threads eagerly execute all possible jobs that have no unresolved dependencies. Since the execution of concurrent flow graphs is inherently undeterministic, it is very unlikely that a manually defined order is flexible enough to be the most efficient. A scheduler can adapt much better here. Computer scientists tend to disagree with this thesis, whereas economists, who are more used to analyzing stochastic systems, often support it.

Priorities can be used to influence the order of execution as well. The priority system used is quite simple: of all integer priorities assigned to jobs in the queue, the highest ones are first handed to an available worker thread. Since the job base class is implemented in a way that allows writing decorators, changing a job's priority externally can be done by writing a decorator that bumps the priority without touching the job's implementation. The combination of priorities and dependencies can lead to interesting results, as will be shown later.

Instead of relying on direct implementations of queueing behavior, ThreadWeaver uses queue policies. Queue policies do not immediately affect when and how a particular job is executed. Instead, they influence the order in which jobs are taken from the queue by the worker threads. Two standard implementations come with ThreadWeaver. One is the dependencies discussed earlier. The other is resource restrictions. Using resource restrictions, it can be declared that of a certain subset of all created jobs (for example, local filesystem I/O-expensive ones), only a certain amount can be executed at the same time. Without such a tool, it regularly happens that some subsystems get overloaded. Resource restrictions act much like semaphores in traditional threading, except that they do not block a calling thread, and instead simply mark a job as not yet executable. The thread that checked whether the job can be executed is then able to try to get another job to execute.

Queue policies are assigned to jobs, and the same policy object can be assigned to many. As such, they are composed, and every job can be managed by any combination of the available policies. Inheriting specialized policy-driven job base classes would not have provided such flexibility. Also, this way, job objects that do not need any extra policies are in no way affected by a possible performance hit of evaluating the policies.

Declarative Concurrency: A Thumbnail Viewer Example

Another example explains how these different ThreadWeaver concepts play together. It uses jobs, job composites, resource restrictions, priorities, and dependencies, all to render wee little thumbnail images in a GUI program. Let us first look at what operations are required to implement this function, how they depend, and how the user expects to be presented with the results. The example is part of the ThreadWeaver source code.

In this example, it is assumed that loading the thumbnail preview for a digital photo involves three operations: to load the raw file data from disk, to convert the raw data into an image representation without changing its size, and then to scale to the required size of the thumbnail. It is possible to argue that the second and third steps could be merged into one, but that is (a) not the point of the exercise (just like streamed loading of the image data) and (b) would