Aspect-oriented Software Development
Objectives

- To explain the principle of separation of concerns in software development
- To introduce the fundamental ideas underlying aspect-oriented development
- To show how an aspect-oriented approach can be used at all stages of development
- To discuss problems of testing aspect-oriented systems
Topics covered

- The separation of concerns
- Aspects, join points and pointcuts
- Software engineering with aspects
Aspect-oriented software development

- An approach to software development based around a new type of abstraction - an aspect.
- Used in conjunction with other approaches - normally object-oriented software engineering.
- Aspects encapsulate functionality that cross-cuts and co-exists with other functionality.
- Aspects include a definition of where they should be included in a program as well as code implementing the cross-cutting concern.
The separation of concerns

- The principle of separation of concerns states that software should be organised 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, etc.) support the separation of concerns.
Concerns

- Concerns are not program issues but reflect the system requirements and the priorities of the system stakeholders.
  - Examples of concerns are performance, security, specific functionality, etc.
- By reflecting the separation of concerns in a program, there is clear traceability from requirements to implementation.
- Core concerns are the functional concerns that relate to the primary purpose of a system; secondary concerns are functional concerns that reflect non-functional and QoS requirements.
Stakeholder concerns

- Functional concerns which are related to specific functionality to be included in a system.
- Quality of service concerns which are related to the non-functional behaviour of a system.
- Policy concerns which are related to the overall policies that govern the use of the system.
- System concerns which are related to attributes of the system as a whole such as its maintainability or its configurability.
- Organisational concerns which are related to organisational goals and priorities such as producing a system within budget, making use of existing software assets or maintaining the reputation of an organisation.
Cross-cutting concerns

- Cross-cutting concerns are concerns whose implementation cuts across a number of program components.
- This results in problems when changes to the concern have to be made - the code to be changed is not localised but is in different places across the system.
- Cross cutting concerns lead to tangling and scattering.
Cross-cutting concerns

Diagram showing the relationship between cross-cutting concerns and core concerns. The diagram includes categories such as 'New customer req.', 'Account req.', 'Customer management req.', 'Security req.', and 'Recovery req.'
synchronized void put (SensorRecord rec ) throws InterruptedException  
{
  if ( numberOfEntries == bufsize)  
    wait () ; 
  store [back] = new SensorRecord (rec.sensorId, rec.sensorVal) ;
  back = back + 1 ; 
  if (back == bufsize)  
    back = 0 ; 
  numberOfEntries = numberOfEntries + 1 ; 
  notify () ; 
} // put
Scattering

```
Patient
<attribute decls>
getName ()
editeName ()
geetAddress ()
editAddress ()
...
匿名ise ()
...

Image
<attribute decls>
getModality ()
archive ()
getDate ()
editDate ()
...
saveDiagnosis ()
saveType ()
...

Consultation
<attribute decls>
makeAppointment ()
cancelAppointment ()
assignNurse ()
bookEquipment ()
...
匿名ise ()
saveConsultation ()
...
```
Aspects, join points and pointcuts

