Module 10

Aspect-Oriented Software Engineering

Objectives:

1. Understand the phenomena of tangling and scattering.
2. Understand what is meant by the separation of concerns.
3. Know what is meant by a cross-cutting concern.
4. Know the general difference between core concerns and secondary concerns, and know which ones tend to be cross-cutting concerns.
5. Be familiar with some of the basic terminology used in aspect-oriented software engineering: aspect, advice, join point, join point model, pointcut, and aspect weaver.
6. Be introduced to the concept of aspect weaving, and know the three different ways of weaving aspects into system code.
7. Be familiar with some of the pitfalls to using aspects.
8. Understand some of the considerations involved in integrating aspects into the software engineering process.

Reading:

1. *Sommerville,* Web Chapter 31 (Aspect-oriented Software Engineering, available on the companion website: http://iansommerville.com/software-engineering-book/files/2014/07/Ch-31-Aspect-oriented-SE.pdf).
2. Web section for chapter 4 (available at http://iansommerville.com/software-engineering-book/web/web-sections/):
    a. Viewpoints (Chapter 4)

Assignment:

No assignment for Module 10.

Quiz:

No quiz for Module 10.

Tangling and Scattering

In previous modules I have discussed the importance of requirement traceability; i.e., being able to trace every requirement to one or more design products, the corresponding code, and even corresponding test cases. One of the main reasons given for ensuring requirement traceability was that if changes need to be made (bug fixes, extension of functionality, etc.), the traceability documentation would indicate which components are involved. If a requirement is present in only one place in the design, and likewise only present in the source code for a single component, changes are localized and relatively easy to make. There are two situations where changes are not so easy to make, and in fact may be quite difficult to implement. These situations have been named **tangling** and **scattering**.

Tangling results when a single component in a software system implements multiple requirements. In Chapter 31, Figure 21.2 (sic) of the text, Sommerville gives the example of a procedure for adding a record to a bounded buffer system. Obviously, the main requirement for this procedure is to add a new record to the buffer. But in the example, data can be concurrently read from the buffer, so to ensure no problems arise from concurrent reads and writes, the procedure is declared as synchronized, and code has been added to enforce the synchronization. This procedure is tangled because there are two separate requirements being fulfilled in the same procedure:

1. adding a new record to the buffer

2. ensuring there are no concurrency issues

Another way to look at tangling is that tangling results when components are not designed to be cohesive units. Ideally, we would like to have any given component be responsible for, and perform, only a single task.

Scattering occurs when a single requirement is implemented in multiple components. User authentication is a prime example of scattering, since users may need to be authenticated at multiple points in a system. Other examples include policy or regulatory constraints that are imposed on a system. In the medical record management system Sommerville mentions, adherence to HIPAA laws would be a concrete example of a regulatory constraint. In source code, scattering is usually manifested as duplications of the code handling the scattered requirement in all the places where that requirement is implemented.

Although both tangling and scattering can cause problems during the development of the initial system, they cause the most problems when the system must be changed. The lack of cohesion caused by tangling means that if one of the requirements of a tangled component is changed, that change may not be compatible with the other requirements implemented by the component, and unexpected system behavior may arise. With scattering, if a scattered requirement is changed, all copies of the code implementing that requirement must be changed. It may not be easy to locate all the places where the code must be changed in a large system. Traceability of requirements will help with this, but each copy of the code must still be changed, with the ever-present risk that any change may inadvertently introduce another fault.

Separation of Concerns

It should be pointed out that the problems of tangling and scattering lie first and foremost in the realm of requirements, not source code. A well-structured, thorough, requirements document will give a good idea of which requirements are likely to end up being tangled, and which ones are likely to end up being scattered. The key to accomplishing this is to ensure **separation of concerns** as the requirements are organized. The separation of concerns is one of the fundamental principles of software design, and its importance has been known for around 60 years now. An accurate definition of what, exactly, constitutes a “concern” is difficult to make. One notion is that a concern represents any single piece of functionality in the system. A broader definition states that a concern is any element of interest in the software system. Though broad, this definition decouples concerns from code, and makes them independent of any particular programming language. Sommerville qualifies this definition by adding that a concern must be of interest to one or more stakeholders in the system, and further categorizes stakeholder concerns into five types:

