A Deep Dive Into the Platform Resource Model

John Arthorne
IBM Rational

The Workspace Tree

John Arthorne
IBM Rational

Naive resource implementation

```
public class Workspace {
    WorkspaceRoot root;
}
class Resource {
    Workspace workspace;
    Marker[] markers;
    ...
}
class Container extends Resource {
    Resource[] children;
}
class WorkspaceRoot extends Container {}
class Project extends Container {
    Builder[] builders;
}
class Folder extends Container {}
class File {}
```

Actual resource implementation

```
public class Workspace {
        ElementTree tree;
}
class Resource {
        Workspace workspace;
        Path path;
}
class Container extends Resource {}
class WorkspaceRoot extends Container {}
class Project extends Container {}
class Folder extends Container {}
class File {}
```

- This is the grand total of fields on the resource classes

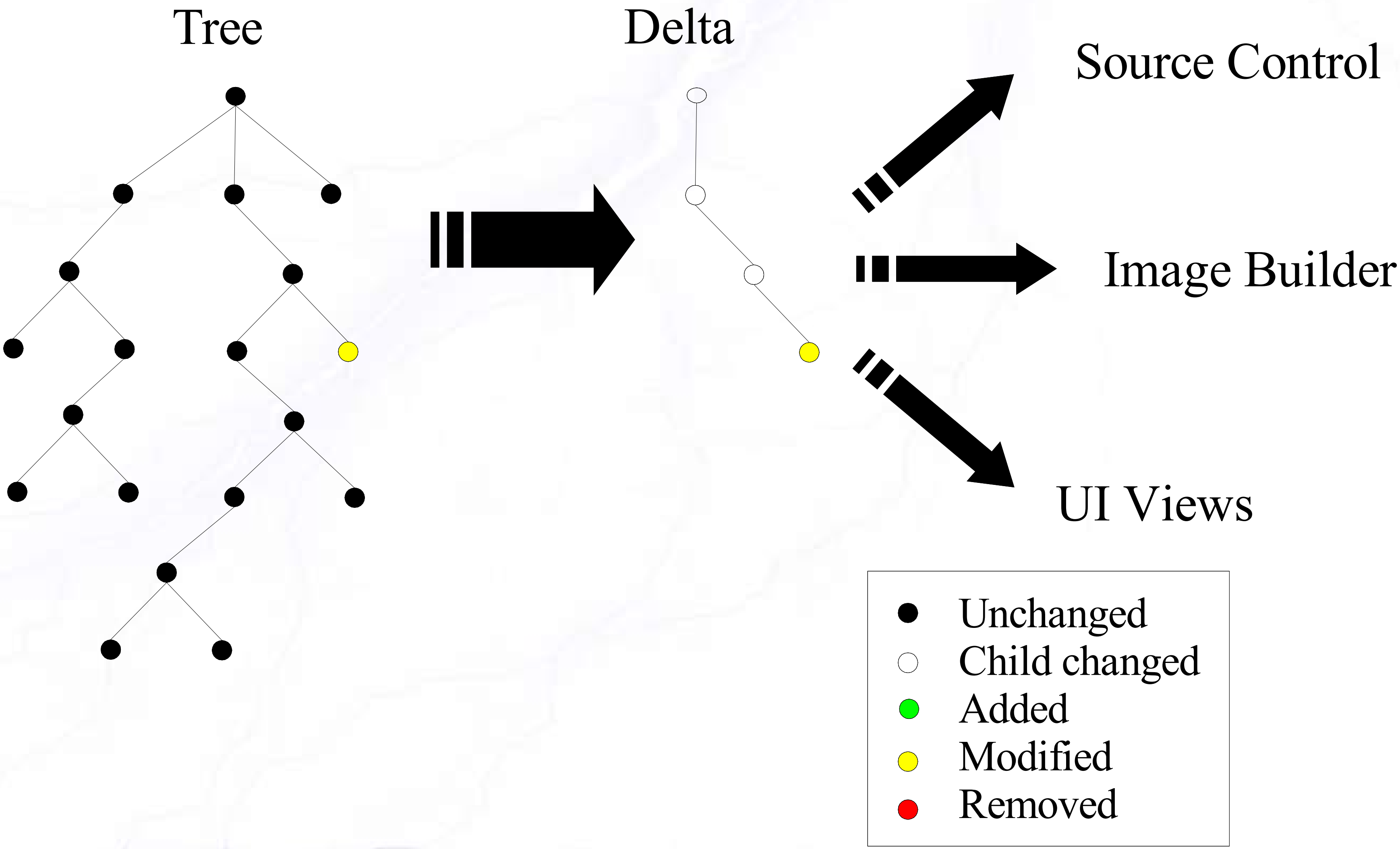
Resources are handles

- The resources plug-in doesn't hold onto any IResource objects – they exist only for clients
- IResource objects come and go as clients use them
- IResource objects are stateless and immutable
- All resource data stored in a single central data structure: the “element tree”.

Background Motivation

- Typical edit/compile/deploy cycle for a developer focuses on a small segment of a potentially very large code base
- Want to aggressively optimize for this common cycle: make performance cost proportional to the change, rather than to the size of the workspace
- Create and manipulate “units of change” that are passed around to interested parties

Data flow



Tree

Delta

Source Control

Image Builder

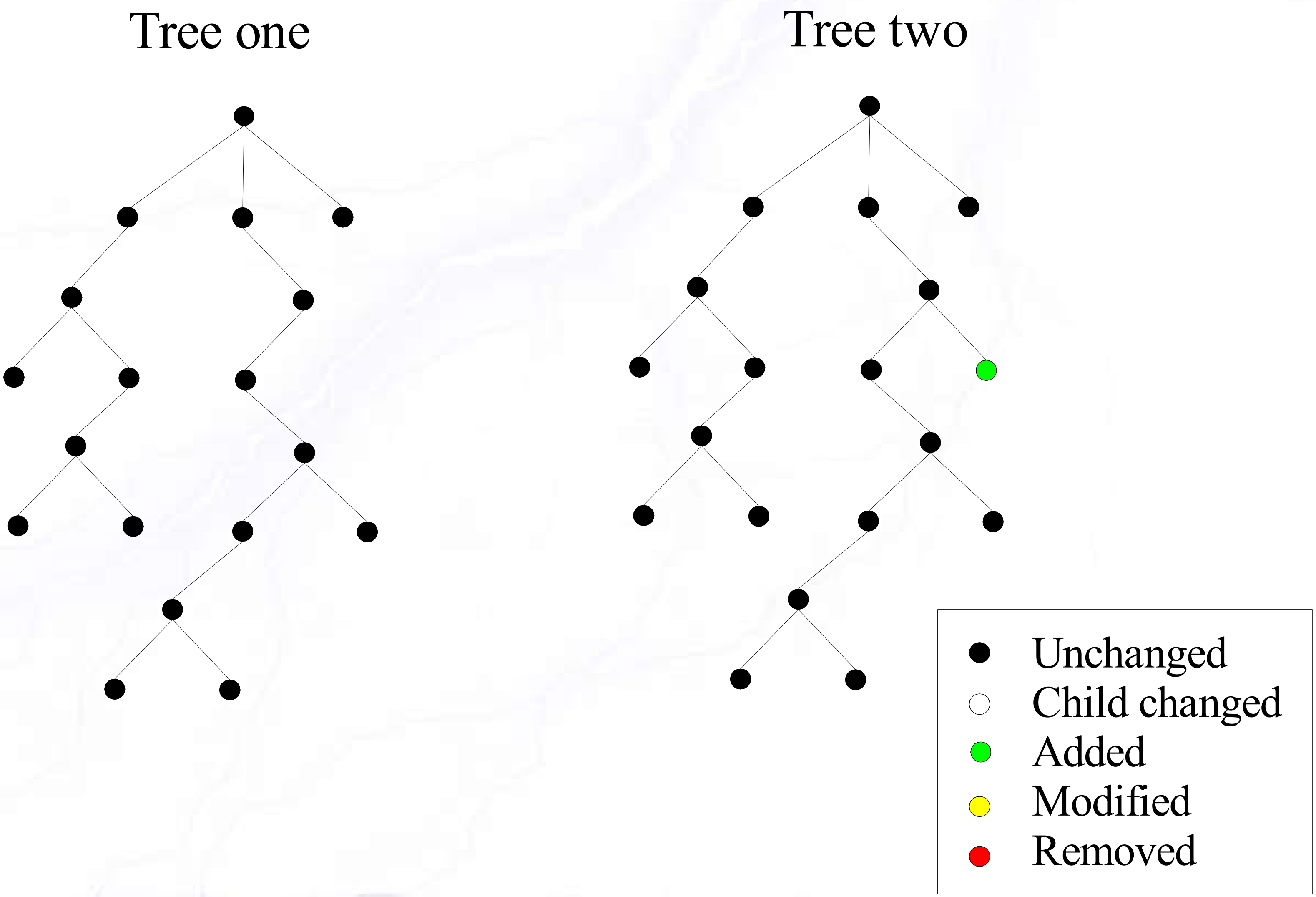
UI Views

- ● Unchanged
- ○ Child changed
- 🟢 Added
- 🟡 Modified
- 🔴 Removed

Additional observations

- Clients want a delta of the workspace state between two moments in time
- Different clients may want deltas with different start and/or end points
- How to efficiently represent all these different deltas in memory?
- How to compute these deltas without traversing entire workspaces?

Representing two tree states



- ● Unchanged
- ○ Child changed
- 🟢 Added
- 🟡 Modified
- 🔴 Removed

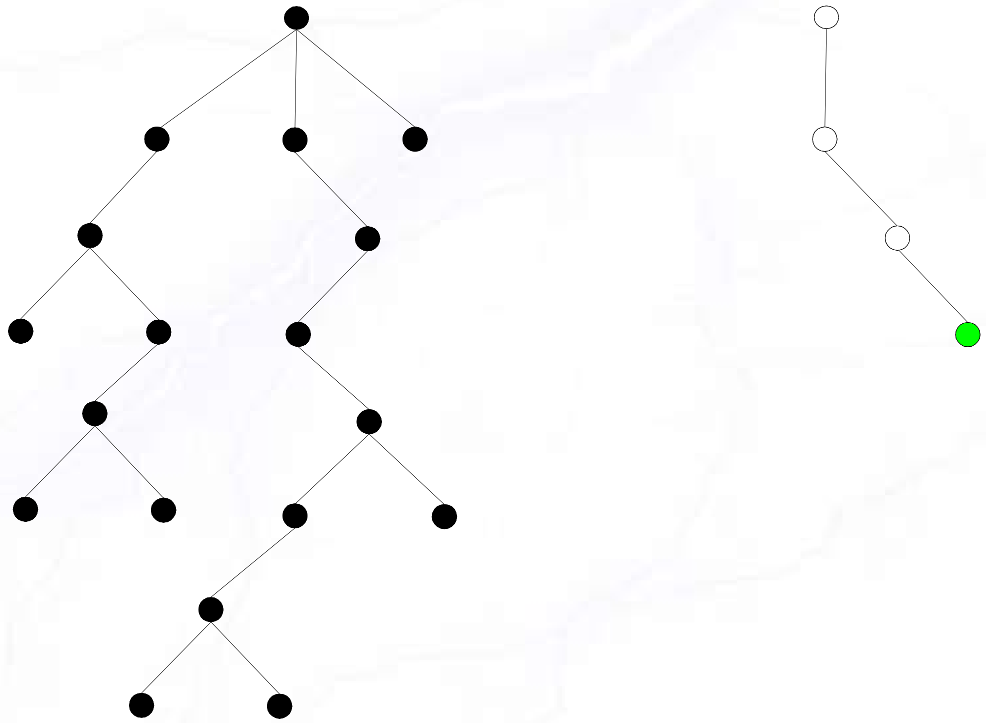
Representing tree states as deltas

Tree one (complete)

Parent = null

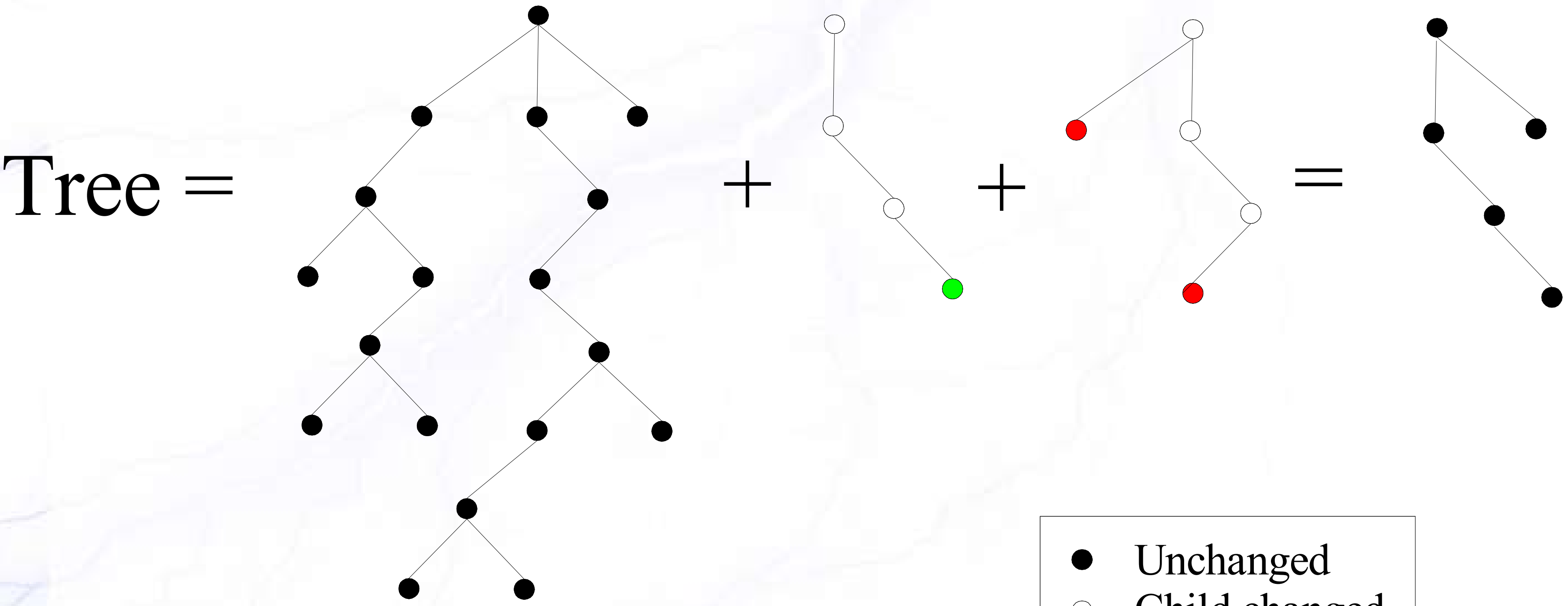
Tree two (delta)

Parent = tree one



- ● Unchanged
- ○ Child changed
- 🟢 Added
- 🟡 Modified
- 🔴 Removed

Tree state determined by assembling deltas with parents



- ● Unchanged
- ○ Child changed
- 🟢 Added
- 🟡 Modified
- 🔴 Removed

See the code! DeltaDataTree.lookup(IPath)

Checkpoint...

