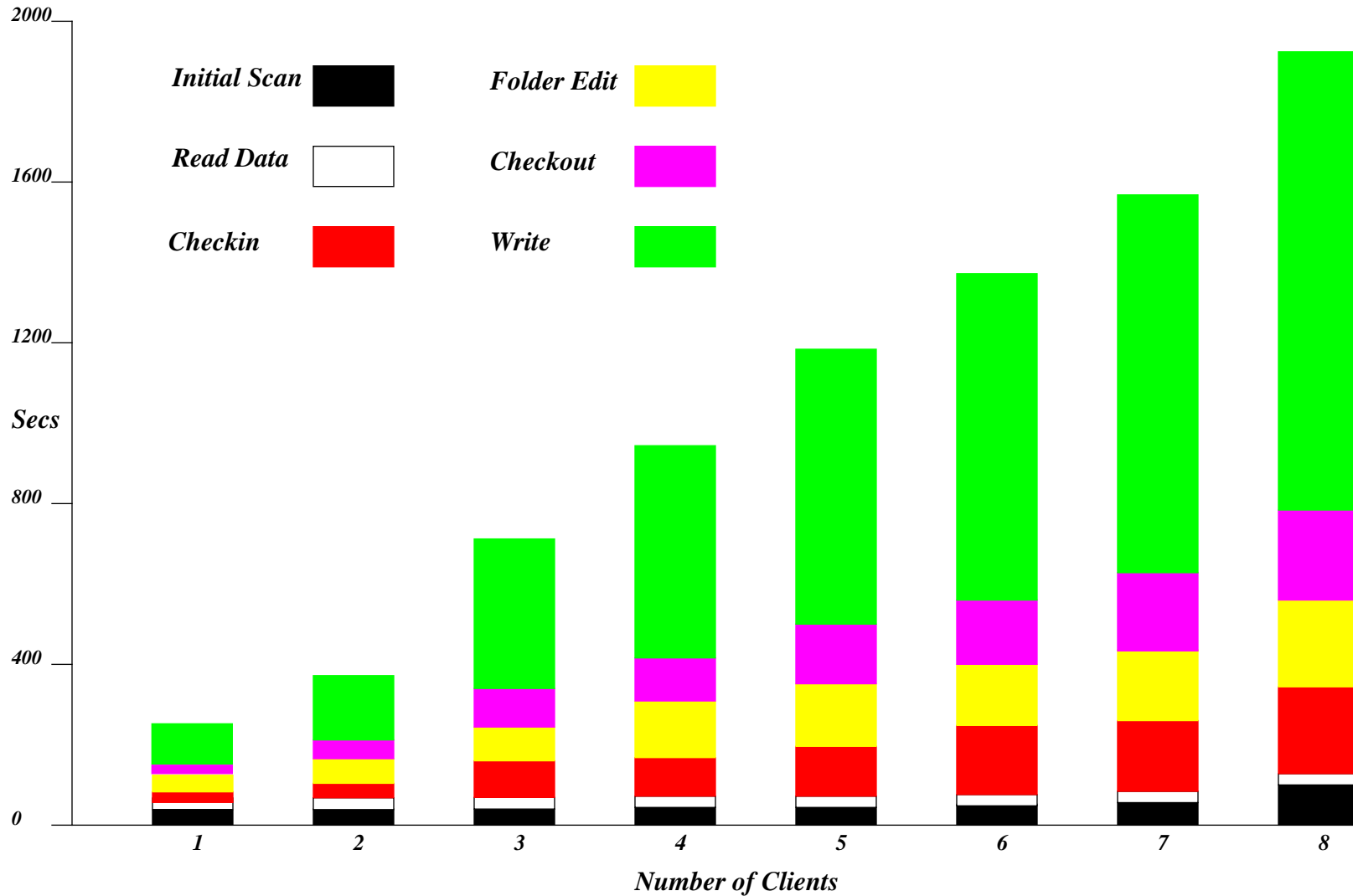
# *Performance Measurements*

---

- **How does performance scale as users are added?**
- **Versioning system benchmark**
  - **simulates a user performing a sequence of operations:**
    - **checkout a package at random**
    - **select a random number of components to edit**
    - **edit each component (by making a copy)**
    - **checkin new package version**
  - **each operation is a separate transaction**
  - **test database is 2500 text files (C++ source) organized into 216 packages**
  - **initial database size is 18Mb**
  - **measurements are for 25 iterations**
  - **multiple users run on separate machines with different random number seed**
  - **experiment run on 64Mb dual-processor SPARCstation™ 10**

SPARCstation is a trademark of SPARC International Inc., licensed exclusively to Sun Microsystems Inc.

