Chapter 21

Aspect-Oriented Software Engineering (AOSE)
Topics covered

- Introduction and motivation
- The *separation of concerns*
- *Core vs. cross-cutting concerns*
- *Aspects, join points, and pointcuts*
- *Information hiding vs. AOSE*
- Problems with AOSE

(Note: we will **NOT** cover 21.3, “SE with aspects,” *except* for 21.3.3, “Verification and Validation”)

In most systems, the mapping between requirements ("concerns") and components is not 1:1.

To illustrate, suppose we have “requirements” with just two attributes: color and shape.

Requirements are either red or green, and are either square (□) or circular (○).

How should we distribute the requirements among two components if our goal is to minimize the cost of change?

(cont’d)
Introduction and motivation (cont’d)
Thus, implementing a requirements change may involve understanding and changing more than one component.

Aspect-Oriented Software Engineering (AOSE) is an approach to SE intended to address this problem.

It is based on a relatively new type of abstraction - an aspect, and is normally used together with OO software engineering.

(cont’d)
Introduction and motivation (cont’d)

- **Aspects** encapsulate functionality that *cross-cuts* other functionality.
- R&D in this area has primarily focused on *aspect-oriented programming*.
- Languages such as **AspectJ** have extended OO languages to include aspects, and have been used in major companies.
- AOSE’s focus on *separating concerns* is an important way of thinking about and structuring software systems, but it is not yet mainstream SE.

**AspectJ** was developed at Xerox PARC and made available in 2001.
The separation of concerns

- The *principle of separation of concerns* states that software should be organized so that *each* program element does one thing and one thing only.

- Each program element should therefore be understandable without reference to other elements.

- Program abstractions (subroutines, procedures, objects, *and aspects*) support the separation of concerns.
What are concerns?

- **Concerns** reflect system requirements and the priorities of system stakeholders.
- Examples of concerns are performance, security, specific functionality, etc.
- Programs that reflect the separation of concerns in a program ⇒ clear traceability from requirements.
- This facilitates program understanding and the implementation of requirements change.
Types of concerns (programming and otherwise)

- **Functional**: related to specific functionality to be included in a system.

- **Quality of service**: related to the non-functional behaviour of a system (e.g., performance, reliability, availability).

- **Policy concerns**: related to the overall policies that govern the use of the system.

- **System**: related to attributes of the system as a whole (e.g., maintainability, configurability).

- **Organizational**: related to organizational goals and priorities (e.g., staying within budget, using existing software assets).
Core vs. cross-cutting concerns

- **Core concerns** relate to a system’s primary purpose and are **normally implemented within separate procedures, objects, etc.**

- **Cross-cutting concerns** are those whose implementation cut across a number of program components, resulting in problems when changes are required due to:
  - *tangling* (a component implements multiple requirements), and
  - *scattering* (a requirement’s implementation is scattered across more than one component).
Cross-cutting concerns

Core concerns

New customer reqmts
Account reqmts.
Customer management reqmts

Cross-cutting concerns

Security reqmts
Recovery reqmts

Internet Banking System
Cross-cutting concerns

Core concerns

New customer reqmts
Account reqmts.
Customer management reqmts

Internet Banking System

Cross-cutting concerns

Security reqmts
Recovery reqmts

Security reqmts
Recovery reqmts

Cross-cutting concerns

tangling
Cross-cutting concerns

Core concerns

New customer reqmts
Account reqmts.
Customer management reqmts

Cross-cutting concerns

Internet Banking System

Security reqmts
Recovery reqmts

tangling

scattering
Tangling of buffer management and synchronization code

synchronized void put (SensorRecord rec )
{
    // Check that there is space in the buffer; wait if not
    if ( numberOfEntries == bufsize)
    wait () ;

    // Add record at end of buffer
    store [back] = new SensorRecord (rec.sensorId, rec.sensorVal) ;
    back = back + 1 ;
    // If at end of buffer, next entry is at the beginning
    if (back == bufsize)
    back = 0 ;
    numberOfEntries = numberOfEntries + 1 ;
    // indicate that buffer is available
    notify () ;
}
} // put

ensures mutual exclusion
Scattering of methods implementing secondary concerns

code associated with secondary statistics collection concern
Aspects, join points, and pointcuts

