memory buffer to process it and display the results. In an imperative program, where all individual operations are blocking, it is of little complexity:

1. Check whether the file exists and is readable.
2. Open the file for reading.
3. Read the file contents into memory.
4. Process them.
5. Display the results.

To be user-friendly, it is sufficient to print a progress message to the command line after every step (if the user requested verbose mode).

In a GUI program, things appear in a different light because during all of these steps, it is necessary to be able to update the screen, and users expect a way to cancel the operation. Although it sounds unbelievable, even recent documentation for GUI toolkits mentions the "check for events occasionally" approach. The idea is to periodically check for events while processing the aforementioned steps in chunks and update or abort if necessary. A lot of care needs to be applied in this situation because the application state can change in unexpected ways. For example, the user might decide to close the program, unaware that the program checks for an event in the midst of a call stack of an operation. To put it shortly, this approach of polling has never worked very well and is generally out of fashion.

A better approach is to use a thread (of course). But without a framework to help, this often leads to weird implementations as well. Since GUI programs are event-based, every step just listed starts with an event, and an event notifies its completion. In the C++ world, signals are often used for the notification. Some programs look like this:

1. The user requests to load the file, which triggers a handler method by a signal or event.
2. The operation to open and load the file is started and connected to a second method for notification about its completion.
3. In this method, processing the data is started, connected to a third method.
4. The last method finally displays the results.

This cascade of handler methods does not track the state of the operation very well and is usually error-prone. It also shows a lack of separation of operations and the view. Nevertheless, it is found in many GUI applications.

This is exactly where ThreadWeaver is there to help. Using jobs, the implementation will look like this:

1. The user requests to load the file, which triggers a handler method by a signal or event.
2. In the handler method, the user creates a sequence of jobs (a sequence is a job container that executes its jobs in the order they were added). He adds a job to load the file and one to process its contents. The sequence object is a job itself and sends a signal when all its