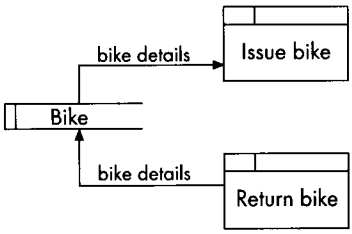

Figure 4.1 *In functional decomposition data (here details about bikes) flows unprotected round the system*

caused unexpected side-effects elsewhere: the infamous 'ripple effect'. Software developers became aware that if the data in the system was accessible to all parts of the code, it could be accidentally corrupted and this was what was causing the ripple effect. Software developers realized that to prevent this, data needed to be encapsulated (protected from unauthorized access).

Poor modularity. However a system is decomposed, it is very important that each separate component or module is self-contained, with a well-defined purpose, and as independent as possible from other modules. Unfortunately, decomposing a system using functional decomposition did not lead to modules with these characteristics. A module in a system based on functional decomposition often was not internally cohesive and was heavily dependent on other modules in the program.

With functional decomposition, making changes to code, either to correct software bugs or implement changes in the user requirements, was a very slow process. In order to understand enough of the code to make the change, preferably without causing side effects, the maintaining programmer had to read a great deal of the code. This was because of poor modularity, the code modules were not independent enough, they were not autonomous. The maintaining programmer could not understand them in isolation, he had to read many peripheral modules to make sense of the one he was changing. This implied that there were hidden dependencies between the modules. What was needed were cohesive (internally coherent) modules that could function either independently or via well-defined interfaces to other modules.

Testing. Software systems based on functional decomposition were still being delivered with errors in the code; they were perceived as unreliable and not sufficiently well tested. It was thought that this might be partly because it was difficult to test units in isolation and manage rigorous integrated testing. This was again partly because units of code were too interdependent. Software developers wanted