Aspect-Oriented Software Development

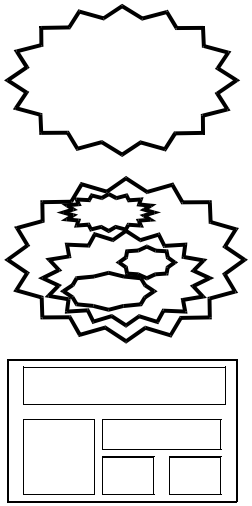
Claudia Marcos

ISISTAN – Instituto de Sistemas Tandil
Facultad de Ciencias Exactas - UNICEN

Current Methods and Languages

- Most current programming languages support abstraction mechanisms for breaking a system down into parameterised components which perform some function.
- Current methods and notations concentrate on finding and composing functional units - generally expressed as objects, modules, procedures.

Decomposition & Composition of Concerns



complex systems

↓

design practices support decomposition into components

↓

programming languages support implementation and composition of components

Separation of Concerns

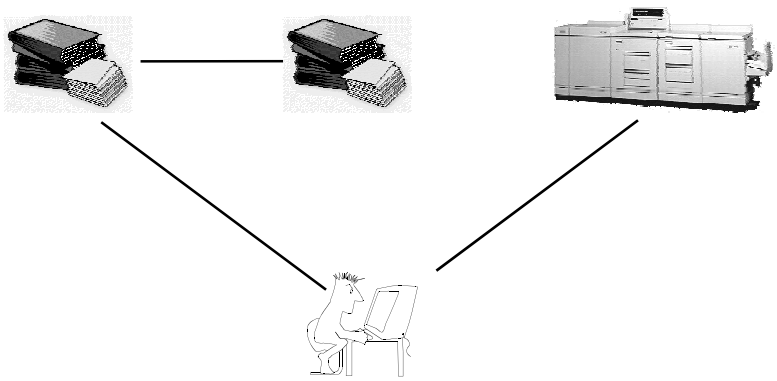
***Concern*: matter for consideration; something that relates to me or is of my interest.**

Good separation of concerns:

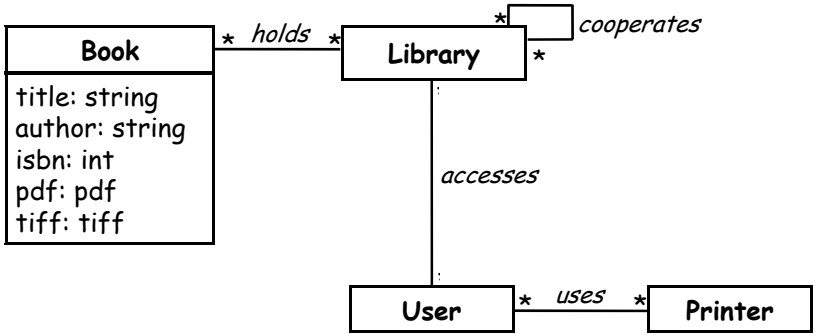
- localized in the design
- localized in the code
- handled explicitly

An Example System

a distributed digital library



Clean Modular Design



Clean Modular Code

Book

```
class Book {
  private String         title;
  private String         author;
  private String         isbn;
  private PostScript   ps;
  private User    borrower;

  public Book(String t, String a, String i, PostScript p) {
    title = t;
    author = a;
    isbn = i;
    ps = p;
  }

  public User get_borrower() {return borrower;}
  public void set_borrower(User u) {borrower = u;}
  public PostScript get_ps() { return ps; }
}
```

Printer

```
public class PrinterImpl {
  String status = "Idle"
  Vector jobs;

  public PrinterImpl() {}
  public get_status() { return status }
  public add_job(int j) {
         jobs.add(j);
     }
}
```

User

```
class User {
  private String name;
  Library theLibrary;
  Printer thePrinter;

  public User(String n) { name = n; }

  public boolean getBook (String title) {
    Book aBook = theLibrary.getBook(this, title);
    thePrinter.print(this,aBook);
    return true;
  }
}
```