- There is one “complete” tree, and any number of “delta” trees that refer eventually to a complete tree as ancestor
- Each delta only stores changes from parent
- Each delta can act like a complete tree state by assembling with contents from parents
- We can represent many tree states efficiently
- Making sense so far?

Mapping terms to classes

- Package org.eclipse.core.internal.dtree
- DataTree – A complete tree
- DeltaDataTree – A tree that appears complete from the outside, but is represented as a delta against some parent
- A tree is made up of DataTreeNode objects
- Each node contains some “data”

● DataTreeNode
○ NoDataDeltaNode
🟢 DataTreeNode
🟡 DataDeltaNode
🔴 DeletedNode

Tree mutability

- Any given tree is either “open” or “immutable”.
- Only the nodes in open trees can be modified
- AbstractDataTree.immutable: makes a tree immutable
- DeltaDataTree.newEmptyDelta: creates new open tree
- Immutable trees are very powerful!
  - Node objects are freely shared between trees
  - Concurrency made easy
  - Can still change internal structure but for clients tree is unchanged

Delta tree calculus

- There are various methods on DataTree for manipulating lists of trees (possibly trees of trees)
- These are non-destructive operations: they have no effect on the contents of the tree from a client's perspective
- They may alter the internal representation of one or more trees

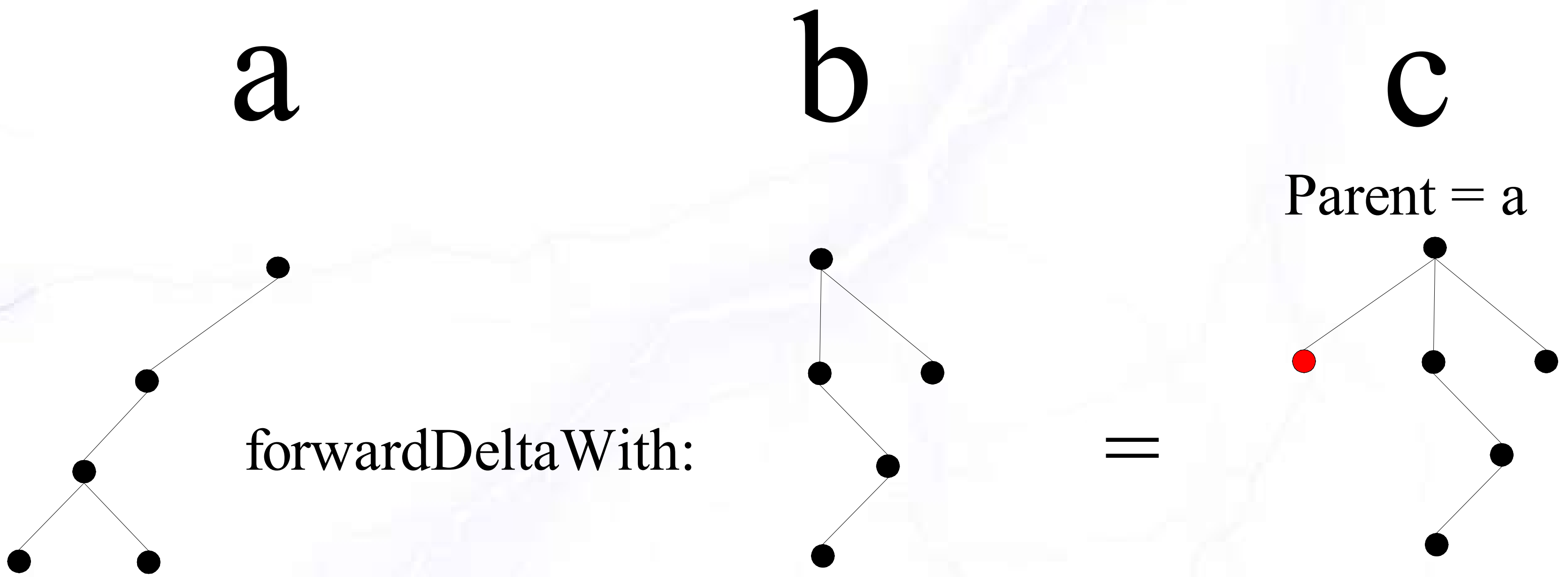
Delta tree calculus

- forwardDeltaWith: Create an equivalent tree represented as a delta against a different parent
- assembleWithForwardDelta: Inverse of above
- reroot: “flips” a chain of trees around to have a new parent
- makeComplete: make this tree complete by copying nodes from parent as necessary
- asBackwardDelta: returns a tree equal to my parent, but represented as a delta against me

Forward deltas

- Forward delta: Represents a tree state as a delta against a particular parent
- **c** = **a**.forwardDeltaWith(**b**);
- **c** has same contents as **b**, but represented as a delta tree with **a** as its parent
- **a**.assembleWithForwardDelta(**c**) -> **b**

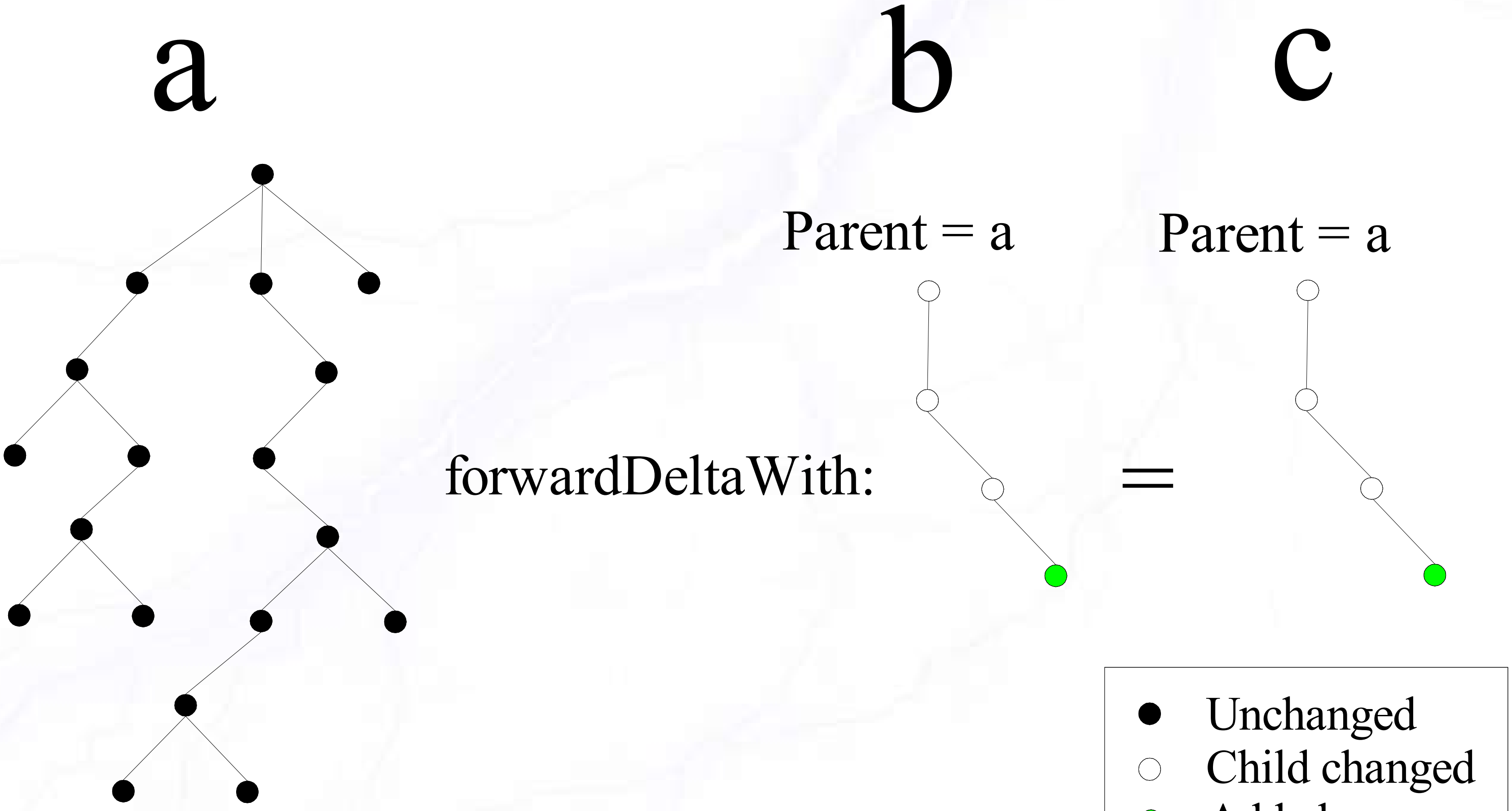
Computing forward deltas



- ➔ Nodes shared between trees if possible

● Unchanged
○ Child changed
🟢 Added
🟡 Modified
🔴 Removed

Computing forward deltas



- ● Unchanged
- ○ Child changed
- 🟢 Added
- 🟡 Modified
- 🔴 Removed

See the code! DeltaDataTree.forwardDeltaWith

Uses for forward deltas

- Used to write incremental snapshots of tree state to snap file
- Used to quickly determine if a build is need
- Tree garbage collection
  - Tree accumulates large number of layers over time
  - Only some of these layers represent tree states we still care about
  - Use forwardDeltaWith to clip out intermediate states

Checkpoint...

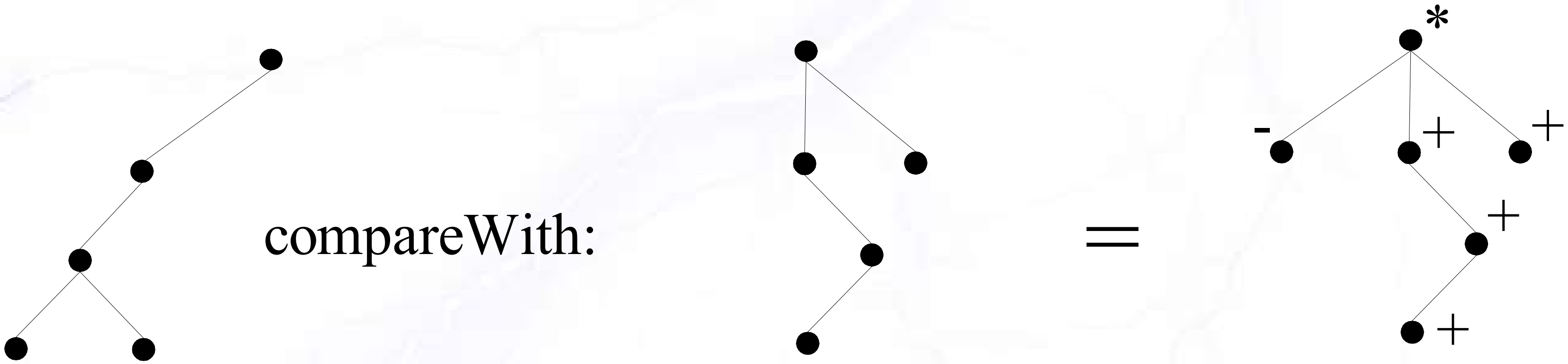
- Trees are either immutable or open
- Delta trees have a powerful set of methods for manipulating trees of trees
- An immutable tree can have its representation completely changed but its external appearance is frozen
- Making sense so far?

Exercise: DeltaTreeSample

Comparison trees

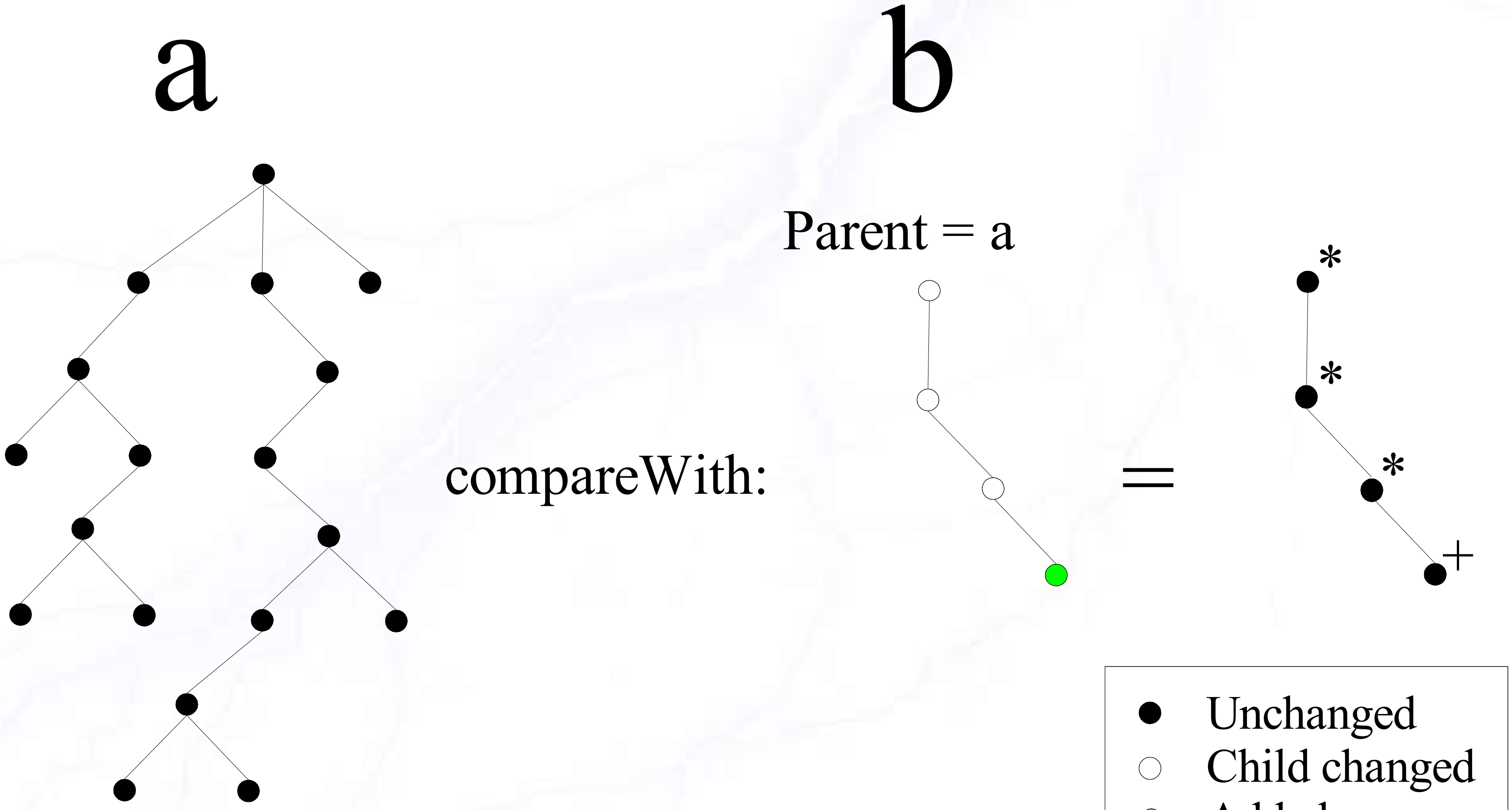
- Now say a client wants to know what changed between two tree states
- Common case: what is the most recent change?
- This info is baked into our tree representation
- Computing deltas (changes) between two states can be computed very quickly
- Most common case is nearly free because current tree **is** a delta against its parent tree

Comparison tree example



- ● Unchanged
- ○ Child changed
- 🟢 Added
- 🟡 Modified
- 🔴 Removed

Comparison tree example



- ● Unchanged
- ○ Child changed
- 🟢 Added
- 🟡 Modified
- 🔴 Removed

See the code! DeltaDataTree.compareWith

Comparison tree implementation

- Comparison trees are implemented using the same class: DeltaDataTree
- Instead of user data in the tree, each node contains a NodeComparison as its contents
- Comparison trees have no parent
- Comparison between related trees implemented by assembling forward deltas
- Client passes in a comparator for producing comparison flags on tree data

Uses for comparison trees

- Interesting tree states used as reference points for comparisons
- Comparison trees created for various kinds of resource deltas:
  - Resource change events
  - Builder deltas
  - Save participant deltas
- Data copied from comparison trees into ResourceDelta objects

Checkpoint...

