another. Obviously, modules can't be totally decoupled, or they wouldn't be working together at all!

Modules interconnect in many ways, some direct, some indirect. A module can call functions on other modules or be called by other modules. It may use web services or facilities published by another module. It may use another module's data types or share some data (perhaps variables or files).

Good software design limits the lines of communication to only those that are absolutely necessary. These communication lines are part of what determines the architecture.

Unnecessary coupling

The Metropolis had no clear layering. Dependencies between modules were not unidirectional, and coupling was often bidirectional. Component A would hackily reach into the innards of component B to get its work done for one task. Elsewhere, component B had hardcoded calls onto component A. There was no bottom layer or central hub to the system. It was one monolithic blob of software.

This meant that the individual parts of the system were so tightly coupled that you couldn't bring up a skeletal system without creating every single component. Any change in a single component rippled out, requiring changes in many dependent components. The code components did not make sense in isolation.

This made low-level testing impossible. Not only were code-level unit tests impossible to write, but component-level integration tests could not be constructed, as every component depended on almost every other component. Of course, testing had never been a particularly high priority in the company (we didn't have anywhere near enough time to do that), so this "wasn't a problem." Needless to say, the software was not very reliable.

NOTE

Good design takes into account connection mechanisms and the number (and nature) of inter-component connections. The individual parts of a system should be able to stand alone. Tight coupling leads to untestable code.

Code problems

The problems with bad top-level design had wormed their way down to the code level. Problems beget problems (see the discussion of broken windows in Hunt and Davis [1999]). Since there was no common design and no overall project "style," no one bothered with common coding standards, using common libraries, or employing common idioms. There were no naming conventions for components, classes, or files. There was not even a common build system; duct tape, shell scripts, and Perl glue nestled alongside makefiles and Visual Studio project files. Compiling this monster was considered a rite of passage!