- An aspect is an abstraction which implements a concern. It includes information where it should be included in a program.
- A join point is a place in a program where an aspect may be included (woven).
- A pointcut defines where (at which join points) the aspect will be included in the program.
## Aspect 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>
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
```
AspectJ - join point model

- Call events
  - Calls to a method or constructor

- Execution events
  - Execution of a method or constructor

- Initialisation events
  - Class or object initialisation

- Data events
  - Accessing or updating a field

- Exception events
  - The handling of an exception
Pointcuts

- Identifies the specific events with which advice should be associated.
- Examples of contexts where advice can be woven into a program
  - Before the execution of a specific method
  - After the normal or exceptional return from a method
  - When a field in an object is modified
Aspect weaving

- Aspect weavers process source code and weave the aspects into the program at the specified pointcuts.

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

- Authentication aspect
- Logging aspect
- Patient
  - ... updateDetails (...) ...

Aspect weaving:
- authentication code
- updateDetails (...) logging code
- ...

Patient
Software engineering with aspects

- Aspects were introduced as a programming concept but, as the notion of concerns comes from requirements, an aspect oriented approach can be adopted at all stages in the system development process.

- The architecture of an aspect-oriented system is based around a core system plus extensions.

- The core system implements the primary concerns. Extensions implement secondary and cross-cutting concerns.
Core system + extensions
Types of extension

- **Secondary functional extensions**
  - Add extra functional capabilities to the core system

- **Policy extensions**
  - Add functional capabilities to support an organisational policy such as security

- **QoS extensions**
  - Add functional capabilities to help attain quality of service requirements

- **Infrastructure extensions**
  - Add functional capabilities to support the implementation of the system on some platform
Concern-oriented requirements engineering

- An approach to requirements engineering that focuses on customer concerns is consistent with aspect-oriented software development.
- Viewpoints (discussed in Chapter 7) are a way to separate the concerns of different stakeholders.
- Viewpoints represent the requirements of related groups of stakeholders.
- Cross-cutting concerns are concerns that are identified by all viewpoints.
Viewpoints and Concerns

[Diagram showing Viewpoints and Concerns]

- Viewpoints:
  - Equipment
  - Users
  - Managers
  - Organisation
  - Society

- Concerns:
  - Regulation
  - Security
  - Dependability

THE SYSTEM
1. Emergency service users
1.1 Find a specified type of equipment (e.g. heavy lifting gear)
1.2 View equipment available in a specified store
1.3 Check-out equipment
1.4 Check-in equipment
1.5 Arrange equipment to be transported to emergency
1.6 Submit damage report
1.7 Find store close to emergency

2. Emergency planners
2.1 Find a specified type of equipment
2.2 View equipment available in a specified location
2.3 Add and remove equipment from a store
2.4 Move equipment from one store to another
2.6 Order new equipment

3. Maintenance staff
3.1 Check-in/Check-out equipment for maintenance
3.2 View equipment available at each store
3.3 Find a specified type of equipment
3.4 View maintenance schedule for an equipment item
3.5 Complete maintenance record for an equipment item
3.6 Show all items in a store requiring maintenance
Availability requirements

AV.1 There shall be a ‘hot standby’ system available in a location that is geographically well-separated from the principal system.

*Rationale*: The emergency may affect the principal location of the system.

AV.1.1 All transactions shall be logged at the site of the principal system and at the remote standby site.

*Rationale*: This allows these transactions to be replayed and the system databases made consistent.

AV.1.2 The system shall send status information to the emergency control room system every five minutes.

*Rationale*: The operators of the control room system can switch to the hot standby if the principal system is unavailable.
Inventory system - core requirements

- C.1 The system shall allow authorised users to view the description of any item of equipment in the emergency services inventory.
- C.2 The system shall include a search facility to allow authorised users to search either individual inventories or the complete inventory for a specific item or type of equipment.
Inventory system - extension requirements

- E1.1 It shall be possible for authorised users to place orders with accredited suppliers for replacement items of equipment.
- E1.1.1 When an item of equipment is ordered, it should be allocated to a specific inventory and flagged in that inventory as ‘on order’.
Aspect-oriented design/programming

- Aspect-oriented design
  - The process of designing a system that makes use of aspects to implement the cross-cutting concerns and extensions that are identified during the requirements engineering process.

- Aspect-oriented programming
  - The implementation of an aspect-oriented design using an aspect-oriented programming language such as AspectJ.
Use-cases

- A use-case approach can serve as a basis for aspect-oriented software engineering.
- Each use case represents an aspect.
  - Extension use cases naturally fit the core + extensions architectural model of a system
- Jacobsen and Ng develop these ideas of using use-cases by introducing new concepts such as use-case slices and use case modules.
An extension use case
Inventory use cases

- Operator
  - Remove equipment from store
  - Add equipment to store
  - Place equipment order
Inventory extension use-case
An AOSD process
UML extensions

Expressing an aspect oriented design in the UML requires:

- A means of modelling aspects using UML stereotypes.
- A means of specifying the join points where the aspect advice is to be composed with the core system.
An aspect-oriented design model
A partial model of an aspect

- **pointcuts**
  - `viewMain = call getItemInfo (..)`
  - `mainco = call removeItem (..)`
  - `main ci = call addItem (..)`

- **class extensions**
  - `ViewMaintenanceHistory`
  - `<viewItem> {after (<viewMain>) displayHistory}`

*More extensions here*
Verification and validation

● The process of demonstrating that a program meets its specification (verification) and meets the real needs of its stakeholders (validation)

● Like any other systems, aspect-oriented systems can be tested as black-boxes using the specification to derive the tests

● However, program inspections and ‘white-box’ testing that relies on the program source code is problematic.

● Aspects also introduce additional testing problems
Testing problems with aspects

- How should aspects be specified so that tests can be derived?
- How can aspects be tested independently of the base system?
- How can aspect interference be tested?
- How can tests be designed so that all join points are executed and appropriate aspect tests applied?
Program inspection problems

- To inspect a program (in a conventional language) effectively, you should be able to read it from right to left and top to bottom.
- Aspects make this impossible as the program is a web rather than a sequential document. You can’t tell from the source code where an aspect will be woven and executed.
- Flattening an aspect-oriented program for reading is practically impossible.
White box testing

- The aim of white box testing is to use source code knowledge to design tests that provide some level of program coverage e.g. each logical branch in a program should be executed at least once.

- Aspect problems
  - How can source code knowledge be used to derive tests?
  - What exactly does test coverage mean?
Aspect problems

- Deriving a program flow graph of a program with aspects is impossible. It is therefore difficult to design tests systematically that ensure that all combinations of base code and aspects are executed.

- What does test coverage mean?
  - Code of each aspect executed once?
  - Code of each aspect executed once at each join point where aspect woven?
  - ???
Key points

- The key benefit of an aspect-oriented approach is that it supports the separation of concerns.
- Tangling occurs when a module implements several requirements; Scattering occurs when the implementation of a single concern is spread across several components.
- Systems may be designed as a core system with extensions to implement secondary concerns.
Key points

- To identify concerns, you may use a viewpoint-oriented approach to requirements engineering.
- The transition from requirements to design may be made using use-cases where each use-case represents a stakeholder concern.
- The problems of inspecting and deriving tests for aspect-oriented programs are a significant barrier to the adoption of AOSD.