Library

```
class Library  {
  Hashtable books;
  Library(){
    books = new Hashtable(100);
  }
  public Book getBook(User u, String title) {
    System.out.println("REQUEST TO GET BOOK " + title);
    if(books.containsKey(title)) {
      Book b = (Book)books.get(title);
      System.out.println("getBook: Found it:" + b);
      if (b != null) {
        if (b.get_borrower() == null)
          b.set_borrower(u);
          return b;
      }
    }
    return null;
  }
}
```


Benefits of Good Modularity

- Also known as “clean separation of concerns”
- Each decision / function in a single place
  - easy to understand
  - easy to modify
  - easy to unplug

Modularity is not always possible... (1)

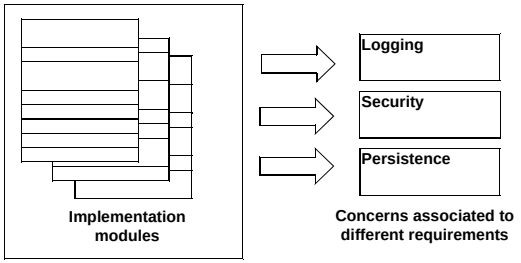
- A system may be seen as a set of concerns:
  - basic functionality
  - performance
  - data persistence
  - resource sharing
  - error handling
  - performance optimizations
  - synchronization
  - tracing
  - security
  - . . .

Modularity is not always possible... (2)

Some concerns don't localize to objects

… their code tends to be orthogonal to the rest of the requirements

… and is spread out through many modules



"Cross-cutting"

Book

```
class Book {
  private BookID        id;
  private PostScript    ps;
  private UserID        borrower;

  public Book(String t, String a, String i, PostScript p) {
    id = new BookID(t,a,i);
    ps = p;
  }

  public UserID get_borrower() {return borrower;}
  public void set_borrower(UserID u) {borrower = u;}
  public PostScript get_ps() { return ps; }
  public BookID get_bid() { return id; }
}

class BookID {
  private String    title;
private String      isbn;

  public BookID(String t, String a, String i) {
    title = t;
    author = a;
    isbn = i;
  }
  public String get_title() {return title;}
}
```

User

```
class User {
  private UserID id;
  Library theLibrary;
  Printer thePrinter;

  public User(String n) { id = new UserID(n); }

  public boolean getBook (String title) {
    BookID aBook=null;
    try{
      aBook = theLibrary.getBook(id, title);
    } catch (RemoteException e) {}
    try {
      thePrinter.print(id, aBook);
    } catch (RemoteException e) {}
    return true;
  }
  public UserID get_uid() { return id; }
}

class UserID {
  private String name;

  public UserID(String n) { name = n; }
  public String get_name() { return name; }
}
```

Library

```
interface LibraryInterface extends Remote {
  public BookID getBook(UserID u, String title) throws
RemoteException;

  public PostScript getBookPS(BookID bid) throws RemoteException;
}

class Library extends UnicastRemoteObject implements
LibraryInterface  {
  Hashtable books;
  Library() throws RemoteException {
    books = new Hashtable(100);
  }
  public BookID getBook(UserID u, String title)
  throws RemoteException {
    System.out.println("REQUEST TO GET BOOK " + title);
    if(books.containsKey(title)) {
      Book b = (Book)books.get(title);
      System.out.println("getBook: Found it:" + b);
      if (b != null) {
        if (b.get_borrower() == null)
          b.set_borrower(u);
          return b.get_bid();
      }
    }
    return null;
  }
  public PostScript getBookPS(BookID bid)
  throws RemoteException {
    if (books.containsKey(bid.get_title())) {
      Book b = (Book)books.get(bid.get_title());
      if (b != null)
        return b.get_ps();
    }
    return null;
  }
}
```

Printer

```
interface PrinterInterface extends Remote {
  public boolean print (UserID u, BookID b) throws RemoteException;
}

public class Printer extends UnicastRemoteObject
                implements PrinterInterface {
  private Vector jobs = new Vector(10, 10);
  private Library theLibrary;

  public Printer() throws RemoteException{}
  public boolean print (UserID u, BookID b) throws RemoteException{
    PostScript ps=null;
    try{
      ps = theLibrary.getBookPS(b);
    } catch (RemoteException e) {}
return queue(newJob);
  }
  boolean queue(Job j) {
  //...
  return true;
  }
}
```

this optimization <u>cross-cuts</u> the primary structure of the program

Cross-cutting

- Symptoms
  - Tangled code (several concerns in the same module)
  - Scattering code (a concern in several modules)
  - Code duplication and dispersion
- Difficult to reason about
  - why is this here?
  - what does this connect to?