- Delta trees play double duty:
  - Represent a new tree state by storing changes from previous state (forward delta)
  - Describe differences between two states (comparison trees)
- Delta trees change algorithmic complexity of many operations from O(n) to O(δ)
- Store multiple states and compare related states very efficiently

Exercise: DeltaTreeSample

ElementTree, my dear Watson

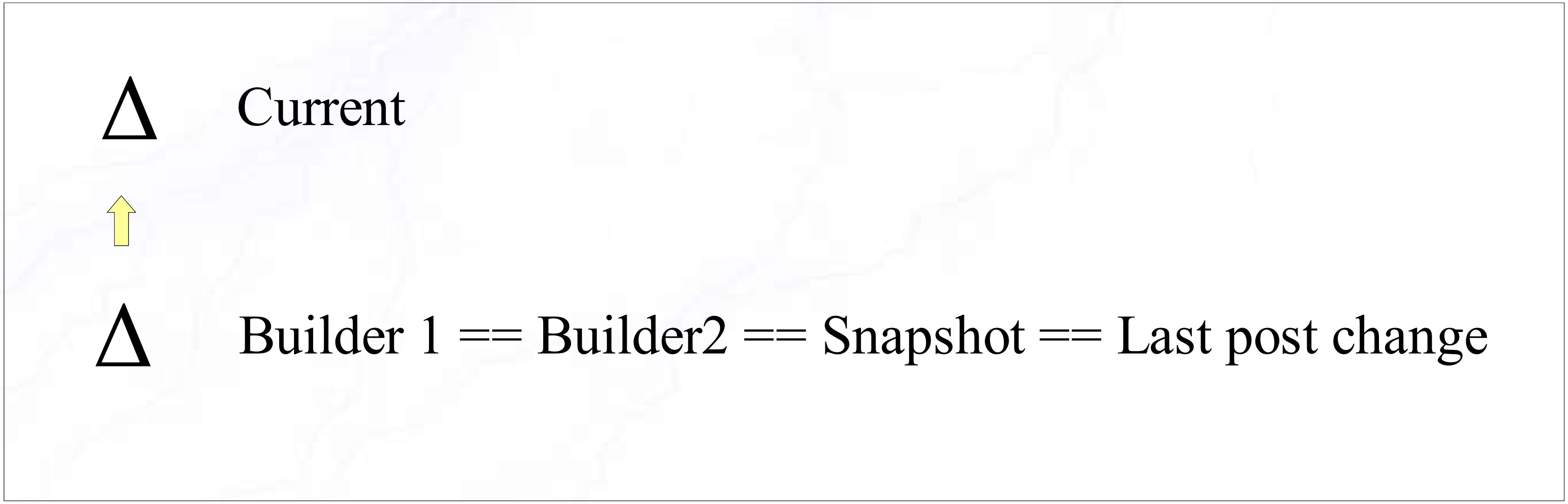
- DataTree API is very low level and complex
- org.eclipse.core.internal.watson.ElementTree
- ElementTree abstracts away some of the complexity of data trees
- Element trees have layers that map 1-1 to underlying data trees
- Adds visitor API
- Workspace implementation works almost exclusively with ElementTree abstraction

Using trees in the workspace

- At any given time in the workspace, there are various “interesting” tree states:
  - Current tree state (Workspace.tree)
  - State at time of last snapshot (SaveManager.lastSnap)
  - State at time of last resource change event (NotificationManager.lastPostChangeTree)
  - State at end of last invocation of each builder (InternalBuilder.oldState)
  - Save participant tree states

Tree state at startup

- If there was an auto-build before shutdown, all trees are the same
- One tree gets created during restore



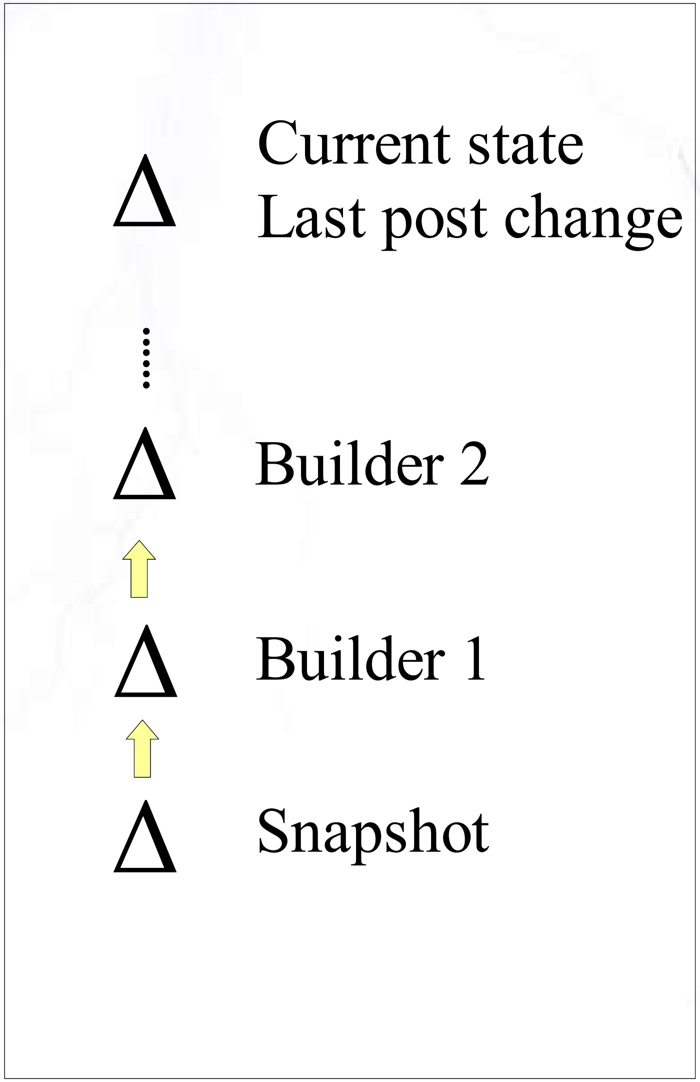
Δ Current

⇧

Δ Builder 1 == Builder2 == Snapshot == Last post change

Tree state after full build

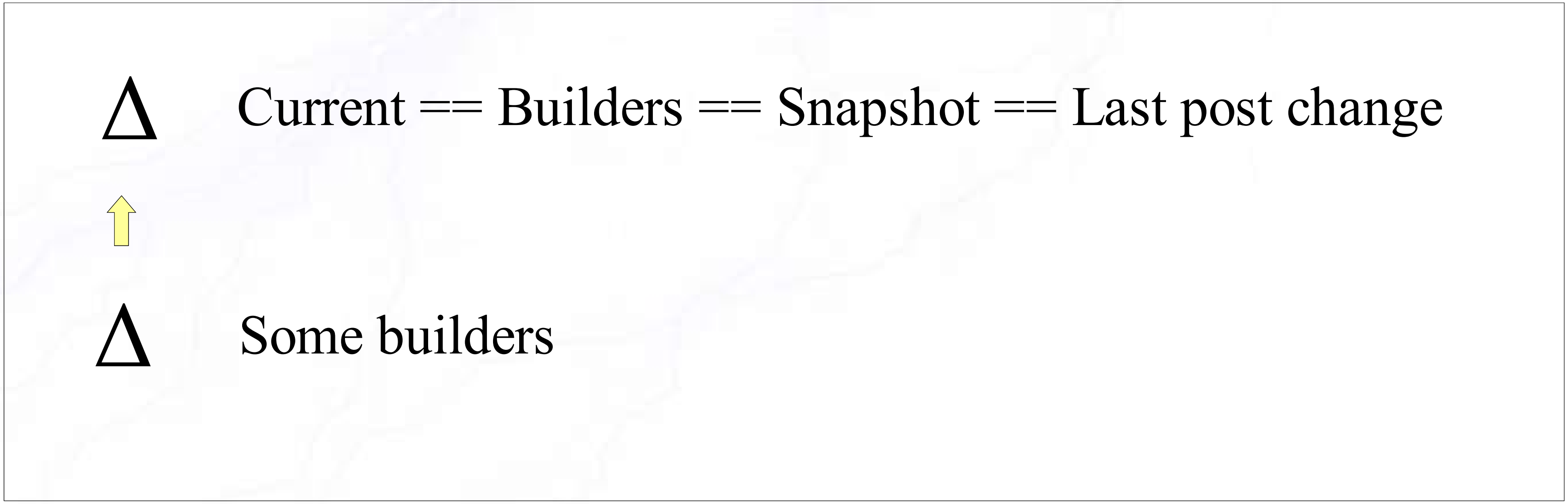
- Each builder that made changes has its own tree state
- Builders that didn't make changes share tree states
- Oldest tree is typically tree of last snapshot



EclipseCon 2009

Tree state with auto-build after snapshot

- Common case is that only one builder ran
- Builders before change will have an older tree
- Snapshot will delete unreferenced trees and move “last snap” pointer to current tree



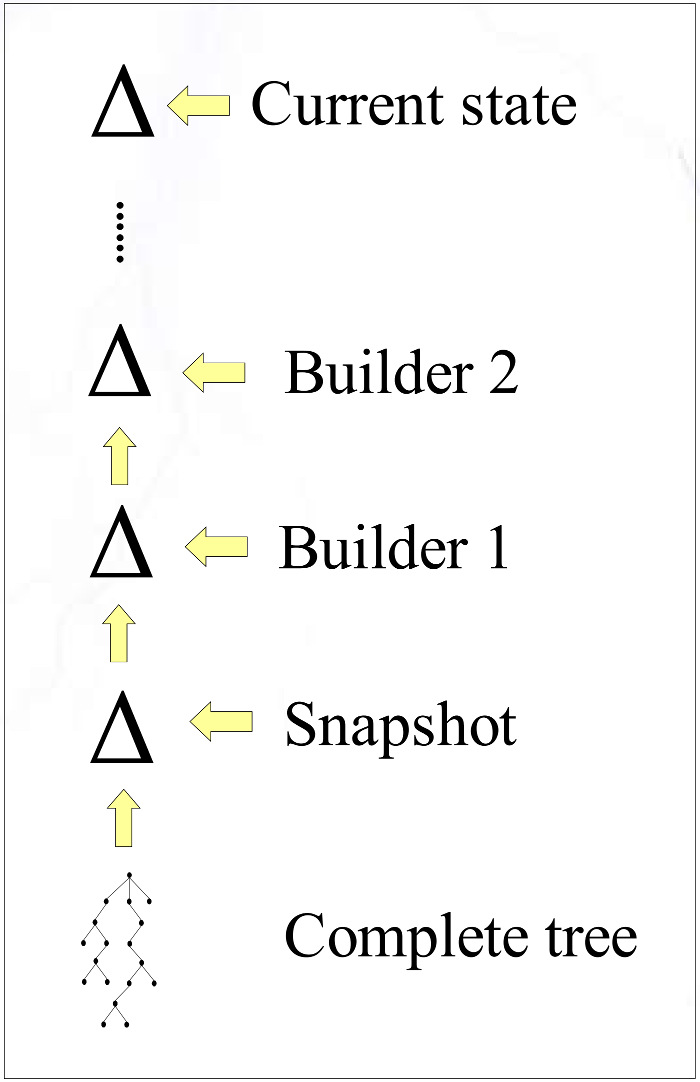
Δ Current == Builders == Snapshot == Last post change

⇧

Δ Some builders

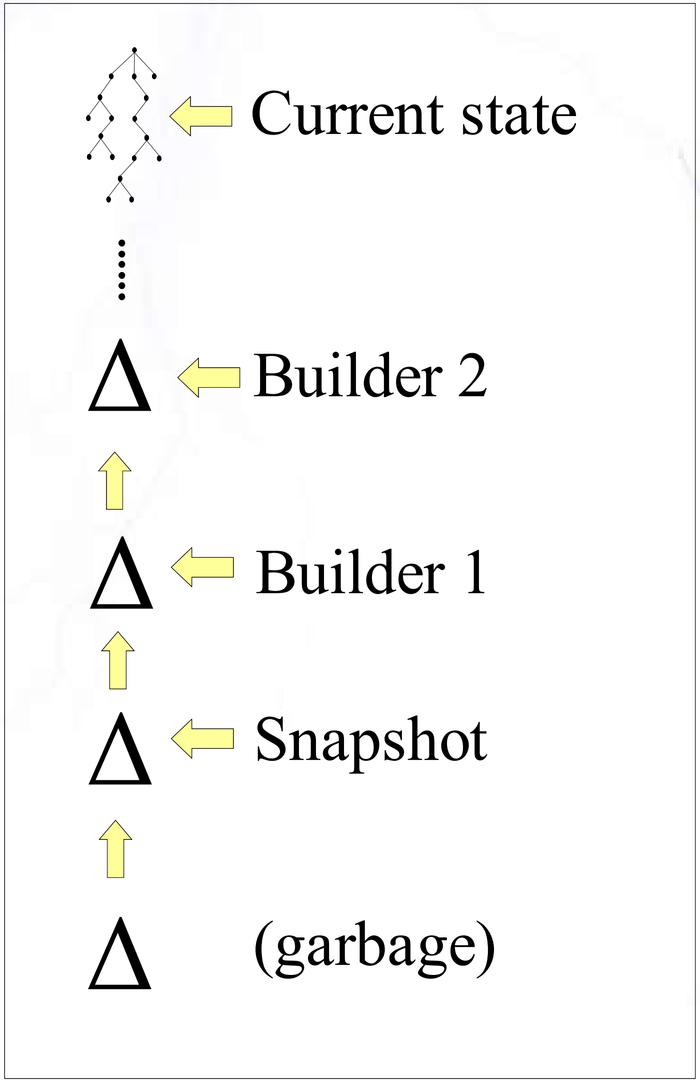
Tree garbage problem

- In original design, the oldest tree was always the complete tree, and newer trees were deltas
- Undo implemented by moving pointer back to older state
- This creates a problem with the tree growing indefinitely
- Performance slowdown from traversing many layers



Garbage solution: reroot

- Now re-root tree at most recent state at end of top level operation
- Old tree states garbage collected automatically
- Re-rooting not as expensive as it seems, since node objects can be shared between trees
- Comparison trees often need to be flipped around too


Current state

Builder 2

Builder 1

Snapshot

(garbage)

