Aspect Oriented Software Engineering

In most large systems, the relationships between the requirements and the program components are complex. A single requirement may be implemented by a number of components and each component may include elements of several requirements. In practice, this means that implementing a change to the requirements may involve understanding and changing several components. Alternatively, a component may provide some core functionality but also include code that implements several system requirements. Even when there appears to be significant reuse potential, it may be expensive to reuse such components. Reuse may involve modifying them to remove extra code that is not associated with the core functionality of the component. Aspect-oriented software engineering (AOSE) is an approach to software development that is intended to address this problem and so make programs easier to maintain and reuse. AOSE is based around abstractions called aspects, which implement system functionality that may be required at several different places in a program. Aspects encapsulate functionality that cross-cuts and coexists with other functionality that is included in a system. They are used alongside other abstractions such as objects and methods. An executable aspect-oriented program is created by automatically combining (weaving) objects, methods, and aspects, according to specifications that are included in the program source code. An important characteristic of aspects is that they include a definition of where they should be included in a program, as well as the code implementing the crosscutting concern. You can specify that the cross-cutting code should be included before or after a specific method call or when an attribute is accessed. Essentially, the aspect is woven into the core program to create a new augmented system. The key benefit of an aspect-oriented approach is that it supports the separation of concerns. As I explain in Section 21.1, separating concerns into independent elements rather than including different concerns in the same logical abstraction is good software engineering practice. By representing cross-cutting concerns as aspects, these concerns can be understood, reused, and modified independently, without regard for where the code is used. For example, user authentication may be represented as an aspect that requests a login name and password. This can be automatically woven into the program wherever authentication is required. Say you have a requirement that user authentication is required before any change to personal

details is made in a database. You can describe this in an aspect by stating that the authentication code should be included before each call to methods that update personal details. Subsequently, you may extend the requirement for authentication to all database updates. This can easily be implemented by modifying the aspect. You simply change the definition of where the authentication code is to be woven into the system. You do not have to search through the system looking for all occurrences of these methods. You are therefore less likely to make mistakes and introduce accidental security vulnerabilities into your program