→ ***it destroys modularity***

Effects of Cross-cutting

- Consequences:
  - Bad quality code
  - Poor traceability
  - Low productivity
  - Low reusability
  - Testing difficulties
  - Bad adaptability
  - Poor evolution

Why Aspect Oriented Programming?

- Most cross-cuts aren't random:
  - they serve specific purposes:
    - performance optimizations, interactions among objects, enforcing consistency, tracking global state…
  - they have well-defined structure:
    - lines of dataflow, boundary crossings, points of resource utilization, points of access…

Is it possible to capture the cross-cutting structure in a modular way?

Problems of Current Languages

- Many systems have properties that do not align with the basic functional components and cannot be expressed in a cleanly localised way.
- Current methods of abstraction and implementation do not support the "separation of these concerns".
- The consequence is "code tangling": the spreading of the code relating to these concerns through many components.

→ ***Aspect-Oriented System Design***

Aspect-Oriented Paradigm

The goal of AOP is to provide methods and techniques for:

- decomposing problems into a number of *functional components* as well as a number of *aspects* which *crosscut* the functional components, and then
- *composing* these components and aspects to obtain system implementations.

Aspect-Oriented Programming

• **Component:** can be encapsulated in procedures well located, acceded and clear

• **Aspect:** can not be encapsulated in an only one procedure, it is disseminated in the functional components

• **An object is something:** Is an entity

• **An aspect is not something is something about something:** An aspect exist to incorporate orthogonal functionality to an object

• **Objects not depend on aspects:** Objects do not change because of aspects

Aspect-Oriented Programming

- Properties or features that don't align with the functional components of a system ***cross-cut*** the primary division of labour that is achieved with objects.
- An ***aspect*** is a feature or unit that cross-cuts other components in the design or implementation.
- ***Weaving*** is the systematic process of combining the aspects and the functional units of a system.
- A ***join-point*** is the place where the weaver introduces the aspect code into the basic component.

Aspect: a cross-cutting module

```
class Book {
  private String   title;
  private String   author;
  private String   isbn;
  private PostScript  ps;
  private User      borrower;

  public Book(String t, String a, String i, PostScript p) {
    title = t;
    author = a;
    isbn = i;
    ps = p;
  }

  public User get_borrower() {return borrower;}
  public void set_borrower(User u) {borrower = u;}
  public PostScript get_ps() { return ps; }
}
```

```
class User {
  private String name;
  Library theLibrary;
  Printer thePrinter;

  public User(String n) { name = n; }
  public boolean getBook (String title) {
    Book aBook = theLibrary.getBook(this, title);
    thePrinter.print(this,aBook);
    return true;
  }
}
```

```
class Library  {
  Hashtable books;
  Library(){
    books = new Hashtable(100);
  }
  public Book getBook(User u, String title) {
    System.out.println("REQUEST TO GET BOOK " + title);
    if(books.containsKey(title)) {
      Book b = (Book)books.get(title);
      System.out.println("getBook: Found it:" + b);
      if (b != null) {
        if (b.get_borrower() == null)
          b.set_borrower(u);
          return b;
      }
    }
    return null;
  }
```

```
public class PrinterImpl {
  String status = "Idle"
  Vector jobs;

  public PrinterImpl() {}
  public get_status() { return status }
  public add_job(int j) {
        jobs.add(j);
    }
}
```