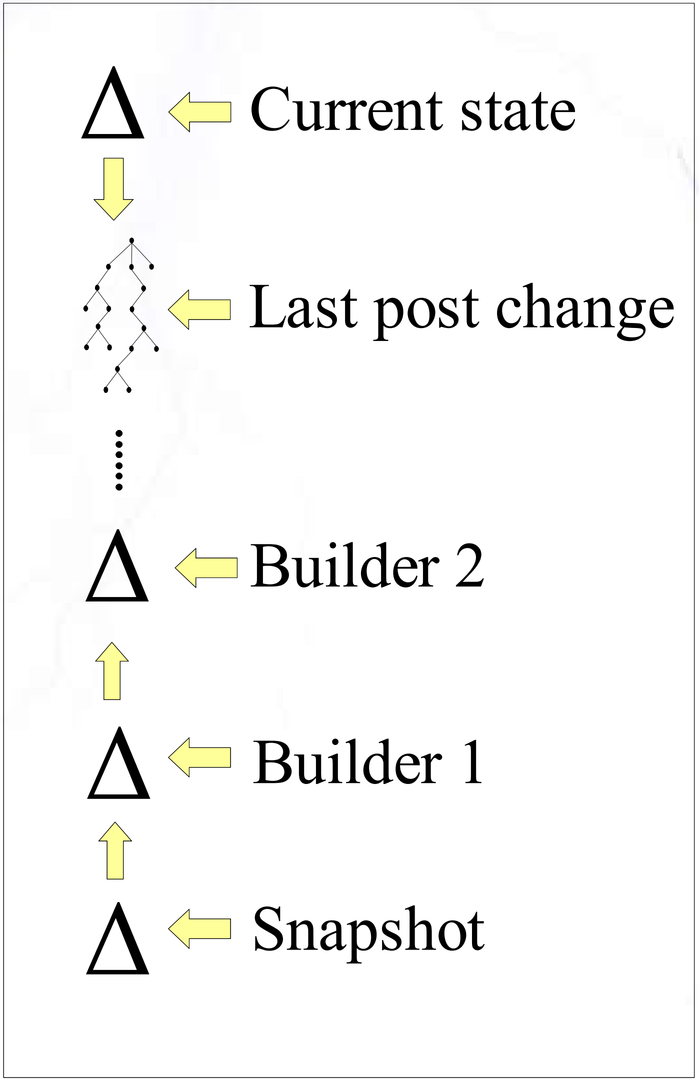
EclipseCon 2009

Handling workspace changes

- Workspace has a notion of “operations” - any code that modifies the workspace runs in the context of an operation
- During an operation, Workspace.tree is a mutable tree
- We need to carefully distinguish workspace-modifying code from read-only code
- Workspace.getResourceInfo – either reading or writing depending on “mutable” argument

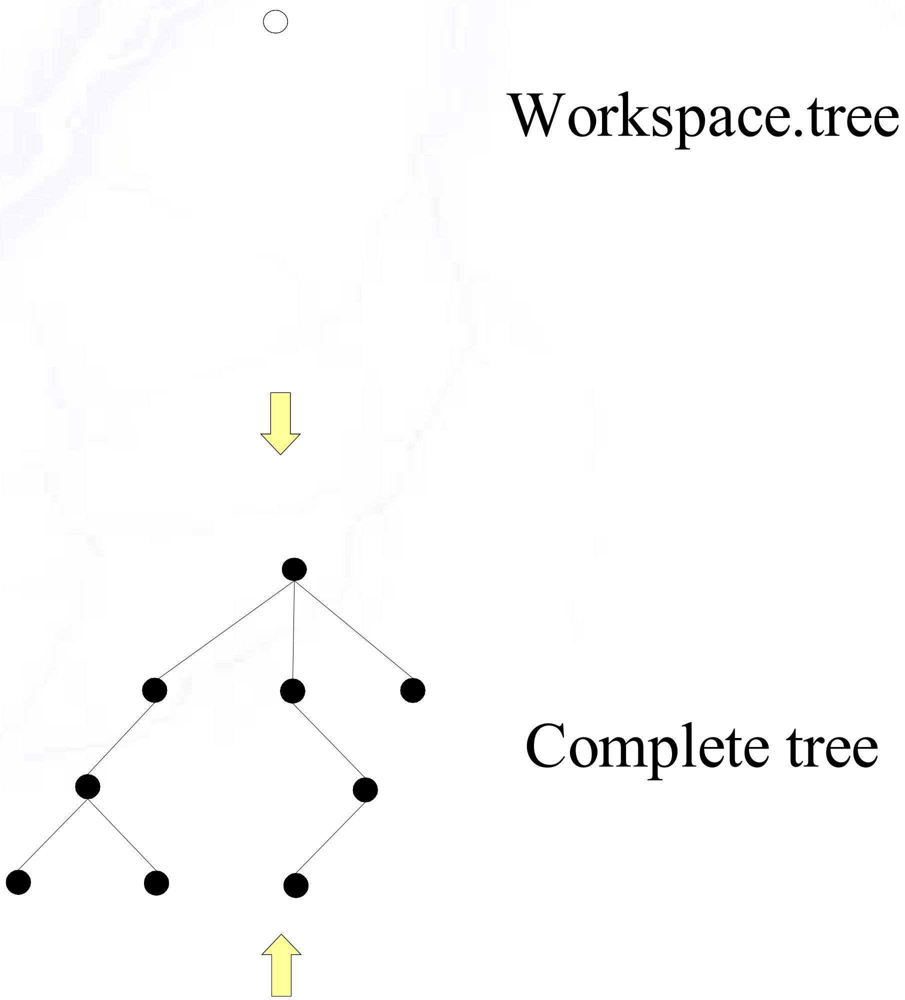
Trees during an operation

- Current tree is an open delta
- Immediate parent is the complete tree (fast lookup)
- Older trees also children of complete tree
- At end of operation, computing resource change event delta is trivial



Example: start of operation

- New empty delta
- Complete tree is parent
- More older trees below complete tree

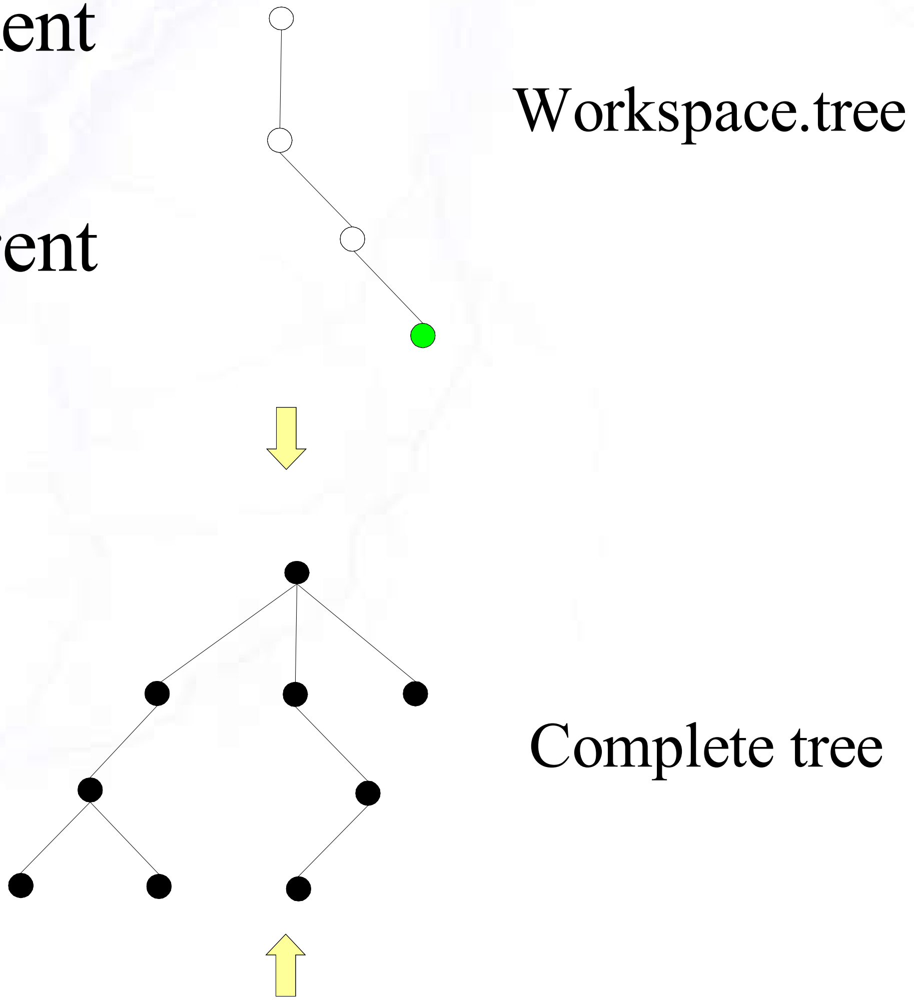


Workspace.tree

Complete tree

Example: file is created

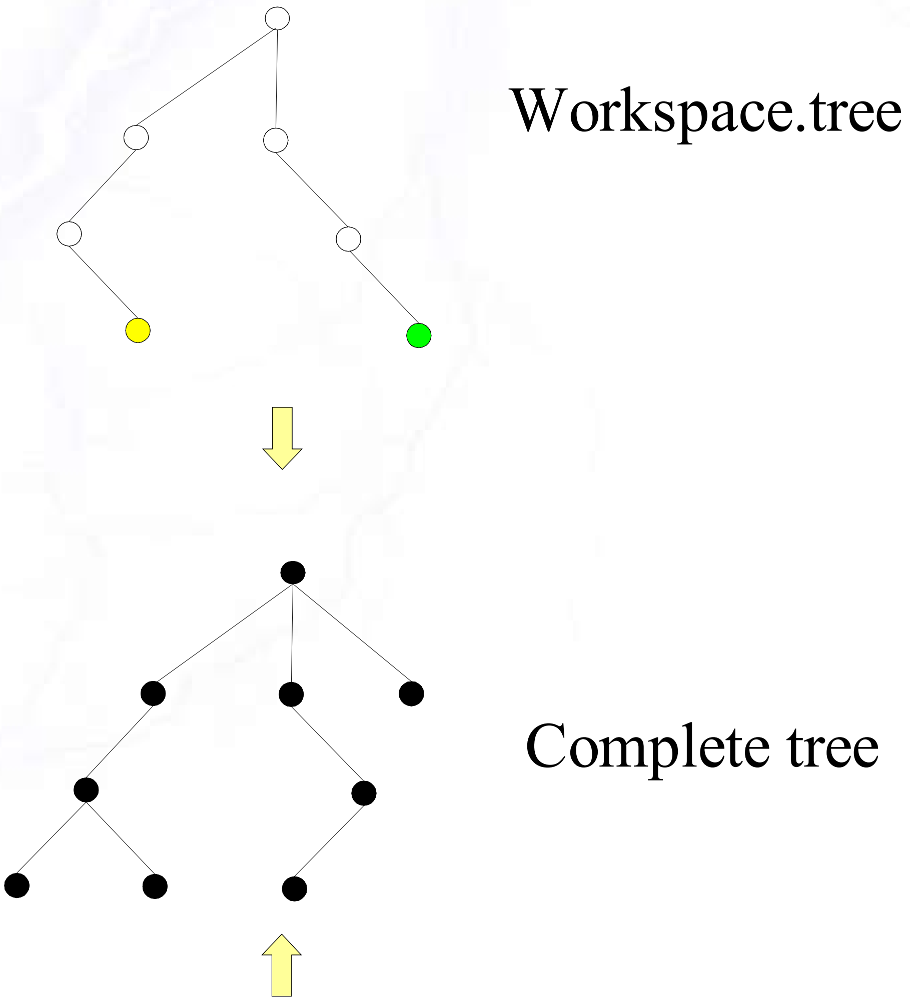
- ElementTree.createElement called
- Empty (see-through) parent nodes



- ● Unchanged
- ○ Child changed
- 🟢 Added
- 🟡 Modified
- 🔴 Removed

Example: file is modified

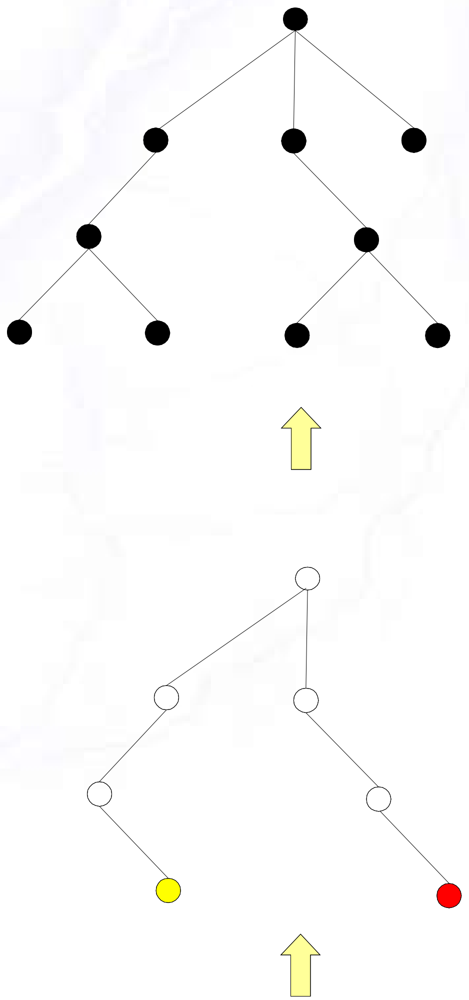
- openElementData
- Modified node is "pulled up" (copied) to the open tree layer



● Unchanged
○ Child changed
🟢 Added
🟡 Modified
🔴 Removed

Example: end operation

- Reroot at current state
- Old tree re-cast in terms of new tree
- All other descents of complete tree unaffected



Workspace.tree is complete

Old tree

● Unchanged
○ Child changed
🟢 Added
🟡 Modified
🔴 Removed

Checkpoint...