1. Functional concerns; these relate to the specific functionality of a system

2. Quality of Service (QoS) concerns; e.g., a system's performance and reliability

3. Policy concerns; specifically, related to policies for system usage, and laws or regulations that impact the system

4. System concerns; e.g., how easy the system is to maintain, how easy it is to configure

5. Organizational concerns; these are related to constraints, such as budget, at the level of the entire organization

The functional concerns are typically referred to as the **core concerns** of the system; the other concerns in the list above are secondary in nature. Secondary concerns are interesting in that they often cut across multiple core concerns, as Sommerville shows in Chapter 31, Figure 21.1 (sic). In that figure, core functionality is represented as vertical bars, while secondary, **cross-cutting concerns** are represented as horizontal bars that overlap all the core concerns they are designed to influence.

Aspects and Aspect-Oriented Software Development

To facilitate the separation of concerns, and to address the issues of tangling and scattering, the concept of Aspect-Oriented Software Engineering (AOSE) was introduced in the late 1990's. Credit for the first appearance of software that supported AOSE is usually given to the AspectJ programming language—designed as an extension for the Java programming language—developed in 1997 at Xerox's PARC Laboratories (now known simply as PARC), but currently aspect-oriented development is supported for most major programming languages. Before going any further, a few definitions are in order:

An **aspect** is an abstraction that defines a cross-cutting concern; i.e., a concern that involves, or cuts across, multiple core functionalities. The code that implements the cross-cutting concern is called the **advice**. Execution of the advice code is triggered by a set of events known as **join points**. A join point can thus be thought of as a point where the cross-cutting concern must be addressed during program execution. The set of all events where the cross-cutting concern must be addressed is called the **join point model**. Within an aspect, the statement that defines where the advice code should be executed is called the **pointcut**.

A generic example of an aspect is shown in the text in Chapter 31, Figure 21.5 (sic). For convenience, I've reproduced the skeleton of the aspect here:

```
aspect authentication
{
      // define the pointcut
      before: call (public void update* (..))

      // actual advice code goes here
}
```

A basic aspect therefore contains the advice code that addresses some cross-cutting concern, as well as a definition of the pointcut, which defines where in the rest of the application code the advice code should be executed. There are a couple of important things to note about pointcuts:

1. Knowledge of the rest of the application code is essential to defining a pointcut, since the identifiers of specific programming constructs must be referenced.

2. Wildcard symbols may be used to allow fuzzy matching of identifiers.

In the example above, the pointcut is defined as being immediately before any public method with a void return type, and with a name that begins with "update". Any parameter list is allowed. Each such method is therefore a join point, and the complete set of all such methods comprises the join point model. Method calls are not the only events that can trigger execution of an aspect. Pretty much any operation in a

programming language can be used as a join point. Object instantiation, attribute updates and accesses, and exceptions also can be used as join points.

Once an aspect has been defined, the advice code must be incorporated somehow into the code for the rest of the system. This is accomplished by a process known as **weaving**, and is performed by a program known as an **aspect weaver**. Weaving an aspect into a system essentially entails merging the functionality of the aspect with the functionality of the advice code, at the location of the pointcut. There are three approaches to doing this, differing in the time at which the advice code is woven into the system:

1. *At the level of the source code*. The advice code can be inserted at the appropriate locations in the system code, thereby generating modified source code that contains both the original code and the aspect code. The modified code is then compiled using a standard compiler for the programming language. This method is easiest to understand conceptually, but suffers from the extra time required to generate new, integrated, source code.

2. *At link time*. Instead of generating modified source code, which can then be compiled, the advice code and system code are compiled and joined simultaneously. This requires a special compiler that has been modified to include an aspect weaver. This approach eliminates the extra step of having to generate new source code by combining advice code integration into the compilation process. The disadvantage is that the standard compilers for the programming language can no longer be used.