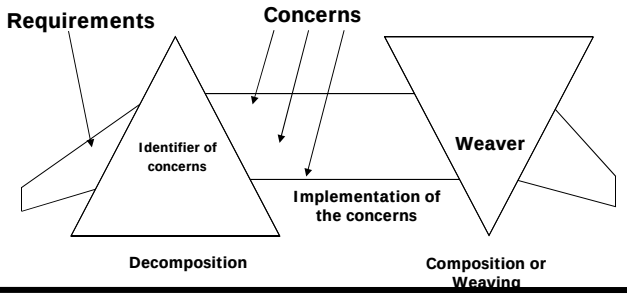
```
portal Printer {
  void print(Book book) {
    book: Book: {direct pages;}

}
portal Library {
  Book find (String title){
    return:
      Book: {copy title, author, isbn;}
  }
}
```

this aspect localizes the cross-cutting in one place

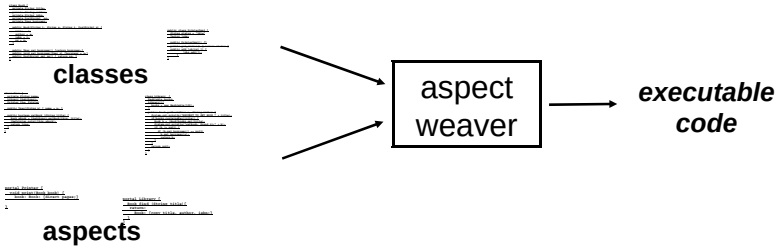
Stages in AOP

- Decomposition
- Implementation
- Composition or weaving



Aspect Weaving

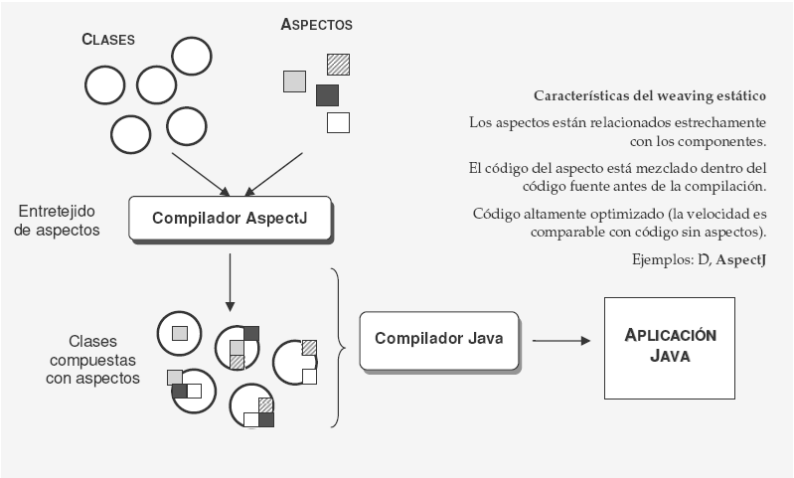
- A ***weaver*** combines classes and aspects
  - compiler / pre-processor / interpreter
  - statically or dynamically
  - unambiguously coordinates cross-cutting



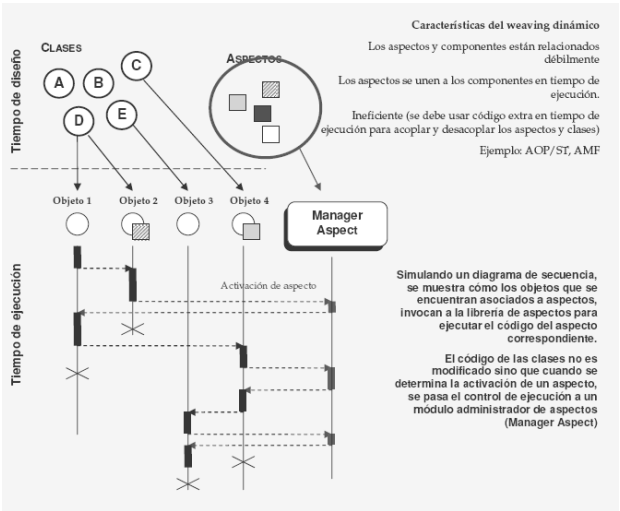
Kind of Weaving

- Static weaving
  - Sentences with the joint points are incorporated at the original code
  - Better performance
  - It is not possible to change the aspects dynamically
- Dynamic weaving
  - The aspects exist explicitly at compile and execution time
  - It is possible to add, change and delete aspects dynamically
  - Performance is bad and it is needed more memory

Static Weaving



Dynamic Weaving



Additional Concepts

- **Planes**
- **Composition Strategies**
- **Conflicts between aspects**

Planes: Encapsulating Aspects

Collection of aspects which carry out a specific functionality

- **Different levels of granularity**
- **Planes will facilitate the handling of groups of aspects, their reuse, their interaction with the rest of the system, their adaptability**
- **The concept of plane is not to be confused with the concept of level or hierarchy of levels**
- **Heterarchical vs. Hierarchical structure (*heterarchy: a form of organization resembling a network or fishnet)***