- Workspace holds onto all tree states it is still interested in
- Deltas can be computed between known tree states for resource change events, build deltas, etc
- Single open tree layer during operations, all other trees immutable
- Must always “pull up” any modified resource into the open tree layer

Exercise: element tree spy

Summary

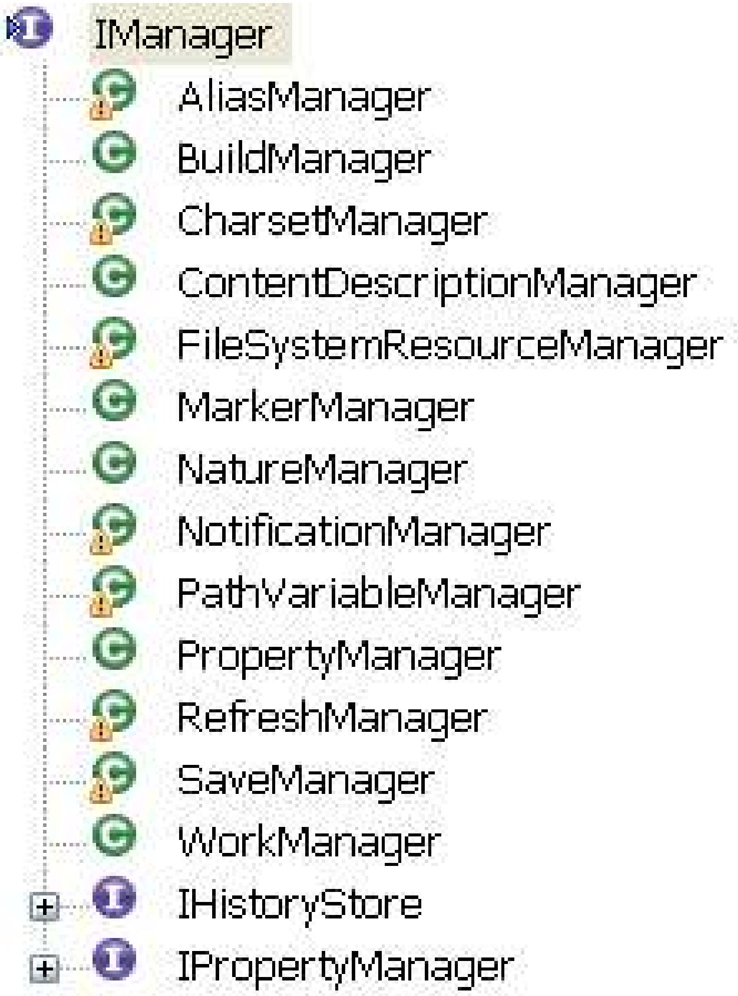
- Delta trees allow us to efficiently represent a large number of different tree states, and efficiently compute differences between tree states
- Whether a tree is complete, or represented as a delta against some parent is not evident to tree clients
- Mutable trees used during operations, immutable trees for old states
- Can manipulate chains of trees without changing their contents from clients' perspective

Survey of resource API internals

John Arthorne
IBM Rational

EclipseCon 2009

Upper management



IManager
- AliasManager
- BuildManager
- CharsetManager
- ContentDescriptionManager
- FileSystemResourceManager
- MarkerManager
- NatureManager
- NotificationManager
- PathVariableManager
- PropertyManager
- RefreshManager
- SaveManager
- WorkManager
- IHistoryStore
- IPropertyManager

- IManager: participation in workspace startup/shutdown
- FileSystemResourceManager: manages mapping from resource layer to file system layer
- SaveManager: everything to do with persistence of workspace and resource metadata
- NotificationManager: resource change events

Other interesting bits

- LifecycleEvent: magic internal resource events only for internal use
- OS: Captures platform-specific rules such as reserved characters and file names
- LocalMetaArea: Abstracts away all interaction with the workspace metadata location (workspace/.metadata)
- Policy: trace options, log helper methods

ResourceInfo

- The data stored in tree nodes is ResourceInfo
- Also ProjectInfo, RootInfo
- All resource state in these info objects
- ResourceInfo copied the first time it is modified in a top-level operation
- ResourceComparator: The delta tree comparator that compares resource states (ResourceInfo)

ResourceInfo

```
public class ResourceInfo implements {
  protected volatile int charsetAndContentId;
  protected FileStoreRoot fileStoreRoot;
  protected int flags;
  protected volatile long localInfo;
  protected volatile int markerAndSyncStamp;
  protected MarkerSet markers;
  protected long modStamp;
  protected volatile long nodeId;
  protected ObjectMap sessionProperties;
  protected ObjectMap syncInfo;
}
```

ResourceInfo timestamps

- Content ID: Incremented every time content changes (file contents or project description)
- Local info: The local file system timestamp
- Modification stamp: allows clients to detect changes (IResource.getModificationStamp)
  - Used to support undo
  - Also affected by project open/close, existence of link target

Back doors for team systems

- VCM systems often have unique requirements different from most other clients
- Rather than opening up special functionality to everyone, we opted for special “back door” hooks for team providers:
  - FileModificationValidator: pessimistic VCMs
  - TeamHook: generic place for team hooks
  - MoveDeleteHook: for tracking moves/deletes
  - “Team private” resources

IFileModificationValidator

- Some VCM systems require a checkout before a file is modified (pessimistic model)
- This hook gives VCM's a chance to perform checkout
- Well-behaved clients of resources should call validateEdit before making changes to read-only files
- Safety net: validateSave always called

IMoveDeleteHook

- Clients implementing IMoveDeleteHook can completely re-implement copy and move operations
- Or, can just insert special code before or after the default copy/delete implementations
- IResourceTree: special back door API for move/delete hooks
  - Default move/delete methods
  - Methods to update resource tree

TeamHook

- Lesson learned in API design: don't use interfaces for bits implemented by clients
- All team hook extensions could have been rooted at a single extension / single base class
- TeamHook is an abstract class, so we can add future methods without breaking clients

History and properties

- Local history and persistent properties stored directly on disk (see “persistence” slides)
- Had old b-tree implementation that was replaced because it was unstable, too complicated
- org.eclipse.core.resources.compatibility contains the old implementation
- IHistoryStore and IPropertyManager either old or new implementation
- Compatibility eagerly migrates to new format

Builders

- BuildManager implements build logic
- Determines whether or not each builder invocation is needed
  - Does builder respond to current trigger?
  - Is delta non-empty?
- BuilderPersistentInfo: used to hold data maintained about each builder across sessions
  - Stored in session property until first run

Exercise

- Fix “Resource Spy” and “Project Spy”

File System Synchronization

John Arthorne
IBM Rational

Basic principles

- The file system is king
- When there are conflicts between changes in workspace and changes in file system, the file system always wins
- Synchronization is always just updating workspace tree based on disk state
- Should only ever become out of sync if file system is modified external to resource API

Basic principles

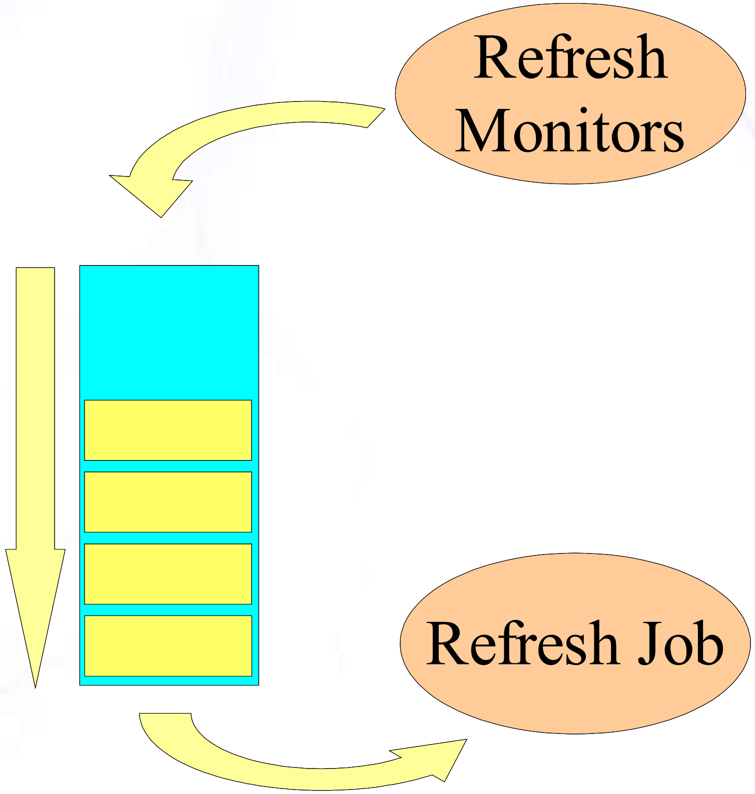
- Synchronization between workspace and file system is explicit in the API
- Avoid data loss from workspace being unexpectedly synchronized
- Prevent accidentally running/testing/releasing code that doesn't match what user sees in workspace
- Synchronization can be expensive

Auto-refresh

- Ongoing attempts to implement auto-refresh over the years
- Jed Anderson's auto-refresh plug-in
- Auto-refresh added to platform in 3.0
- Native implementation on Windows
- Attempted to implement FAM native support on Linux (bug 52859)
- Polling based auto-refresh outside Windows

Auto-refresh structure

- Completely asynchronous
- Pluggable refresh monitors can issue refresh requests
- Requests stored in queue
- Refresh job performs actual refresh in the background
- Refreshes in small chunks to avoid interference



Refresh job tricks

- Workspace is locked during refresh, so we want background refresh to avoid locking for too long
- Vast differences in file systems makes this difficult to optimize
- Refresh job learns refresh speed, and adapts refresh depth dynamically
- Start by only refreshing to depth 2, keep doubling depth while longest refresh is < 1s

See the code! RefreshJob#runInWorkspace

Polling job tricks

- The polling job's work is never done
- Want to keep polling unobtrusive, so it only runs for fixed periods
- Job starts with a collection of roots that need polling
- Poll the root where recent changes have been found more frequently (“hot root”)
- Reschedule job based on function of last run's duration:

```
long delay = Math.max(MIN_FREQUENCY, time * 20);
schedule(delay);
```

Checkpoint...

- File system is king
- Synchronization is explicit in API
- Want to make synchronization unobtrusive for end users using auto-refresh
- Questions on refresh principles and auto-refresh?

UnifiedTree

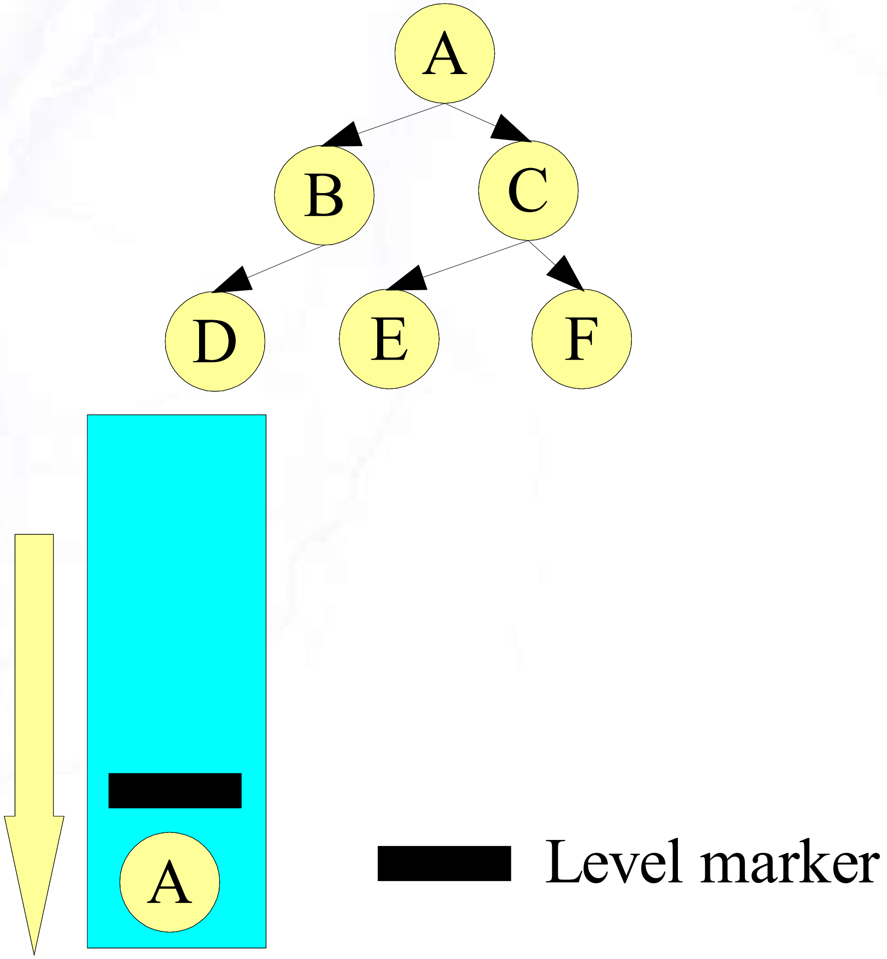
- UnifiedTree is a data structure that represents the union of the file system and the workspace
- Used for refresh, for isSynchronized, “forced” copy, and best-effort deletion
- Uses visitor pattern with breadth-first traversal
- Visitors implement IUnifiedTreeVistor, which accepts UnifiedTreeNodes
- Each node represents a file/folder in workspace, file system, or both

Implementing UnifiedTree

- Too expensive to represent entire tree in memory at once
- Tree representation is a queue
- Only keep nodes in memory for one tree layer at once
- Use special marker nodes to record current depth, and distinguish one node's children from another

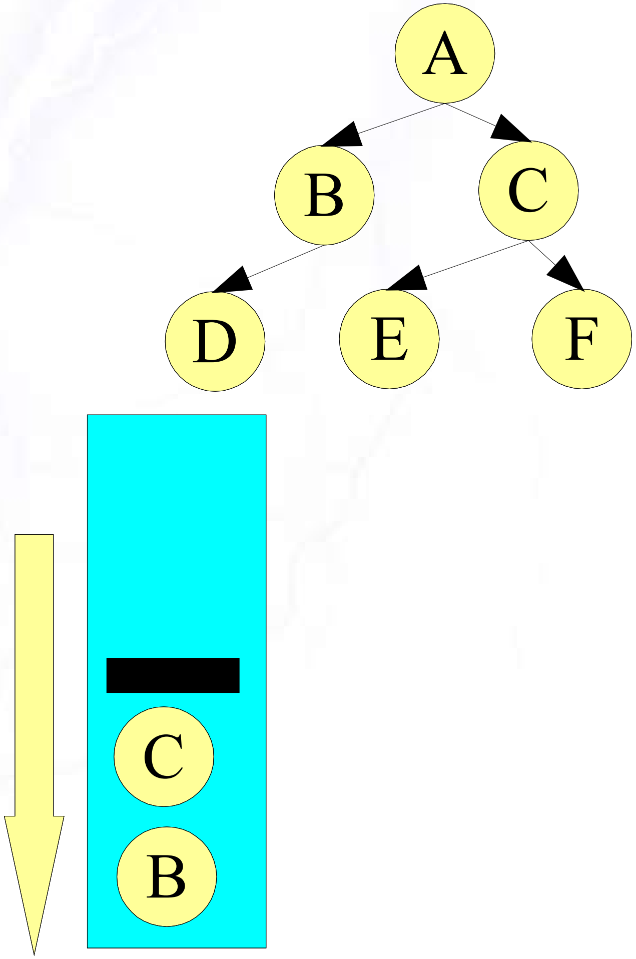
UnifiedTree: depth zero

- Start with root in queue



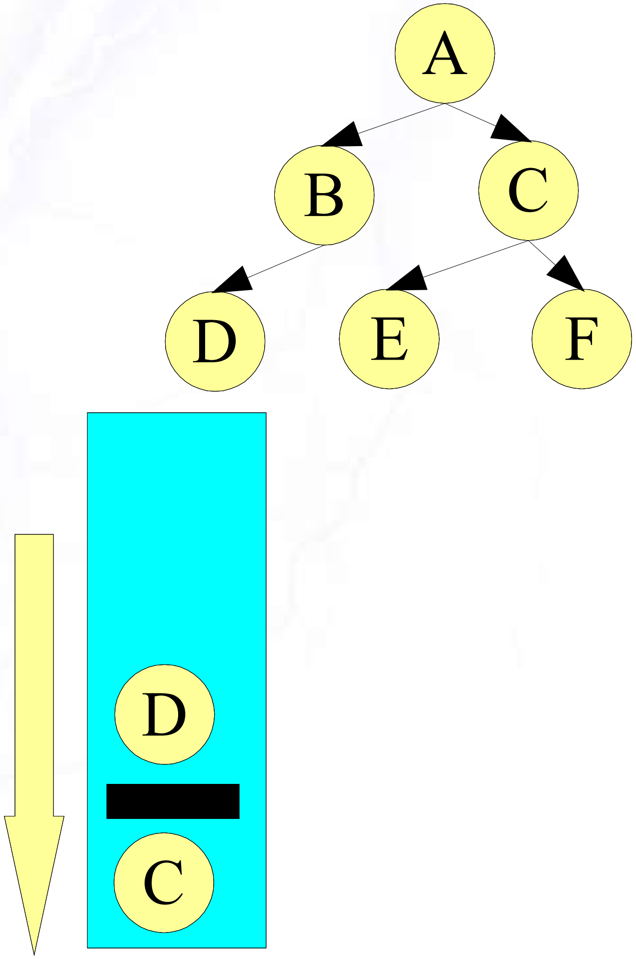
UnifiedTree: depth one

- After completing A, its children are added to queue
- Next is a level marker, which we move to back of queue