3. *At execution time*. In this case, both the system code and the advice code remain as separate entities. During execution, when the advice code is needed it is incorporated into the executing system on the fly. This approach allows the greatest flexibility, but the continuous monitoring of join points that is required to allow dynamic integration of advice code can incur serious overhead at runtime, thus decreasing system performance.

Pitfalls to Using Aspects

Aspects are great for handling concerns that span multiple core functionalities, but there are also some inherent problems with using aspects. Since aspect definition requires special knowledge of the source code for a system, great care must be taken when assigning values to the identifiers in the system source code. In the authentication example discussed in the text, user authentication will be triggered immediately before the invocation of any public, void, method whose identifier begins with the string “update”. As written, the aspect does not discern between methods that fit that constraint, it simply triggers execution of the user authentication functionality before them all. For proper system behavior, there must be no methods fitting the constraint that should **not** require user authentication prior to their execution.

Care must also be taken not to assume too much about the internal details of the system when designing aspects, since unwarranted assumptions may lead to unpredictable behavior. In a sense, aspects violate to an extent the principle of information hiding.

Aspects also tend to corrupt structure in a program similarly to the way the no obsolete “GOTO” statement did. In the case of aspects, it’s more of a “COME FROM” situation rather than a “GOTO” situation, but the effects are similar.

Although the example shown in the text involves only a single aspect, it is possible to have many aspects incorporated throughout a system. If all the aspects are designed independently of each other, it is possible for two or more aspects to interfere with each other if they are both triggered by the same join point.

Verification and validation are more problematic when aspects are included. Since the advice code for an aspect is separate from the source code for the rest of the system, reading the source code sequentially is difficult, if not impossible, especially when you consider the fact that wildcards used in pointcut definitions force excessive jumping back and forth between system code and aspect code. This problem may be alleviated by integrating aspects at the source code level, rather than at link time or during execution. Aspect weaving at the level of the source code is not commonly employed; link time weaving is more commonly used. Special code readers are available that perform this function for the purposes of code reviews, but

unless a code reader is able to integrate the aspects into the system code in exactly the same way as the aspect weaver accompanying the compiler used to generate the executable code, the code that is produced may not accurately reflect what will happen in the compiled program.

Aspects as Part of a Software Engineering Process

If aspects are to be effectively used, the software process must be modified to accommodate them. Since the cross-cutting concerns that are at the heart of aspects relate primarily to user requirements, accommodation for aspects should be included as an integral part of the requirements engineering process. If the requirements have been well organized, it is usually possible to identify cross-cutting concerns as requirements that appear in multiple core functionalities. Use case models are particularly effective for this, since they detail the step-by-step procedures the end users will follow to accomplish certain tasks.

During design, several factors must be considered, including:

1. Naming conventions should be created with aspects in mind, in order to avoid situations where aspects are executed at the wrong locations.

2. Should a particular aspect stand on its own, or should it be decomposed into multiple aspects?

3. Where in the core functionality should the join points be located?

4. Are there any conflicts or potential conflicts between two or more aspects? If so, how will these conflicts be resolved?

Finally, test cases must take the presence of aspects into account. Since a system that incorporates aspects tends to be less well-structured than a system that does not use aspects, the code cannot be followed as easily. Traditional white box testing approaches make use of various types of code coverage, such as branch coverage, statement coverage, and path coverage. But there is no clear definition of what aspect coverage means. In basis path coverage, for example, the goal is to ensure that the minimum number of paths that allow complete statement and branch coverage are executed at least once. If that idea were extended to define aspect coverage, it would mean that each aspect would be covered, in its entirety, once. However, since a given aspect can be triggered at many locations throughout a system, it is possible that an aspect will behave as expected for some of the locations, but not all of them. There are other problems, too, such as how to deal with different aspects that share the same pointcut. One of the areas of active research in the area of aspect-oriented software engineering is to find ways to test aspects independently of the rest of the system code to which they will be joined.

Aspects are relatively new, and they have yet to be embraced as a mainstream software engineering process, despite the fact that aspects are supported by the commonly used programming languages. There are many active areas of research into the notion of aspects, and how best to integrate aspects into both traditional, and agile, software processes.