Planes: Encapsulating Aspects

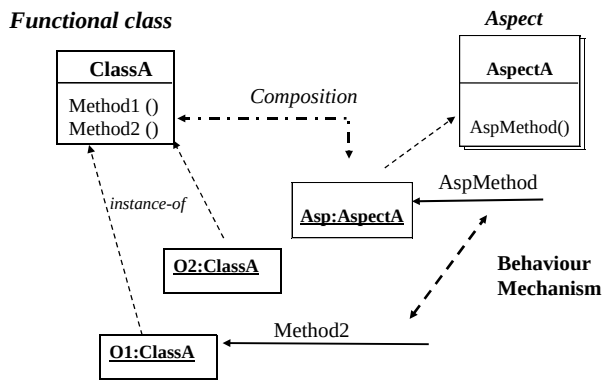
- **Encapsulation of Functionality**
- **Semantic sets**
- **Handling of set attributes**
- **Reusability**
- **Adaptability**

Composition Strategies

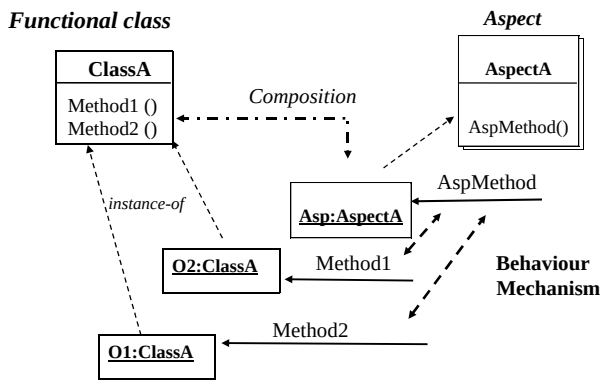
Different association strategies imply different join-points between aspects and other components

- Class composition: *the aspect code is activated when all objects of the reflected class receive a message*
- Method composition: *the aspect code is activated when all objects of the class receive a message with the reflected method*
- Object composition: *the aspect code is activated when the reflected object receives a message*
- Object-Method composition: *the aspect code is activated when the reflected object receives a message with the reflected method*