# Benchmark Results (Single UID Generator)

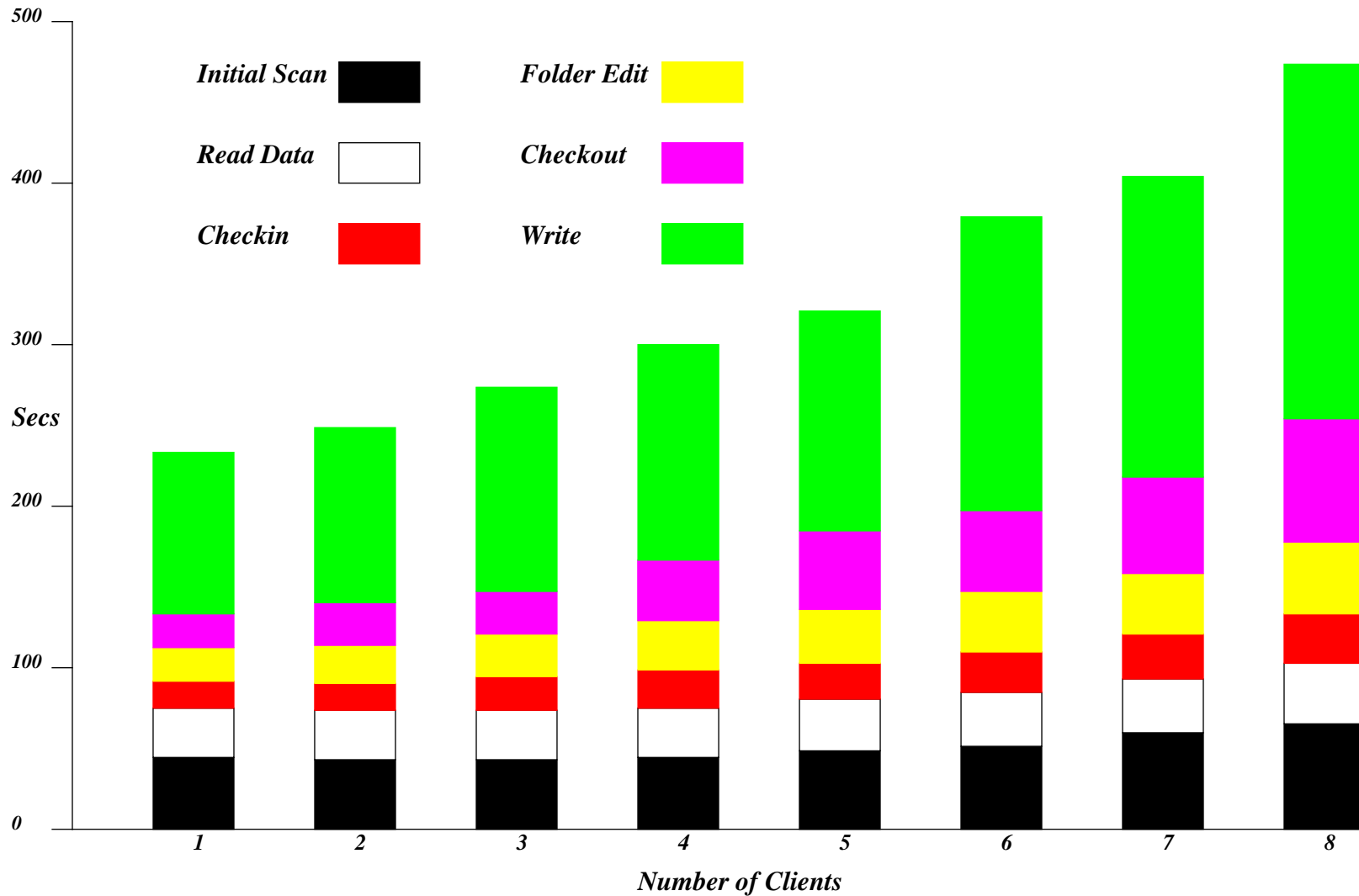


## *Multiple UID generators*

---

- **The UID data structures are a hot spot**
  - direct consequence of distributed shared memory model
  - applies to any global resource with high update rate
- **Solve by distributing UID space among components**
  - define a `UIDGenerator` interface
  - any component can elect to inherit and implement `UIDGenerator`
  - UID generating components form a tree structure
- **Exploit expected access patterns**
  - make each package a UID generator
  - could extend this to more levels if needed

# Benchmark Results (Multiple UID Generators)



## *Status and Future Plans*

---

- **Prototype**
  - versioning system implemented and stable
  - component-based compiler for Clarity C++
  - system modeller for Clarity C++ programs
- **Build a working prototype for mainstream languages**
  - e.g. ANSI C++, Modula-3
- **Explore the further use of fine-grained components**
  - language-dependent build-avoidance
  - structured document editors
  - configuration management for HTML

## *Conclusions*

---

- **Component objects are a sound basis for an SDE**
  - strong typing
  - immutability
  - separate interface and implementation
- **Current OODB technology can support a multi-user SDE**
  - relatively simple programming model
  - transacted data is well worth the cost
  - object clustering is important (together *and* apart)
- **Clear benefits from integrating:**
  - configuration management
  - editing
  - system building