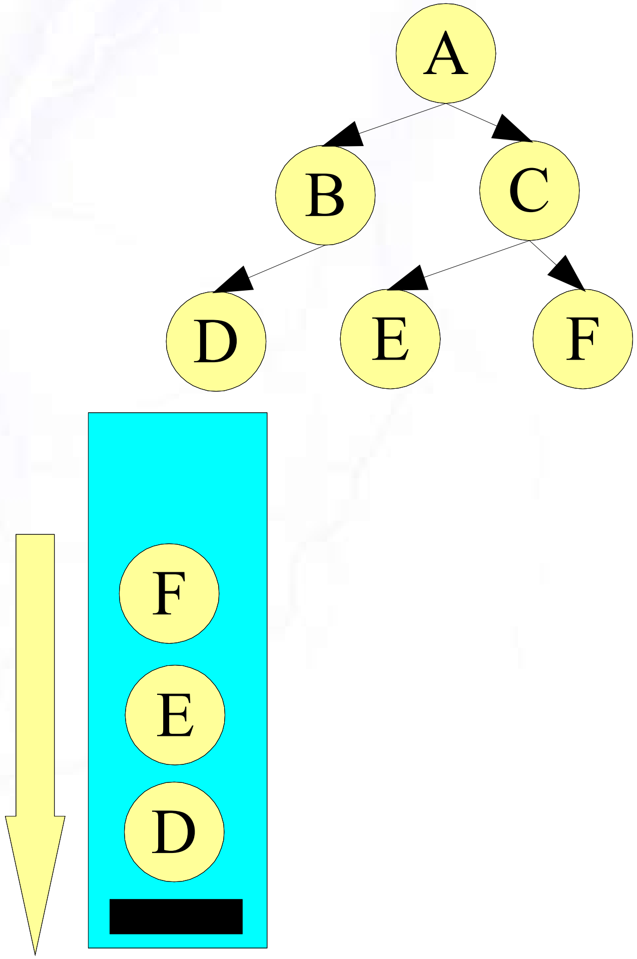
UnifiedTree: depth one

- Process B, add child to queue



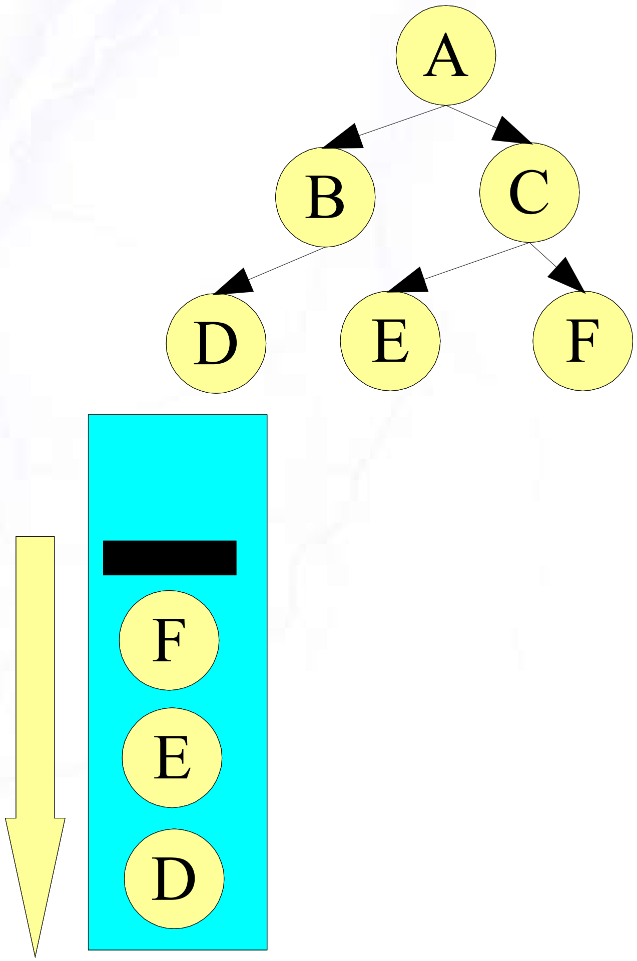
UnifiedTree: depth one

- Process C, add children to queue



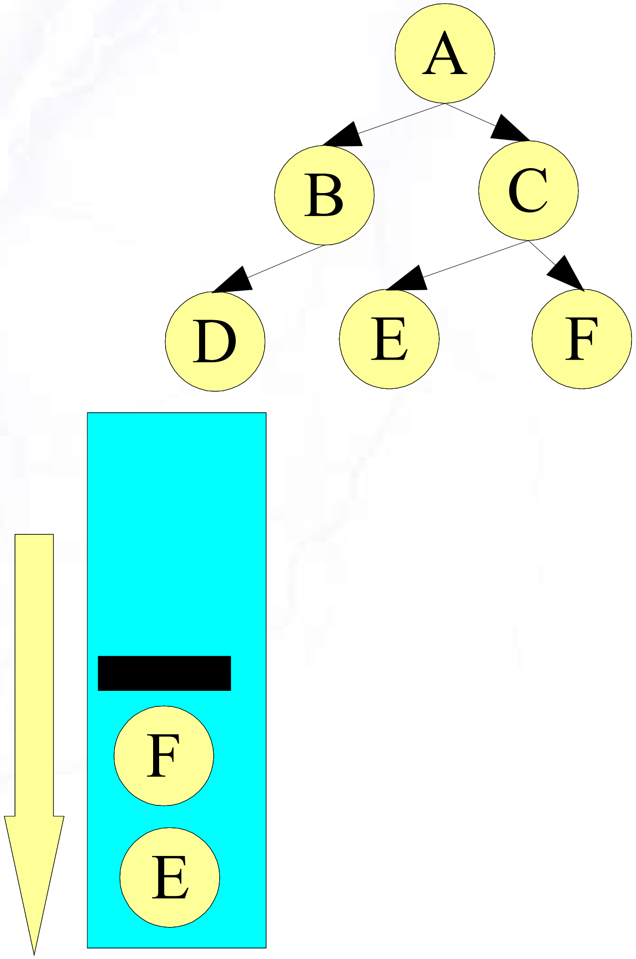
UnifiedTree: depth two

- Move level marker to back of queue
- Process remaining children



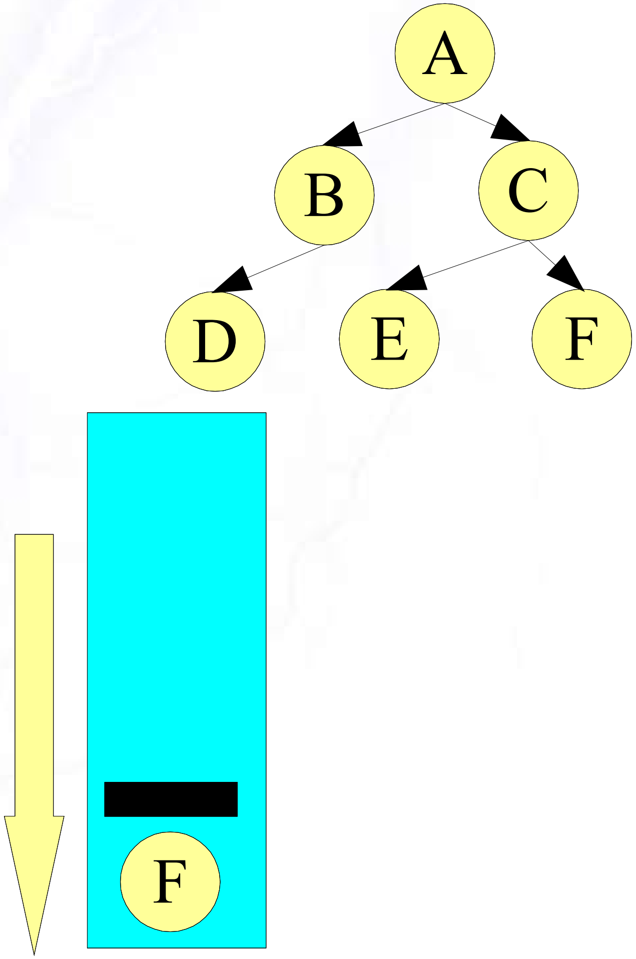
UnifiedTree: depth two

- Process remaining children



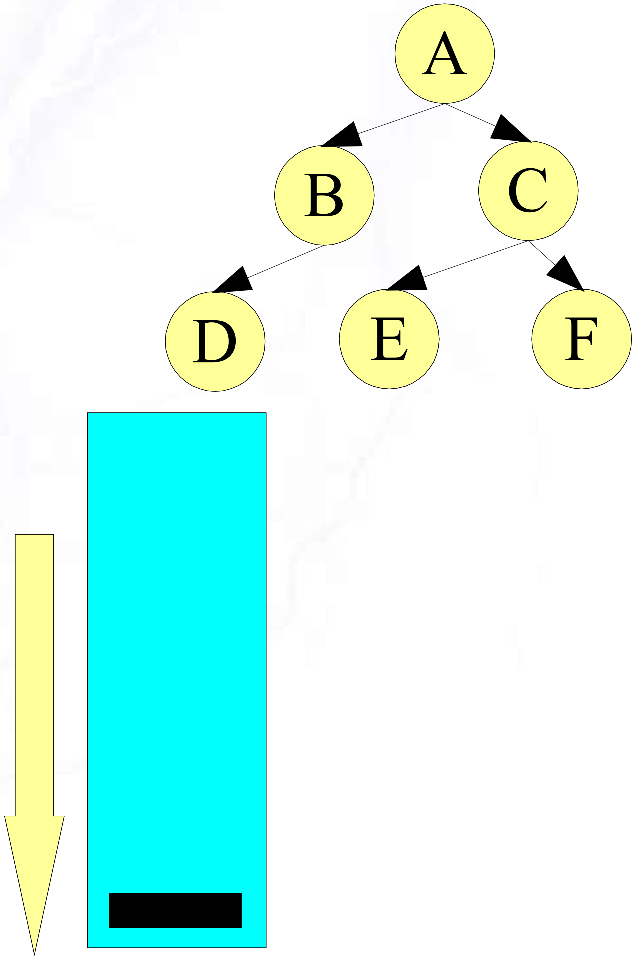
UnifiedTree: depth two

- Process remaining children



UnifiedTree: depth two

- Process remaining children



Refresh performance

- Refresh needs to be **very** fast
- Refresh implementation optimized to only make one file system call per resource
- Aggressive optimizations made to avoid creating garbage
- Unified tree nodes hold onto all data that is needed for duration of single resource refresh
- Recycle node objects and reuse them for next layer

Checkpoint...

- UnifiedTree used for operations that require synchronization with the file system
- Tree represented as a lazily-populated queue that performs breadth-first traversal
- Exercise: Writing sync state spy

Linked resource history

- Eclipse 1.0 resource design very simple, with each project having a file system location
- Within projects, resource tree matched file system tree 1-1
- Project locations not allowed to overlap
- No two resources share the same file system location

Linked resource motivation

- Users had complex existing file system layouts that they wanted to use in Eclipse
- Projects sharing a common root directory (different ideas about what constituted a project)
- Library folders shared between projects
- Want to allow more complex mappings between workspace and file system

Linked resource principles

- Linked resources don't point to other resources
- Linked resources point to a different file system location than their parent
- Fundamental difference from sym-links: they do not introduce cycles in the workspace tree
- Not a special resource type. Apart from the location, act like regular files and folders
- Exception: links continue to exist when location does not (file system is not king)

Linked resource overlaps

- Original linked resources only allowed as direct children of project
- Later relaxed to allow links at any depth
- Later relaxed rule against overlapping project locations
- Now resource trees overlapping in the file system are common

Aliases

- Alias: The aliases of a given resource are the other resources in the workspace that share the same file system location
- Our principle of not getting out of sync using resource API means we need to update the state of all aliases on every resource change
- Need to do this efficiently without expensive alias search on each change

Aliases

- Three level optimization of alias search:
  - Maintain counter of all resources with "non-default" locations
  - Maintain list of projects containing overlaps
  - Maintain map of locations to "roots" at that location (linked resources or projects)
- Minimal added overhead if you have no overlaps
- Use TreeMap.subMap to find overlaps

See the code! AliasManager.computeAliases

Checkpoint...

