As a development environment, Eclipse provides valuable features that Emacs lacks. For example, the Java Development Tools plug-ins provide extensive support for refactoring and code analysis. In comparison, Emacs has only a limited understanding of the semantic structure of the programs it edits, and can't offer comparable support.

Eclipse's architecture is nothing if not open-ended, as plug-ins provide nearly all significant functionality. There are no second-class citizens: the popular plug-ins are built on the same interfaces available to others. And because Eclipse gives each plug-in relatively low-level control over its input and display, the plug-in is free to choose whatever Model, View, and Controller best suits its purpose.

However, this approach has a number of drawbacks:

- Eclipse plug-in development is not safe, in the sense attributed to Emacs Lisp code. A buggy plug-in can easily cause Eclipse to crash or lock up. In Emacs, the Lisp interpreter ensures that the user can interrupt runaway Lisp code, and the strong boundary between Lisp code and the Model implementation protects the user's data from the more insidious forms of corruption.
- Because the interfaces that Eclipse plug-ins offer each other are relatively complex, writing an Eclipse plug-in is more like adding a module to a sophisticated application than writing a script. Certainly, these interfaces are what have made Eclipse's features possible, but plug-in authors must pay that price for both ambitious and simple-minded projects.
- A plug-in requires enough boilerplate code that Eclipse includes a plug-in to help people write plug-ins. The Eclipse Plug-in Development Environment can generate code skeletons, write default XML configuration files and manifest files, and create and tear down test environments. Providing a "wizard" to generate boilerplate code automatically can help get a plug-in developer started, but it doesn't reduce the complexity of the underlying interfaces.

The overall effect is to leave Eclipse plug-ins as a rather coarse-grained extension facility. Plug-ins are not suited for quick-and-dirty automation tasks, and they're not especially friendly to casual user-interface experimentation.

This suggests the second question of the three I promised at the beginning of the chapter, one we can ask of any plug-in facility: *what sort of interfaces are available for plug-ins to use?* Are they simple enough to allow the kind of rapid development associated with scripting languages? Can a plug-in developer work at a high level of abstraction, close to the problem domain? And how is the application's data protected from buggy plug-in code?

Firefox

The current generation of sophisticated web applications (Google Mail, Facebook, and so on) makes heavy use of techniques such as *dynamic HTML* and *AJAX* to provide a smooth user experience. These applications' web pages contain JavaScript code that responds to the user's