eclipse CON 2004

Writing responsive UIs using the Eclipse 3.0 concurrency architecture

John Arthorne, Eclipse Platform Core
Jean-Michel Lemieux, Eclipse Platform Team

 February 5, 2004

What we'll cover

- Overview of UI responsiveness problems in Eclipse 2.1
- Underlying factors affecting responsiveness
- Enumerate responsiveness problems and what we did about it
- Pointers to new API classes and methods in 3.0
- How to make your own plug-ins more responsive

What do we mean by responsiveness?

- Responsiveness is not the same as performance
- Performance: how much work can be done in a given time
- Responsiveness:
  - Feedback: How often is the user interface displaying accurate information and how quickly is user interaction acknowledged
  - Availability: How often is the user interface in a state where the user can interact with it
- Responsiveness and performance are sometimes mutually exclusive. To keep the UI continually updated and in an interactive state affects performance. Allowing the user to do two things at once affects the performance of both.

Responsiveness in Eclipse 2.1

- Responsiveness focus was on feedback
  - Busy cursors and progress indicators
  - Support for canceling long running operations
- Less focus on increasing availability
  - Availability is often viewed simply as a performance problem
  - Simpler programming model: no major worries about contention, deadlock, thread-safety, etc

Symptoms of poor responsiveness: feedback

- Long running operation run in the UI thread
  - The UI seems locked (e.g. menu bar doesn't repaint)
  - The user doesn't know what is happening
- Progress dialog appears but cancel button is disabled or doesn't do anything.
  - A progress dialog that doesn't show accurate progress can also be confusing and give the impression that the application is stuck
- Buttons that don't respond to clicks in the required 0.1 second (so we click them again, and again, and again…)

Symptoms of poor responsiveness: availability

- Too many blocking progress dialogs
- Delay when typing in an editor
- Delay opening views and editors

Examples of poor responsiveness

- Open a view or editor – ***wait*** while the part's model is initialized and then finally shown.
- Contributed decorations – ***wait*** while decorators doing expensive calculation.
- Check out a project from CVS – ***wait*** until entire check out is complete before browsing files.
- Save a document - ***wait*** until auto-build is complete before continuing, even if you didn't need it to be built immediately.

Examples of poor responsiveness (cont…)

- Synchronize with repository – ***wait*** while it refreshes all projects in the workspace before you can start browsing changes and doing commits and updates.
- Launching an application for debug/running – ***wait*** for debug connection before doing anything else
- Task view updating and filtering – ***wait*** while views are updating and filtering every time a marker is generated.
- Type a character in an editor – ***wait*** for editor to validate model.

Approach for solving feedback problems

- Remove long running operations from the UI thread
  - Use new IProgressService.busyCursorWhile(IRunnableWithProgress)
  - Caveat: moving processing to another thread can mean breaking changes to your plug-in
- Progress dialogs must honor cancellation
  - Plug-ins **must** provide responsive cancellation by checking monitor.isCanceled()
- Update UI more often during operations

Approach for solving availability problems

- Give control back to the user as soon as possible (quick initialization)
- Offload expensive processing to background thread
- Start with the worst offenders instead of trying to solve everything at once
- Use a work queue to combine/buffer background processing

Problem 1: lack of common infrastructure

- In a pluggable environment, nobody has complete knowledge of what is happening.
- Hard to coordinate work to avoid bottlenecks and prevent deadlock.
- Leads to thrashing when everyone tries to do something at once
- Need a common infrastructure so tasks can describe what resources they need, how important they are, what other tasks they depend on, etc

Solution 1: new concurrency infrastructure

- Job API: org.eclipse.core.runtime.jobs
- Job: a unit of work scheduled to run asynchronously
- Why not just java.lang.Thread?
  - Lighter weight: uses a shared thread pool
  - Support for progress feedback and cancellation
  - Priorities and mutual exclusion
  - Richer scheduling: run now, run later, run repeatedly
  - Job listeners can find out when jobs start, finish

Problem 2: how does the user know what is happening in the background?

- UI design principle: when user initiates action, there should be feedback within 0.1 seconds
- What feedback is appropriate for background operations?