- Linked resources allow for more complex mappings between resource tree and file system
- More flexibility added over the years based on community demand
- Introduces problem of overlapping resource regions and aliases

Persistence

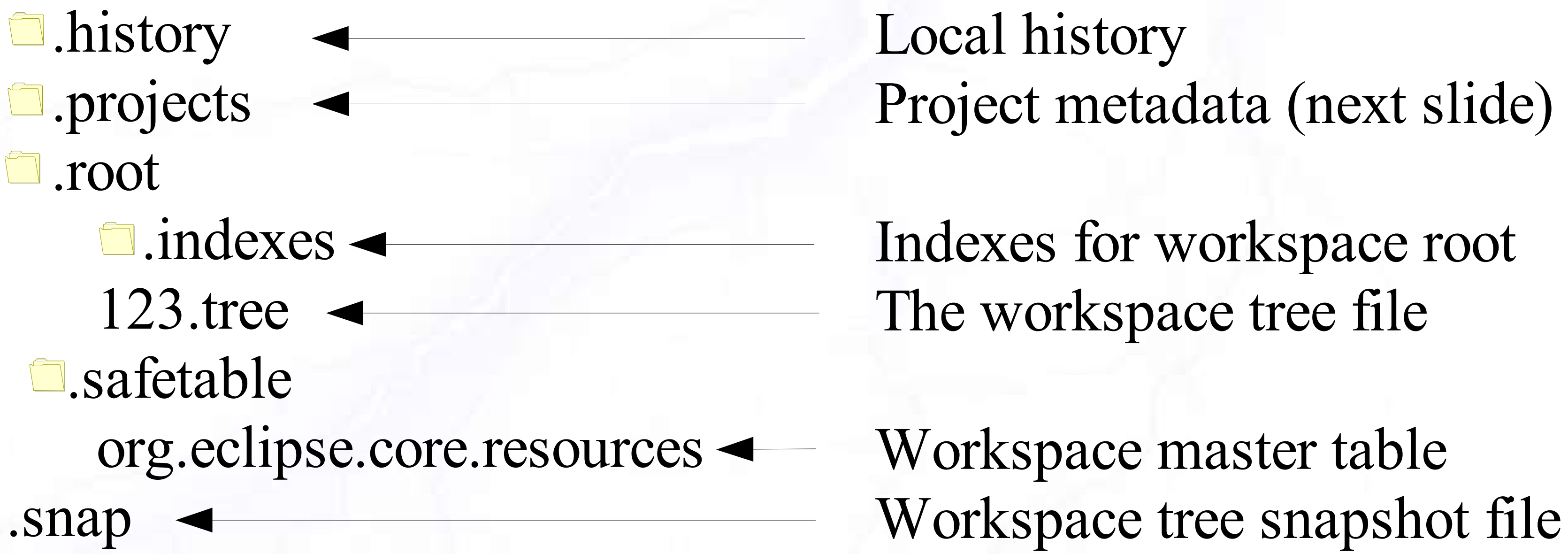
John Arthorne
IBM Rational

Basic principles

- Should be able to unplug computer at any moment and be able to restart
- State on disk is always consistent
- Critical state written to disk eagerly
- State that can be recomputed written less frequently
- State stored at project granularity to support closing projects, and facilitate project renames: only store project-relative paths

Workspace metadata disk layout

Under workspace/.metadata/.plugins/org.eclipse.core.resources:

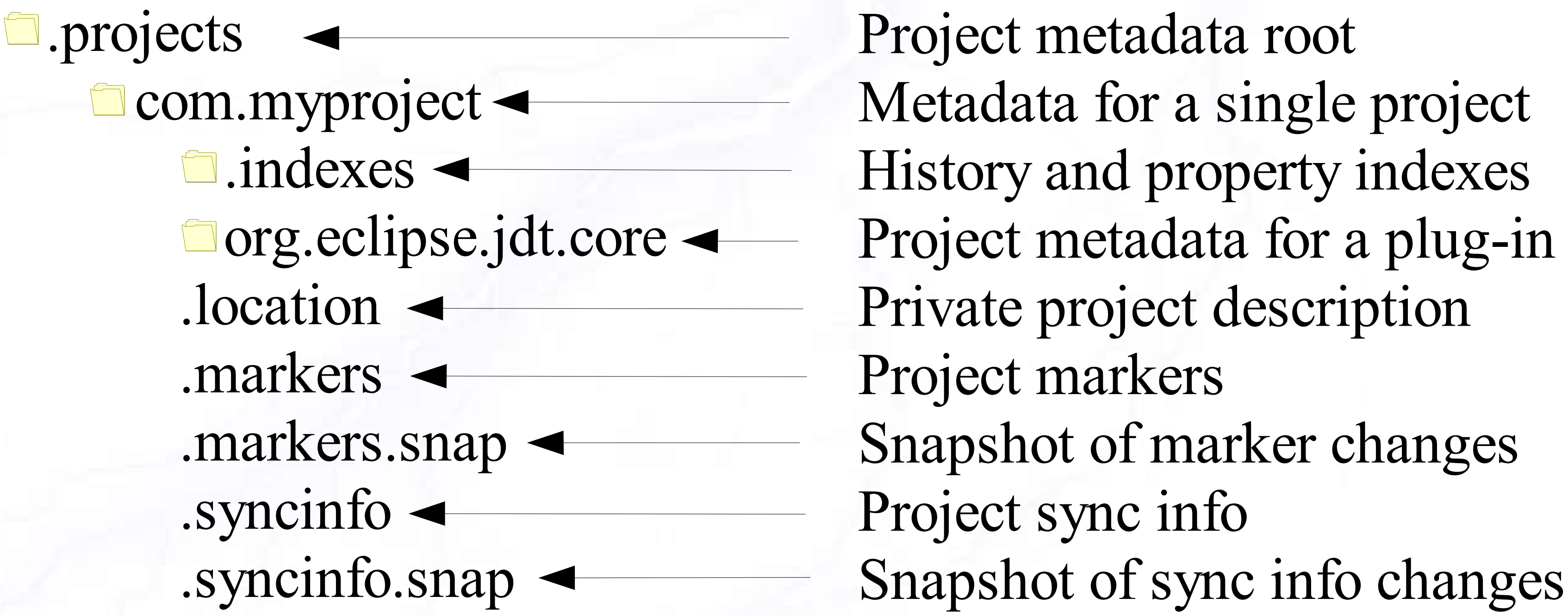


.history	Local history
.projects	Project metadata (next slide)
.root	
.indexes	Indexes for workspace root
123.tree	The workspace tree file
.safetable	
org.eclipse.core.resources	Workspace master table
.snap	Workspace tree snapshot file

Indexes are for local history and persistent resource properties

Project metadata disk layout

Under workspace/.metadata/.plugins/org.eclipse.core.resources:



- .projects ← Project metadata root
  - com.myproject ← Metadata for a single project
    - .indexes ← History and property indexes
    - org.eclipse.jdt.core ← Project metadata for a plug-in
    - .location ← Private project description
    - .markers ← Project markers
    - .markers.snap ← Snapshot of marker changes
    - .syncinfo ← Project sync info
    - .syncinfo.snap ← Snapshot of sync info changes

Writing files

- Each critical state file is written in steps:
  - Write new state to backup file
  - Delete real state file
  - Copy backup file to real file
  - Delete backup file
- At any moment, either the real file or the backup file is valid
- Reading in steps:
  - Attempt to read real file
  - On failure, attempt to read backup file
- SafeFileInputStream / SafeFileOutputStream

Writing the tree file

- Tree file can be very large, so this copying approach too expensive
- Writing tree:
  - Increment tree counter, write file with new tree counter
  - Record new tree counter in “master table” using safe writer
  - After successful save, delete old tree files
- Master table also used for other miscellaneous persistence state related to the tree

Workspace save API

- Saving workspace is a client responsibility
- Clients can also request a fast incremental save (snapshot)
- Workspace does snapshot itself based on policy:
  - Every project creation/deletion
  - Every five minutes (configured via preference)
  - Every 100 non-trivial workspace operations

Tree snapshots

- Each snapshot records changes since previous
- Snapshots appended to the same file in chunks
- On restore, successively read each well-formed chunk from snapshot, and apply delta to tree
- Chunks in file delineated with special bytes
- SafeChunkyInputStream/OutputStream

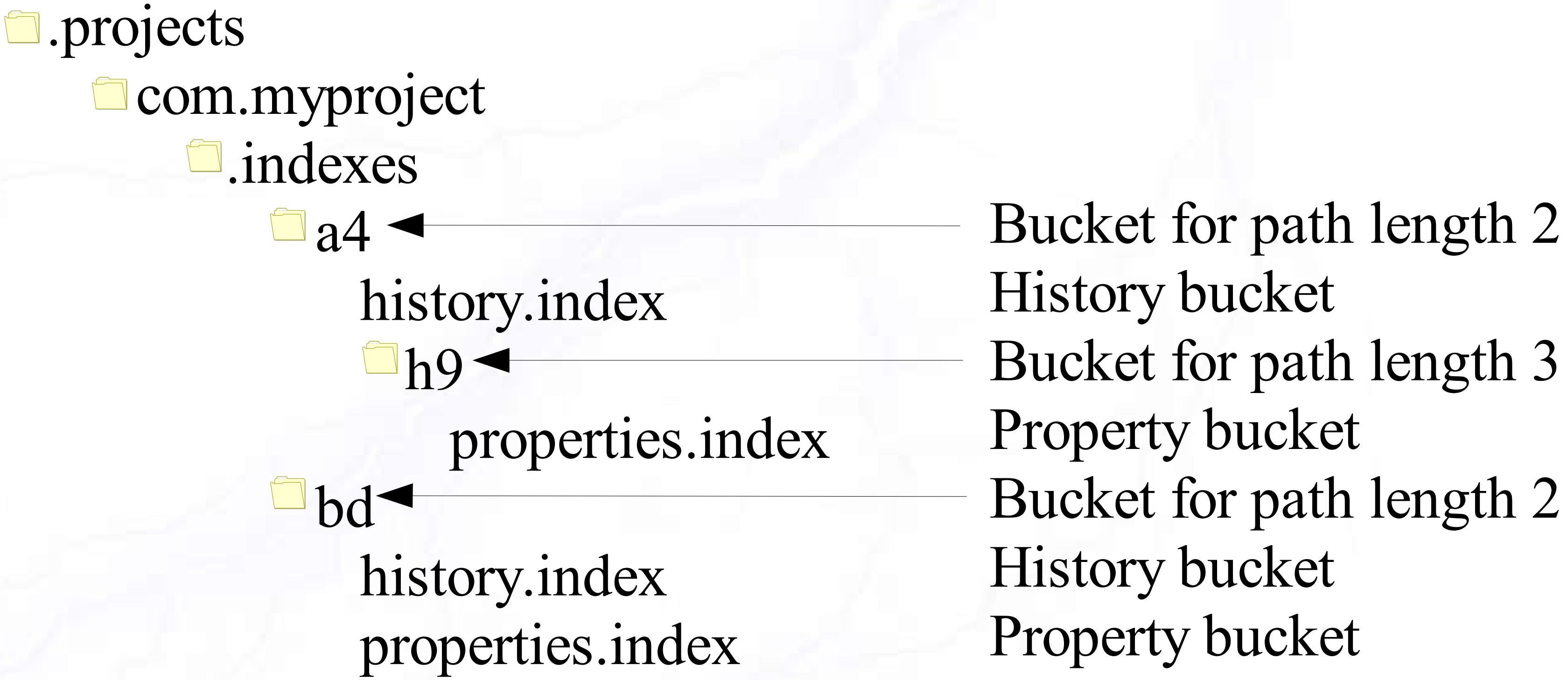
BucketTree

- Local history and persistent properties stored immediately on disk
- Implemented by BucketTree and Bucket
- Bucket tree stores key/value pairs according to folder path
- BucketTree hierarchy on disk mirrors path hierarchy, but using two-digit hash of each path segment (different folders may share a bucket)

BucketTree

- Each bucket is a separate file on disk
- Values in each bucket sorted, binary search used to look up entries
- For properties, the “value” stored is the actual property value
- For history, the “value” is the UUID of an entry in the history blob store

BucketTree disk layout



```
.projects
    com.myproject
        .indexes
            a4            <----------  Bucket for path length 2
                history.index          History bucket
                h9        <----------  Bucket for path length 3
                    properties.index   Property bucket
            bd            <----------  Bucket for path length 2
                history.index          History bucket
                properties.index       Property bucket
```

BlobStore

- Blob = Binary Large OBject
- Stores (UUID->Blob) pairs
- Each blob stored in a separate file
- Organized into folders based on first two chars of UUID
- Blobs can be arbitrary length
- Used to store file history

Workspace description

- Historically workspace description written to xml file in workspace metadata (.workspace)
- Migrated to storing workspace settings in preference store (WorkspacePreferences)
- Some preference values cached in memory for performance reasons

Project description

- Historically project description written to xml file in workspace metadata location (.prj)
- Later moved into project content area as .project file to facilitate project interchange
- Some parts of project description stay in metadata (.location file)
  - Project location
  - Dynamic project reference

Core tools

- Core tools has a metadata browser view
- Point it at a workspace metadata location, and browse contents of any file
- Extension point for adding support for browsing other metadata files

Performance

John Arthorne
IBM Rational

Performance principles

- Can never be fast enough or small enough
- Optimize common code paths
  - Example: Edit/Save/Compile cycle
- Make costs proportional to magnitude of change rather than size of workspace
- Heavily optimize code that must traverse entire workspace
- Don't penalize common cases to handle fringe cases

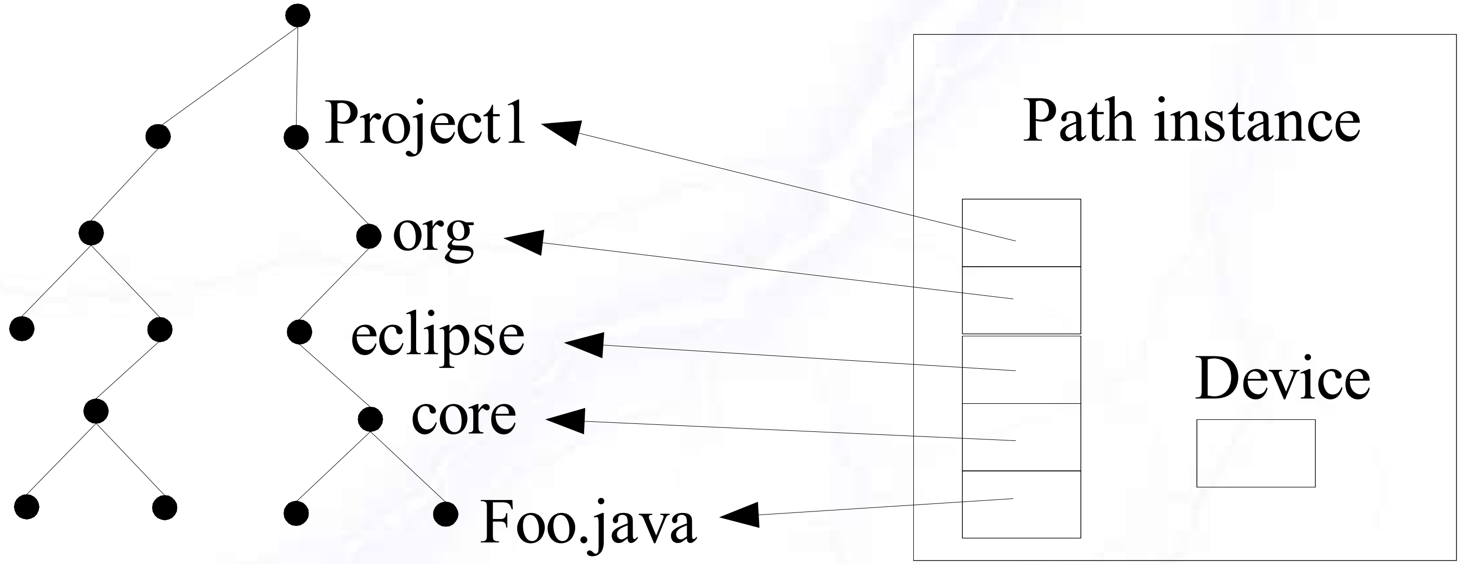
Path class: problem

- Original representation just stored two strings: path + device
- This is a memory-efficient representation
- However, most common operation on Path is to iterate over its segments
- We found during tree lookup, significant amount of time was taken by String garbage

Path class: solution

- Now represented as an array of strings for segments, a device string, and a bit-mask int
- More memory-intensive, but zero garbage creation during path traversal, tree lookup
- Store very few paths so the performance gain outweighed the memory footprint
- For workspace tree paths, segment strings taken from tree nodes, so no strings created
- Worse performance for Path.toString()

Path class example



- Also use Path.segment(int) to iterate: no garbage
- Lesson: smaller isn't always better – optimize for common usage

Resource tree look-up

- Original resource tree didn't order children
- Tree look-ups required linear search over children
- Changed to sorted children and binary insertion of new nodes
- Look-up changes from O(n) to O(log(n))
- Lesson: algorithms matter

Resource tree look-up

- Found very common pattern of multiple queries on same item (locality of reference)
- Adding a tree look-up cache of just one element resulted in significant speed-up of real world scenarios like searches and builds
- Lesson: caches don't have to be fancy, they just have to be tuned for usage patterns

Fast loops

- Delta trees are built out of arrays
- Lots of array iteration and manipulation in critical performance paths
- We discovered writing loops backwards was much faster (compare with zero is typically one chip-level instruction, whereas comparing a field value is more expensive

```
for (int i = children.length; --i >= 0;)
    names[i] = children[i].getName();
```

Fast loops

- BUT: The old assumptions no longer hold:

	Normal loop	Reverse loop
IBM Java 4 sr9	938ms	984ms
IBM Java 5 sr4	1375ms	2609ms
IBM Java 6 sr3	4468ms	735ms

- Lessons:
  - Retest your assumptions.
  - JIT-style optimizations rarely hold across VMs.
  - Benchmark real scenarios with real VMs.