- An **aspect** is an abstraction which implements a cross-cutting concern.
- It includes the code (**advice**) to be executed and a statement (**pointcut**) that defines the **events** for which the advice is woven into the program.
- The program events specified by the **pointcut** are called **join points**.
### Summary of terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>advice</td>
<td>The code implementing a concern.</td>
</tr>
<tr>
<td>aspect</td>
<td>A program abstraction that defines a cross-cutting concern. It includes the definition of a pointcut and the advice associated with that concern.</td>
</tr>
<tr>
<td>join point</td>
<td>An event in an executing program where the advice associated with an aspect may be executed.</td>
</tr>
<tr>
<td>join point model</td>
<td>The set of events that may be referenced in a pointcut.</td>
</tr>
<tr>
<td>pointcut</td>
<td>A statement, included in an aspect, that defines the join points where the associated aspect advice should be executed.</td>
</tr>
<tr>
<td>weaving</td>
<td>The incorporation of advice code at the specified join points by an aspect weaver.</td>
</tr>
</tbody>
</table>
AspectJ join point model

- **Call events**: calls to a method or constructor
- **Execution events**: execution of a method or constructor
- **Initialization events**: class or object initialization
- **Data events**: accessing or updating a field
- **Exception events**: the handling of an exception
An “authentication” aspect

```java
aspect authentication {
  before: call (public void update*(..)) // this is a pointcut 
  { 
    // this is the advice that should be executed when woven into 
    // the executing system 
    int tries = 0;
    string userPassword = Password.Get( tries );
    while (tries < 3 & & userPassword != thisUser.password( ))
    { 
      // allow 3 tries to get the password right 
      tries = tries + 1;
      userPassword = Password.Get( tries );
    } 
    if (userPassword != thisUser.password()) then 
      // if password wrong, assume user has forgotten to logout 
      System.Logout(thisUser.uid); 
  } // authentication
```
An “authentication” aspect

```java
aspect authentication
{
    before: call (public void update*(..)) // this is a pointcut
    {
        // this is the advice that should be executed when woven into
        // the executing system
        int tries = 0;
        string userPassword = Password.Get( tries );
        while (tries < 3 & & userPassword ! = thisUser.password( ) )
        {
            // allow 3 tries to get the password right
            tries = tries + 1;
            userPassword = Password.Get( tries );
        }
        if (userPassword ! = thisUser.password( )) then
            // if password wrong, assume user has forgotten to logout
            System.Logout (thisUser.uid);
    }
} // authentication
```

Meaning: Before executing any method whose name starts with “update,” execute the advice.
Summarizing the info provided on slides 16-18…

- A **pointcut** is a *statement in an Aspect* that defines (identifies) one or more **join points** (events in an executing program) where the Aspect’s **advice** (code) should be executed.

- In AspectJ, the **join points that may be identified in a pointcut** are: call events, execution events, initialization events, data events, and exception events.
Aspect weaving

- Aspect weavers process source code and weave advice into a program at the join points defined in pointcuts.

- Three approaches to aspect weaving
  - Source code pre-processing
  - Link-time weaving
  - Dynamic, execution-time weaving

(cont’d)
Aspect weaving (cont’d)
**Information hiding vs. AOSE**

**Information Hiding:**
isolating potentially changeable design decisions *(core concerns)* in separate program components

**AOSE:**
isolating potentially changeable “*cross-cutting concerns*” (design decisions that cut across program components) in separate program aspects.
Problems with AOSE

- AO programs can be “black-box tested” using requirements to design the tests.

- But program inspections and “white-box testing” can be problematic, since you can’t always tell from the source code alone where an aspect will be woven and executed.

- “Flattening” an aspect-oriented program (to make it readable sequentially from top to bottom) is problematic.
White-box testing

- White-box or *structural* testing uses control flow knowledge to *systematically* design defect tests.
- The aim is to provide some level of code coverage (*statement coverage*, *branch coverage*, *path coverage*, etc.)
- But deriving a control flow graph of a program with aspects is problematic.
- And it may be difficult to design tests to cover *all combinations* of program joint points and aspects.
AOSE white-box testing issues

- What does test coverage *mean* in AOSE?
- How should aspects be specified so that tests may be designed?
- How can aspects be tested independently of the base system?
- How can *aspect interference* (when two or more aspects use the same pointcut specification) be tested?
Key points

- AOSE supports the separation of concerns…
- By representing cross-cutting concerns as aspects, individual concerns can be understood, reused and modified without changing other parts of the program.
- *Tangling* occurs when a module in a system includes code that implements different system requirements.

(cont’d)
Key points (cont’d)

- **Scattering** occurs when the implementation of a concern is scattered across several components.

- Aspects include a **pointcut** statement that defines where the aspect will be woven into the program, and **advice** – the code to implement the cross-cutting concern.

- **Join points** are the events specified in a **pointcut**.

(cont’d)
Key points (cont’d)

- The problems of inspecting and designing structural tests for aspect-oriented programs are significant barriers to the adoption of AOSE in large software projects.
Chapter 21

Aspect-Oriented Software Engineering (AOSE)