Solution 2: add job progress support

- Jobs report progress via the IProgressMonitor passed to the run() method.
  - The UI provides the actual monitor and can display the Job. The following information can be shown: job name, task name, sub tasks, percent complete.
- System jobs aren't shown to the user
- Progress animation hint
  - workbench icon animation
  - view specific progress indicator (e.g. busy/wait cursor and view animating icon)

Solution 2: add job progress support (cont…)

- Errors occurring in the background are important enough to interrupt the user.
- Progress groups: for grouping related jobs
- Prompt for information without interrupting the user
  - IProgressService.requestInUI

Problem 3: UI is often in a transient state

- Background jobs can invalidate the state of things shown to the user
- The UI can be stale while the user is interacting with it
  - Click on an error marker while the file is building
  - Expand a folder that takes a while and start browsing another portion of the structure
  - Resource in the Synchronize View is being committed to repository
- The UI can be changing as the user interacts with it
  - Disco ball syndrome

Solution 3: transient state affordances

- WorkbenchPart will provide default progress hint if job is scheduled from a view or editor.
- Gray items that are transient or stale (examples: problem markers in Java files, resources in Synchronize View)
- DeferredTreeContentManager: Infrastructure for lazy population of trees shows hint that children are pending when a node is expanded.
- Other plug-in specific feedback should be avoided if possible to ensure that the UI doesn't become too distracting (e.g. progress monitor shown in every view)
- Batch updates to prevent flashing using UIJob

Problem 4: deadlock

- More things in the background means greater chance of deadlock
- SWT syncExec is especially deadlock prone
- Many Eclipse APIs with hidden locks: resources, text, JDT, CVS
- No chance of resolving deadlock since every component rolled their own locking infrastructure

Solution 4: locks and scheduling rules

- ILock: re-entrant lock, like Java object monitors, except:
  - Fairness: acquires are granted in FIFO order
  - Aware of syncExec: carries over to UI thread
- ISchedulingRule: a scheduling rule can be attached to a Job
  - Specifies when it is safe to run the job
  - A job will not start while any job is running that has a scheduling rule that conflicts with its scheduling rule
  - Implemented by clients – every plug-in can define their own rules
  - Rules can be nested to form rule hierarchies

Solution 4: locks and scheduling rules (cont…)

- Avoid hold and wait:
  - ISchedulingRule: allows jobs to specify requirements before they even start (two phase locking).
  - Impossible to hold a rule and be waiting for a rule
  - Avoid holding locks while client code is called
  - Avoid syncExec and use asyncExec where possible
- Avoid circular wait
  - Always acquire locks in a consistent order
- Preemption:
  - If all else fails, preempt the thread that introduced deadlock and report error in log. This happens for free with ILock and ISchedulingRule

Problem 5: increased contention

- Multiple threads compete for resources and cause contention.
- Heavy-handed use of locking constructs makes real concurrency difficult
- Contention can block the user and make responsiveness worse (the user tries to do something but is blocked by some long running operation happening in the background.

Solution 5: smaller locks and contention reporting

- Minimize locking by using fine-grained locks
- ISchedulingRule allows clients to specify hierarchies of locks
- When contention can't be avoided it must be reported to the user
- IProgressService.busyCursorWhile()
  - Uses traditional busy cursor and progress dialog
  - Augments itself when progress is blocked
  - IProgressMonitorWithBlocking allows an operation to report when it is stuck

Concurrency in the resources plug-in

- IResource implements ISchedulingRule
- Changing a resource now only locks a portion of the workspace, allowing concurrent modification of the workspace
- New IWorkspace.run() method uses scheduling rule to avoid locking the entire workspace
- Auto-build (and auto-build events) moved into background job to reduce perceived duration of operations
- Resource changes now broadcast periodically during operations. Example: can view files in a project that is in the middle of being checked out from the repository

Making your plug-in responsive

- Step 0 - If you make no changes, you will be ok (even slightly better)
- Step 1 - Revisit locks to reduce contention with background jobs
- Step 2 – Move long read only operations to background
  - Watch for assumptions about UI thread and thread safety
- Step 3 – Move long writing operations to background
  - Trade-off is added complexity of code versus important responsiveness gains
  - Need to be aware of concurrency requirements of code you're calling: locks acquired, etc
  - Be aware of deadlock risks and know avoidance strategies

Further reading

- Examples plug-in (available soon) org.eclipse.ui.examples.jobs
- http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-core-home/plan_concurrency.html
- *GUI Bloopers*, Jeff Johnson – Chapter 7: Responsiveness Bloopers
- *Concurrent Programming in Java*, Doug Lea
- *Modern Operating Systems*, Andrew S. Tanenbaum

