software units that could be both easily tested in isolation and logically incorporated into an integration testing schedule.

Software reuse. Reuse of software had never taken off as it was expected to do. One of the principal reasons for this was that a suitable software construct didn't exist. A module that is heavily dependent on other software modules cannot easily be put in a software library – all the bits it depends on have to be in the library also. To be reused, modules need to be independent. Useful library modules must also have a clear single purpose that is general enough to be useful in more than one system.

Data versus function. The structured approach built systems based on their functionality – what the system had to do, the tasks that it had to carry out. The problem with this was that typically the functionality of a system is much more volatile and subject to change than its data. Extensive studies have shown that the most common changes in user requirements are to the functionality of the system. Let us consider, for example, a system for allocating patients and staff to wards in a hospital. We can imagine that over time the ways in which patients are assigned to beds in a ward and the way that the nurses' rota is worked out may change. In addition, it is likely that the system will be required to produce more and different reports for its users. Whatever these changes may be, however, the system will still be dealing with patients, beds, nurses and wards – in other words, the data remains the same. The data is much more stable than the functionality, and is a much sounder basis on which to build a system. In functional decomposition the software architecture (the way the system was divided up) was based on its functionality, and over time the whole structure of the system eventually became unstable. The data in the system, however, remained relatively unchanged over the life-time of a system; it eventually became obvious to the software development community that a system based on data would be more robust.

Advantages of object-oriented development

In order to address the problems associated with functional decomposition, developers needed a software construct that was:

- Autonomous, i.e. did not depend heavily on other modules either explicitly or in obscure ways
- Cohesive with a single, well-defined purpose
- Easy to understand