Team Explorer HD Core Event Policy

Shaw Terwilliger

December 1, 2009

Describes the design, implementation, and intended patterns of use of events in com.microsoft.tfs.core (core). This document was written at Teamprise, not Microsoft. It is probably a little out of date with respect to proper names, but the philosophy is still sound.

1 Overview

Users of core (plug-in, CLC, SDK applications, etc.) want to be notified of some things that happen as side effects of high-level operations they intiated. Core uses an eventing design to deliver these notifications to interested parties, which are generally end-user applications, but may be other core components.

Eventing implementations vary considerably in different software systems. This document describes how to safely (from a thread-safety and completeness point of view) use the events offered by core, and also how to write core code that fires these events.

2 EventEngine

The EventEngine class is the heart of event registration and dispatch. Each *SourceCodeControlClient* instance has a publically accessible *EventEngine* instance through which events fired by its methods (or by objects created that depend on it) are routed. *WorkItemClient* does not currently use an *EventEngine* (it fires no events), but may in the future.

To register for events from a *SourceCodeControlClient*, get a reference to its *EventEngine* (*getEventEngine()*) and call one of the "add listener" methods. If you wish to stop receiving events, call the corresponding "remove listener" method.

2.1 Events are Synchronous

Core fires events by calling one of the "fire" methods in *EventEngine*, which iterates over the registered listeners and invokes a method on each synchronously. The "fire" method only returns control to core once each handler has returned.

2.2 Avoid Thread Deadlock

Events are synchronous. If your event handler does a thing that requires some other thread to make progress before it can complete, and that other thread is waiting for some event fired only after your handler completes, you will have thread deadlock. *EventEngine* is hard to deadlock on itself. All *EventEngine* events are dispatched without holding any locks on the engine or its data, but other locks on core objects (AWorkspace, etc.) when events are fired may prevent event handlers from calling back into certain regions of core.

2.3 Events can be Fired from any Thread

All event handlers are run on the thread which fired the event, and this thread may *not* be the user's thread that initiated the core work. For instance, *AWorkspace.getItems()* may start multiple threads to accelerate the file transfer process. If one of these worker threads encounters a warning, it fires an event through GetEngine, and event listeners will have their handlers invoked in the context of the worker thread.

It is the responsibility of the event listener to do the correct marshalling of event data info the context where it should process the event. To aid the listener in identifying exactly what thread/command/action/method caused the event, every core event includes an *EventSource* instance in its data. See section 3.

3 Event Sources

The *EventSource* class contains information about the context (where and when) the event was fired. Currently only the thread that reached the *origination point* is included in the source data, but other information may be added later. Listeners can filter events they receive to take interest only in the ones that were caused by one of their worker threads.

3.1 Event Origination Points

A *core event origination point* is a specially-documented method in core code where once execution flow enters, all events fired before execution returns from that method will have an *EventSource* that describes the execution context where and when the method was invoked. All methods that exhibit this behavior have Javadoc text describing them as "core event origination points."

All core code that fires events via *EventEngine* must document their event origination points. Only public methods are required to carry this information.

3.2 Multiple Origination Points

If an execution path through core code enters more than one origination point, it is the context of the *first* execution point reached that is sent with the events.

This behavior comes "for free" in most core code. Most public methods on core classes simply call some private methods, create some local objects, and do all their event firing in the same thread that invoked them. However, in the case where a core method creates new threads that might fire events, or for another reason fires events in an execution context that is no longer matches the first origination point, the first event source information must be used for all events. See *GetEngineDownloadWorker* for an example of how a new thread is passed the original context information.