File loaders simply apply the policy in their constructor or, if generic classes are used, when they are created:

```
fileloader->assignQueuePolicy( resourceRestriction() );
```

But that still does not completely arrange the order of the execution of the jobs exactly as wanted. The queue might now start only four file loaders at once, but it still might load all the files and then calculate the previews (again, this is a very unlikely behavior). It needs one more tool, and a bit of thinking against the grain, to solve the problem, and this is where priorities come into play. The problem, translated into ThreadWeaver lingo, is that file loader jobs have lowest priority but need to be executed first; image loader jobs have precedence over file loaders, but a file loader must have finished first before an image loader can be started; and finally, thumbnail computer jobs have highest priority, even if they depend on the other two phases of processing. Since the three jobs are already in a sequence, which will make sure they are executed in the right order for every image, assigning priority one to file loaders, two to image loaders, and three to thumbnail computers finally solves the problem. Basically, the queue will now complete one thumbnail as soon as possible, but will not stop to load the images if slots for file loading become available. Since the problem is mostly I/O bound, this means that the total time until the thumbnails for all images are shown is a little more than the time it takes to load them from the hard disk (other factors aside, such as extremely high-resolution RAW images). In any sequential solution, the behavior would likely be much worse.

The description of the solution might have felt complex, so lightening it up with a bit of code is probably in order. This is how the jobs are generated after the user has selected a couple of hundred images for processing:

```
m_weaver->suspend();
for (int index = 0; index < files.size(); ++index)
{
    SMIVItem *item = new SMIVItem ( m_weaver, files.at(index ), this );
    connect ( item,  SIGNAL( thumbReady(SMIVItem* ) ),
              SLOT ( slotThumbReady( SMIVItem* ) ) );
}
m_startTime.start();
m_weaver->resume();
```

To give correct progress feedback, processing is suspended before the jobs are added. Whenever a sequence is completed, the item object emits a signal to update the view. For every selected file, a specialized item is created, which in turn creates the job objects for processing one file:

```
m_fileloader = new FileLoaderJob ( fi.absoluteFilePath(),  this );
m_fileloader->assignQueuePolicy( resourceRestriction() );
m_imageloader = new QImageLoaderJob ( m_fileloader,  this );
m_thumb = new ComputeThumbNailJob ( m_imageloader,  this );
m_sequence->addJob ( m_fileloader );
m_sequence->addJob ( m_imageloader );
m_sequence->addJob ( m_thumb );
weaver->enqueue ( m_sequence );
```