Fault tolerance techniques for coping with the occurrence and effects of anticipated hardware component failures are now well established and form a vital part of any reliable computing system. However, it is more unusual to find that strategies for fault tolerance have been included in a system for coping with design faults, although such strategies are becoming increasingly common in systems with high reliability requirements. For instance, applications in railway systems, nuclear reactor control and aircraft control are reported by Voges. Design faults may not have been a problem in hardware systems (or at least not recognized as such) but are of major concern in software systems.

Software is a crucial component of all computing systems. Indeed, the major proportion of the complexity of most systems is to be found in the software. Given the evident fallibility of software designers and programmers (some examples are discussed in Chapter 1) and the lack of any technique which can guarantee that a complex software system does not contain residual design faults, it is somewhat surprising that the provision of tolerance to software faults has rarely been considered necessary, even in systems with extremely stringent reliability requirements. One reason for this may have been the absence of suitable fault tolerance techniques, since it is clear that the standard techniques used to cope with (anticipated) hardware component failures are not directly appropriate for the unanticipated situations which result from design faults. The preceding chapters have shown that appropriate
techniques are available (and in many cases are obvious when the characteristics of design faults have been recognized). The purpose of this chapter is to draw together the relevant techniques in order to describe coherent approaches to achieving software fault tolerance in particular, and design fault tolerance in general.

It is worth stating more precisely what is meant by software fault tolerance since two interpretations are possible. One interpretation is that software fault tolerance is concerned with techniques designed to tolerate the effects of faults in the underlying (hardware) interpreter, but which are implemented in software. However, in this book software fault tolerance is considered to embrace all of the techniques necessary to enable a system to tolerate software faults, that is, faults in the design and construction of the software itself. Thus, in the term 'software fault tolerance', 'software' is taken to be a qualification of 'fault' and not of 'fault tolerance'. Since the implementation in software of the fault tolerance techniques discussed in the previous chapters raises few new issues, this chapter will concentrate on the more novel problems of providing tolerance to software faults.

Two main methods have been proposed for structuring a software system, and providing software fault tolerance: recovery blocks, and N-version programming. Because this is an important application for fault tolerance, each of these approaches will be examined in detail. Both approaches make the fundamental assumption that despite all of the fault prevention techniques which may have been used, a complex software system will always contain residual faults when it is put into operation, and software fault tolerance will be necessary if these faults are not to lead to failures of the overall system.

Note, however, that as with any approach for tolerating unanticipated faults, neither recovery blocks nor N-version programming can provide an absolute guarantee that the fault tolerance provided will be successful. Such a guarantee would require identification of the faults before they occurred, coupled with a demonstration that the tolerance provided was effective. Since residual software faults are by their nature unanticipatable this identification cannot be achieved. (Moreover, it seems unwise to use fault tolerance to cope with any software faults which have been identified - such faults should be removed from the system.) As should be the case for any proposed technique for tolerating design faults, the two schemes provide 'last-resort' defences against faults. They should be regarded as complementary to, and not as a substitute for, other techniques (such as testing) which remove faults before the software is put into operation.