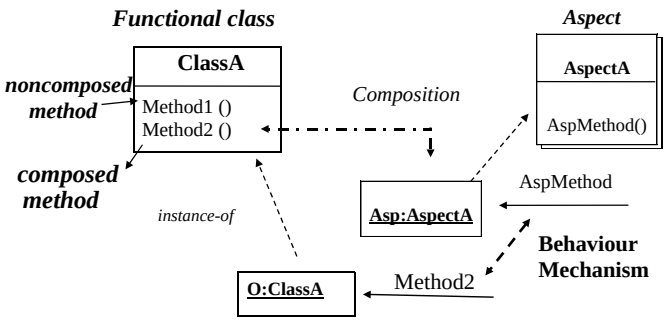
Class Composition



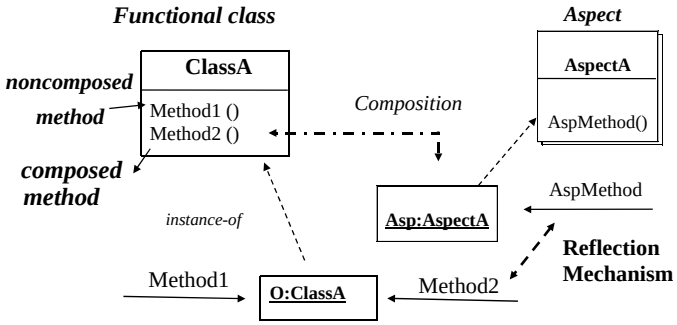
Class Composition



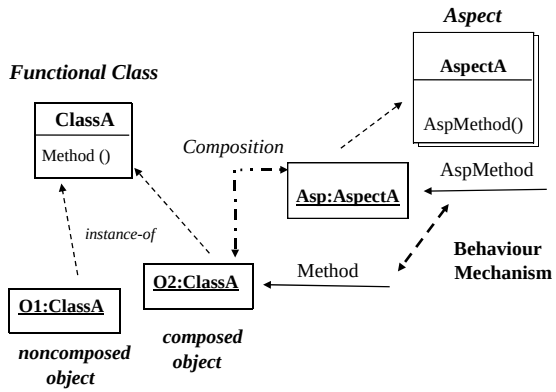
Method Composition



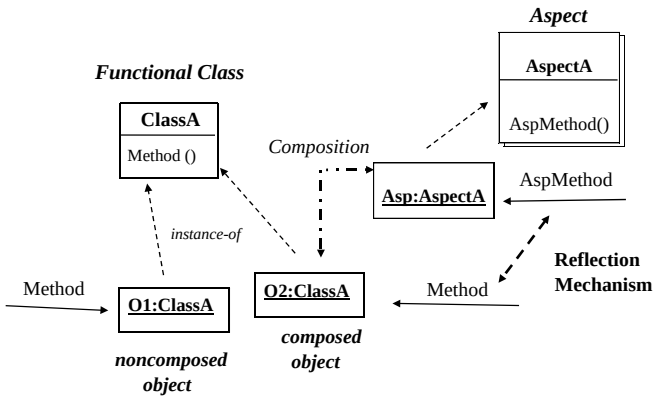
Method Composition



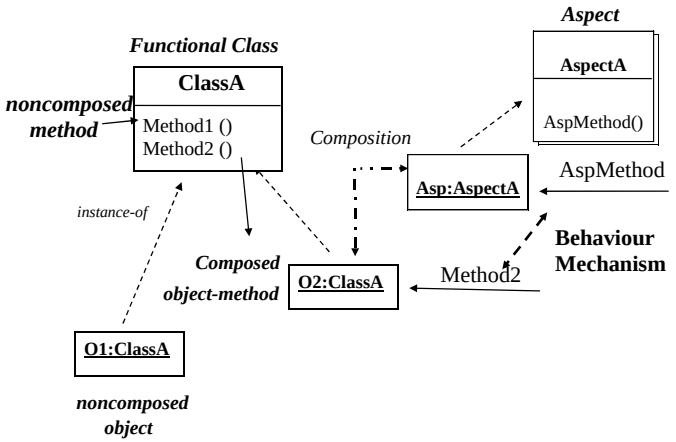
Object Composition



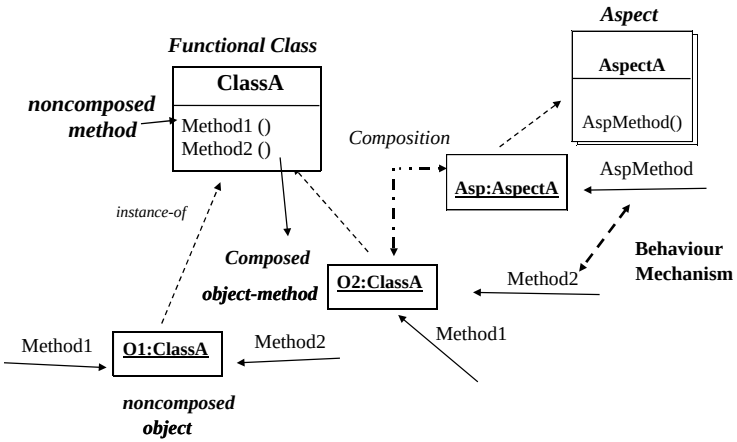
Object Composition



Object-Method Composition



Object-Method Composition



Aspect Activation Domain

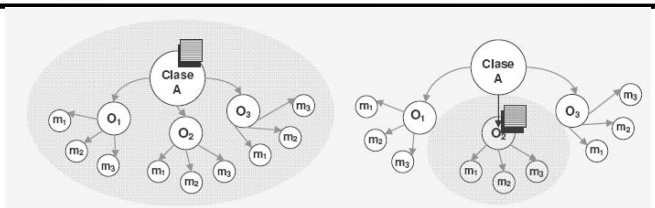


Figura 1. Dominio en asociación a Clase

Figura 2. Dominio en asociación a Objeto

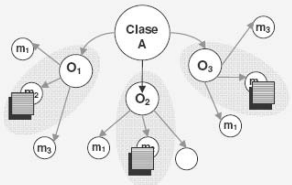


Figura 3. Dominio en asociación a Método

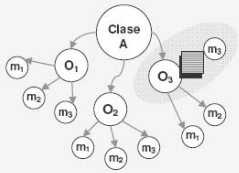


Figura 4. Dominio en asociación a Método-Objeto

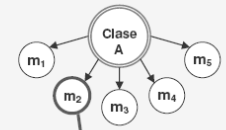
Activation of Aspect Code

The aspect code may be activated at different times:

