UI Performance Monitoring
SWT API to Monitor UI Delays
Terry Parker, Google
Contents

1. Definition
2. Motivation
3. The new API
4. Monitoring UI Delays
5. Diagnosing UI Delays
6. Problems Found!
7. Next Steps
Definition

UI Delay: an interrupt of a user’s workflow, caused by mismanagement of the UI thread.

Any work on the UI thread can cause a delay, but small delays do not make the UI unresponsive.

Sometimes moderate delays are acceptable, other times even short delays are disruptive.
Motivation

Would you use an editor with a $\frac{1}{2}$ second delay after each keystroke?

Short delays are notoriously hard to debug, especially when they’re in someone else’s code.
Motivation

It doesn’t take much to degrade the user experience.
API

In Eclipse 4.4, we extended the SWT Event API.

Added 4 new event types to Display:

1. PreEvent is invoked before every event is dispatched
2. PostEvent is invoked after every event is dispatched
3. SleepEvent is invoked before each interruptible sleep
4. WakeupEvent is invoked after each interruptible sleep

http://bugs.eclipse.org/360052
API

All 4 new events are ‘unscheduled’ events.

They are invoked automatically and ‘wrap’ other events.

They are invoked on the UI thread just like any other event.
API

Listeners subscribe to these events just like any other Display events:

```java
Display display = ...;
display.addListener(SWT.PreEvent, new Listener() {
    @Override
    public void handleEvent(Event event) {
        // Do stuff
    }
});
```

The `event` argument for `handleEvent` will always be `null`. 
These event handlers get invoked very, very frequently (1000s to 10,000s of times per second) so they must run fast or risk causing UI delays themselves!

In practice, handlers must do very little most of the time (e.g., check a timestamp) but can take longer when a long event has been detected (e.g., log a message).
The Event Sequence

A long-running event handler can implement its own event loop (e.g., popping up a modal dialog), resulting in recursively nested events.

Monitoring the transitions between states will discover sources of UI delays, no matter what the level of nesting.
Basic Monitoring

// Use event notifications with timestamps
class MonitoringListener implements Listener {
    static long previousEventTransition;
    @Override public void handleEvent(Event event) {
        long now = System.currentTimeMillis();
        if (event.type != SWT.WakeupEvent) {
            if (now > previousEventTransition + 1000) {
                // +1 second UI freeze, log it!
                previousEventTransition = now;
            }
        }
    }
}
Monitoring Deadlocks

// Need to run an independent thread to catch deadlocks
public class MonitoringThread extends Thread implements Listener {
    private final Object sleepMonitor = new Object();
    @Override
    public void run() {
        while (!canceled) {
            sleepMonitor.sleep(FIVE_MINUTES);
            if (NOW > previousEventTransition + FIVE_MINUTES)
                // log the possible deadlock with a stack trace!
        }
    }
    @Override
    public void handleEvent(Event event) {
        sleepMonitor.notify();
        ...
        previousEventTransition = now;
    }
}
Diagnosing Delays

Measuring delay durations is nice, but what is causing them?

If we use a background thread to capture stack traces during long events, we can find out!

This is a *Sampling Profiler*. It takes periodic samples, and those samples can be used to discover problems.
Diagnosing Delays

Conflicting goals: we want a good picture of the sources of UI delays, but we don’t want to blow out memory.

Solution: place an upper-bound on memory usage
- Start sampling at fixed intervals
- After reaching a maximum number of samples, throw out some and continue sampling at a reduced frequency

Yields a reasonable number of traces, evenly distributed in time.
Stack Sampling

Keeping samples even in time:

- Collect samples at a fixed rate
- After 8 samples, toss out every other sample and continue at half the frequency
- Continue collecting samples
- After 8 samples, toss out every other sample and continue at half the frequency
- Continue collecting samples
private void handleLongEvent(Event event) { // called on UI thread
    longEvent = new LongEvent(lastEventTransition); // event start & end
}

@Override public void run() { // runs on monitoring thread
    while (!canceled) {
        sleepMonitor.sleep(delay);
        samples.add(new Sample());
        if (longEvent != null) {
            logEvent(longEvent, samples, sampleCount);
            longEvent = null; samplesCount = 0;
        }
        if (sampleCount > 4) {
            evictEveryOtherSample(); // updates samples & sampleCount
            delay *= 2;
        }
    }
}
Problems Found - JDT

Using our sampling profiler plugin, we’ve identified a few substantial performance improvements in JDT:

- A major portion of time spent on checking for external folders while resolving a classpath
  [http://bugs.eclipse.org/411423](http://bugs.eclipse.org/411423)
- Rerunning a test from JUnit results view is launching the test on UI thread
  [http://bugs.eclipse.org/411841](http://bugs.eclipse.org/411841)
- New JavaModel caches performance improvement
  [https://bugs.eclipse.org/421165](https://bugs.eclipse.org/421165)
Case Study - Eclipse

Other UI performance issues

- Property testers that used JavaModel rather than IAdaptable to determine if the UI element corresponds to a Java file
- C++ Codan loading of CDT bundles on a non-C++ project triggered bundle loading due to use of ‘instanceof’
What’s Next?

Use this API in a way that benefits the Eclipse ecosystem
- Summer intern project to work on a “UI Performance Problem” report generator for end users

Proposal to support parallel execution in the Jobs API
- Look at speeding up things like text searches, searching for Java types, F5 refresh (if feasible)

Come to the BOF tonight!
Evaluate This Session

1. Sign-in: [www.eclipsecon.org](http://www.eclipsecon.org)

2. Select session from schedule

3. Evaluate: +1 | 0 | -1