Resource traversal: problem

- Resource model doesn't hold onto IResource objects
- We need to instantiate IResource handles every time a client traverses the tree
- The overhead of creating/gc'ing those handles was a big chunk of the traversal cost
- We found that most visitors are only interested in a small number of resources

Resource traversal: solution

- Pass a resource proxy object to visitor instead of real resource
- Create real IResource only if requested by visitor
- Can update singleton proxy object with new resource path after each visit
- Traversal up to 23x faster using proxies (27948)
- Lesson: Sometimes a special-purpose variant of an object is needed (in this case to allow for a mutable proxy with bounded lifetime)

Bloated data structures: problem

- Java collection classes very high quality, but any general-purpose implementation needs to make design trade-offs
- In general Java collections optimized for read speed over write speed and memory overhead
- Not well suited to large numbers of relatively stable instances with small set of values
- For example, 100,000 HashMaps, each typically containing 1-10 items

Bloated data structures: solution

- Custom data structures with different design parameters: small typical size and memory efficiency
- ObjectMap: map backed by a single array that alternates keys and values. No hashing.
- KeyedHashMap: map backed by a single array. Uses hashing and linear probing for collisions
- MarkerSet, MarkerAttributeMap: custom set and maps for storing markers and values
- Use array over ArrayList where valuable

Bloated data structures: solution

	ObjectMap	HashMap
1 element	48 bytes	150 bytes
5 elements	128 bytes	502 bytes
10 elements	228 bytes	936 bytes

- But space isn't everything!
- HashMap has much better lookup performance in a large map

Hashing: problem

- NodeIDMap tracks resource moves. Map of node id -> (old path, new path)
- Array-backed hashing map, using linear probing
- Original hash algorithm was (nodeId% table.length)
- Horrible hash performance with many changes
- Worst case: run several days, crash, delete 60,000 files, restart (bug 30342)

Hashing: solution

- Hash resulted in O(n^2) worst case performance: 9.6 billion comparisons for 60,000 changes
- Changed to Knuth's multiplicative hash function (multiply by large prime)
- Prime table sizes to improve hash
- Startup time cut from 106s to 16s
- Lesson: algorithms matter, worst case will always happen eventually

ResourceInfo: problem

- The one big object that has an instance per resource
- Size of this object is very important!
- Holds onto all interesting state about a resource
- 72 bytes per instance with traditional fields

ResourceInfo: solution

- Pack fields together when full range of values not needed
- Down to 64 bytes per instance
- Also use null for empty collections
- Similar technique used in path to merge hash code into same field storing leading/trailing slash data
- Could also use specialized classes: ResourceInfoNoMarkers, etc

Overall Lessons

- End users will always take things 10-100x further than you imagined: 10x more resources, 10x slower disks, 100x larger files
- Optimize for real world scenarios using stopwatch timing
- Constant tension between speed and space optimization
- The stuff you learned back in algorithm and data structure courses matters!

Introducing EFS

John Arthorne
IBM Rational

EclipseCon 2009

Resources circa 2004

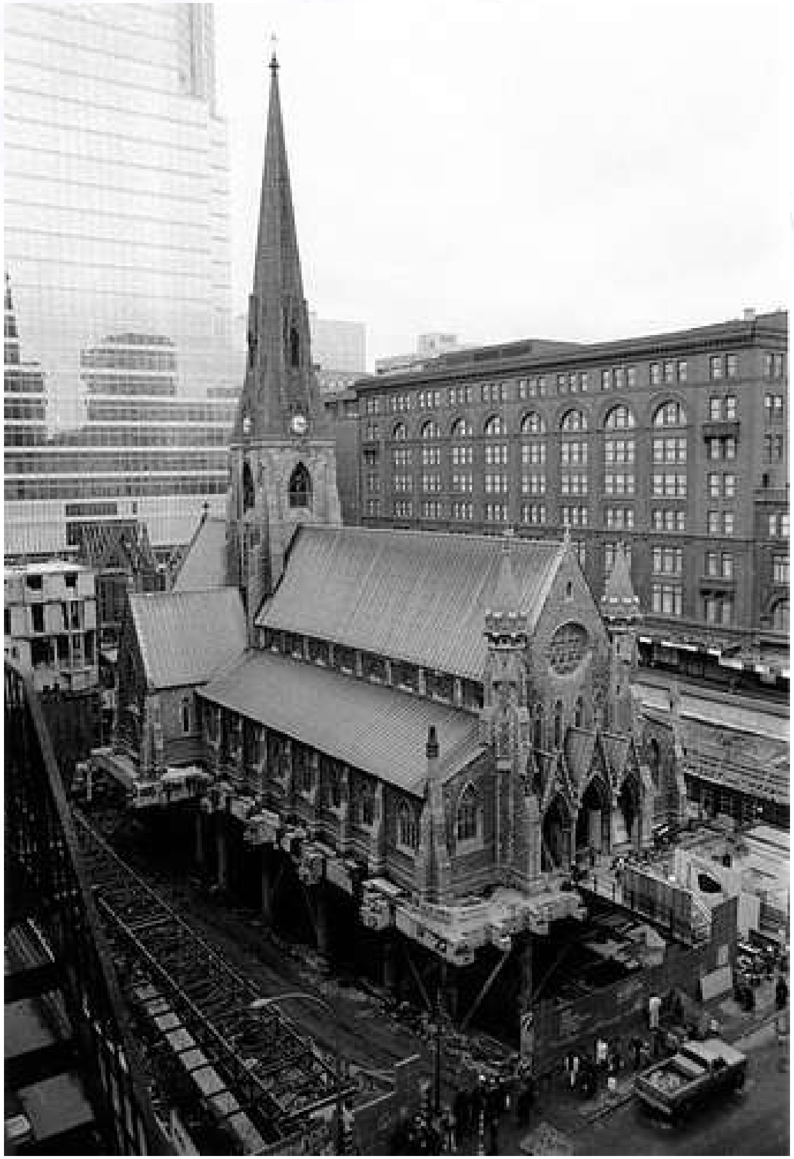
- Sacred: must not break
- Limited: local file system only
- Christ Church Cathedral, Montreal

How it used to work

- Resource implementation used a combination of java.io.File and home-brewed natives for:
  - Getting/setting file attributes
  - Finer granularity of file timestamps
  - Getting multiple values from the file system with one native call
- Mostly isolated to a single FileSystemStore class, but other uses of java.io.File scattered around

Lifting the floor

- Challenge was to unlock new potential without breaking the existing structure
- Want to slide a new layer underneath that abstracts away the file system
- A shopping mall under a church



Exploring the options

- Apache commons VFS
  - Broad set of platforms but very shallow integration
- KDE Input/Output (KIO)
  - Both synchronous and asynchronous variants
  - Very cool but not in Java
- Java file system API
  - JSR 51 -> JSR 203
  - May work when it arrives (8 year wait so far)

EFS design principles

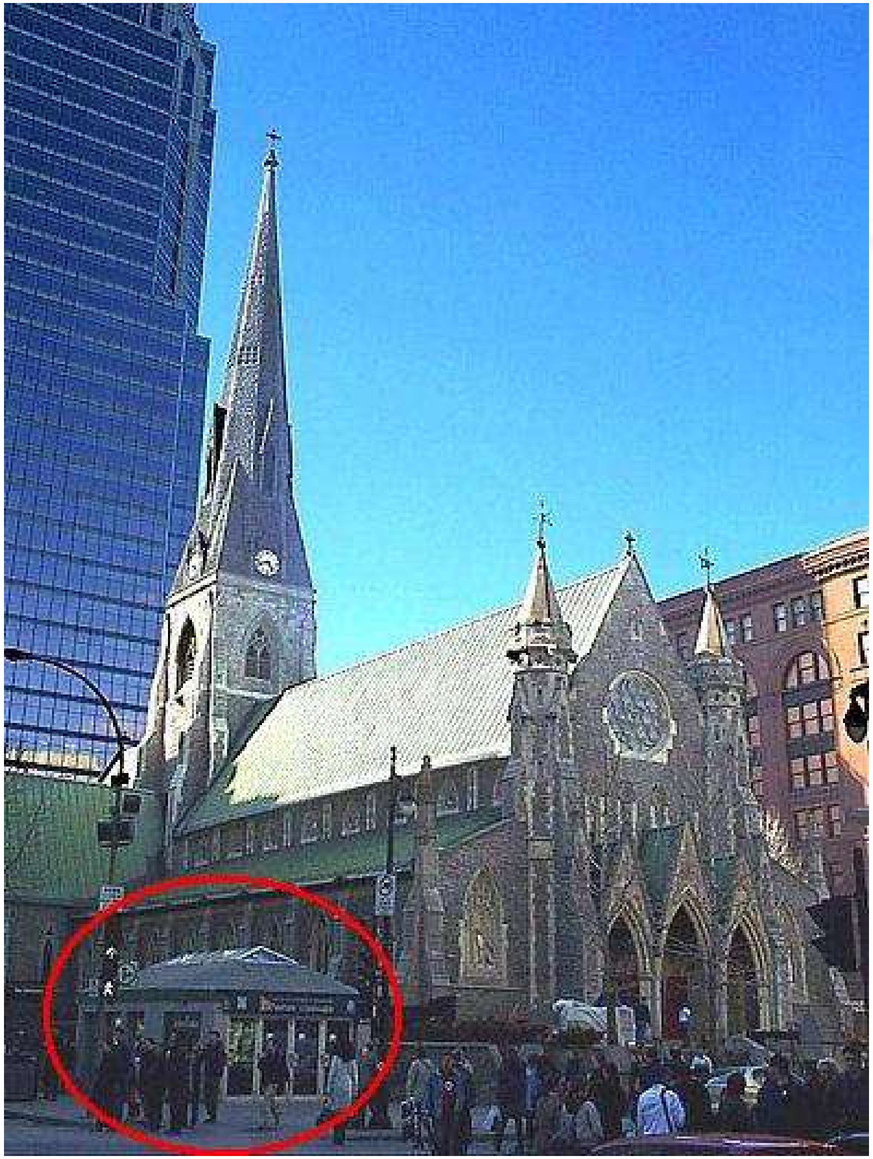
- Small, simple replacement for java.io.File
- Stateless
- Honour local file system behaviour as much as possible
- Add progress monitors, cancelation, better error reporting
- Models fast, highly available, tree-based file systems very well

How it hooks in

- Certain resources designated as “file store roots” (projects or linked resources)
- Resource sub-tree below each file store root assumed to mimic the EFS
- Typical algorithm: Walk up to nearest file store root, get EFS location, append
- File system interactions go through IFileStore
- Avoid using URI except as external location format

End result

- Almost no change to IResource API
- Generally replaced use of IPath with URI when dealing with locations
- To exploit EFS plug-ins must adapt, but existing plug-ins will continue working within old limitations



Lessons

- Building it was not enough
- API not created in conjunction with real world implementations
- Took years for plug-ins to adapt to EFS
- There was no “killer app” to encourage plug-ins to convert quickly
- There are now many implementations, and plug-in authors are adapting to it
- Silver lining: greatly streamlined interaction with f/s, centralized workarounds for flakiness of java.io.File

Concurrency

John Arthorne
IBM Rational

Basic principles

- Lock on write, no locks on read
- Use immutable objects as much as possible:
  - Tree nodes
  - Immutable trees for all states other than current
  - Immutable maps for properties, markers, etc
- Copy on write, to allow non-locking concurrent reads

Tree for mutable state

- All mutable workspace state stored in the tree
  - Resource state
  - Project and workspace description
  - Builder state
- Data in immutable trees copied into open tree on first write in operation
- Writes in workspace tree protected by single workspace lock

Workspace Lock

- Workspace lock only held internally when updating tree
- Lock never held when calling client code, with single exception of resource change events (other threads trying to modify tree at that point are blocked anyway)
- Lock is “fine-grained” - never held for extended periods to allow for concurrent modifying operations in multiple threads

Workspace Lock

- Lock management found in WorkManager
- Acquire: Workspace.prepareOperation
- Release: Workspace.endOperation
- Precondition checking done after acquiring lock
- Calls to third party code surrounded with WorkManager.begin/endUnprotected

Scheduling rules

- Until Eclipse 3.0, we just had the workspace lock for any modifying operation
- Lock often held for long periods, while calling client code
- Result was zero concurrency, poor application responsiveness

Scheduling rules

- Scheduling rules introduced as client-facing notion of resource locking
- Clients can “lock” portions of the workspace using the corresponding scheduling rule using IWorkspace.run
- Obtaining resource rule locks all children
- Clients now never hold workspace lock (except during POST_CHANGE events)

Rule factories

- How to specify scheduling rule requirements for various workspace-modifying operations?
- Want to allow freedom to change actual rules
- Those pesky back doors for VCM systems means they need to be able to override rules

Rule factories

- IResourceRuleFactory – abstracts rules used for particular resource change operations
- For complex operation can combine multiple rules with MultiRule
- Via TeamHook, VCM system can set rule factory for a given project

Copy on write

- Tree nodes copied into mutable layer when modified
- On node create/delete, parent nodes are copied
- This happens “for free” via delta tree representation

Copy on write

- Data structures never modified once reachable
- Reads far more frequent than writes
- Write methods still synchronized to ensure propagation of thread-local caches

```
ObjectMap temp = sessionProperties;
if (temp == null)
    temp = new ObjectMap(5);
else
    temp = (ObjectMap) sessionProperties.clone();
temp.put(name, value);
sessionProperties = temp;
```

Copy on write - readers

- Readers don't require synchronization at all
- Just need a stable reference

```
public Object getSessionProperty(QualifiedName name) {
    Map temp = sessionProperties;
    if (temp == null)
        return null;
    return temp.get(name);
}
```

Locking problem: aliases

- Scheduling rules require strict rule tree
- Otherwise have deadlock or multiple threads owning same lock
- Resource tree is a strict tree, so it maps well to rule requirements
- However resources can overlap in the file system, so multiple threads can “lock” resources that share same location

Locking problem: aliases

- Resource rules protect against other threads owning overlapping rule
- File system can still change out from under you
- This is an open problem with no good solution

Summary

- Lock on write, no locks on read
- Use immutable objects as much as possible
- Copy on write, to allow non-locking concurrent reads
- Single workspace lock to protect tree
- Scheduling rules add client-facing mechanism for managing concurrent modifications