- **Only** Before **the intercepted method**
- **Only** After **the intercepted method**
- Before and After **the intercepted method**
- Instead of **the intercepted method**

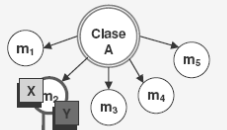
Conflicts between Aspects

A conflict may occur if two or more aspects compete for activation




```
public class A {
  public void m1() { }
  public void m2(int prop) {
        this.propiedad = prop;
        ManagerLogueo(prop);
        ManagerPersistencia(this);
  }
  public void m3() { }
  public void m4() { }
  public void m5() { }
} //fin de clase A
```

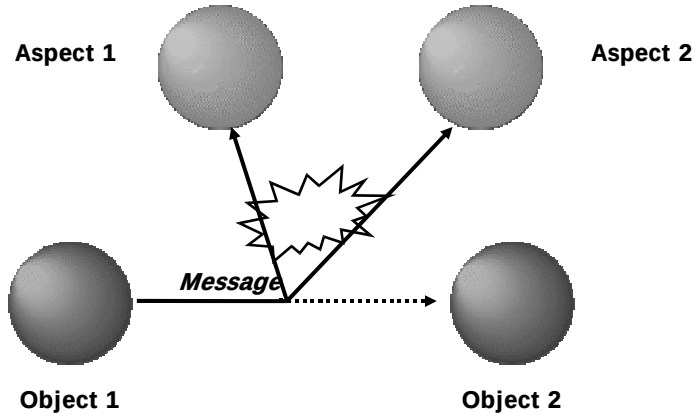
Figura 1. Lenguaje sin orientación a aspectos



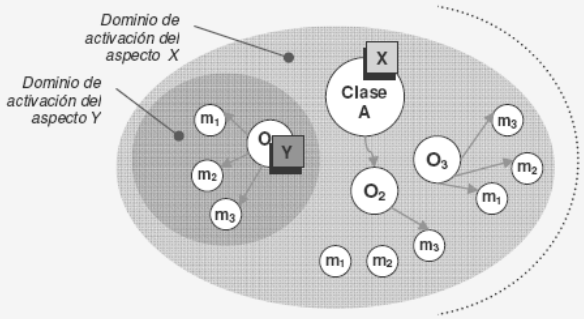

```
public class A {
  public void m1() { }
//Asociación del aspecto X (logueo)
//Asociación del aspecto Y (persistencia)
  public void m2(int prop) {
        this.propiedad = prop;
  }
  public void m3() { }
  public void m4() { }
  public void m5() { }
} //fin de clase A
```

Figura 2. Lenguaje orientado a aspectos

Conflicts



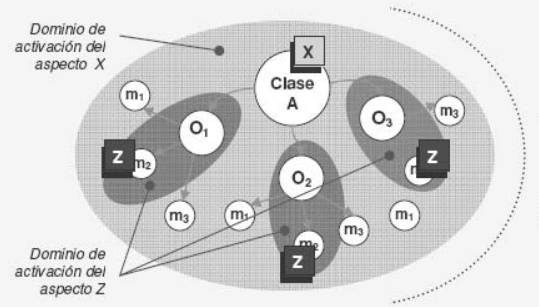
Conflicts between Aspects related to the Composition



Conflicto clase-objeto

Puede observarse que el dominio de activación del aspecto Y asociado al objeto 1 está incluido en el domino de activación del aspecto X asociado a la clase A. Cada vez que el objeto 1 sea activado, invoque sus métodos , sea llamado por otro objeto, etc., se detectará un conflicto entre ambos aspectos que deberá ser resuelto. En realidad debería estar considerada su resolución al momento de diseño .

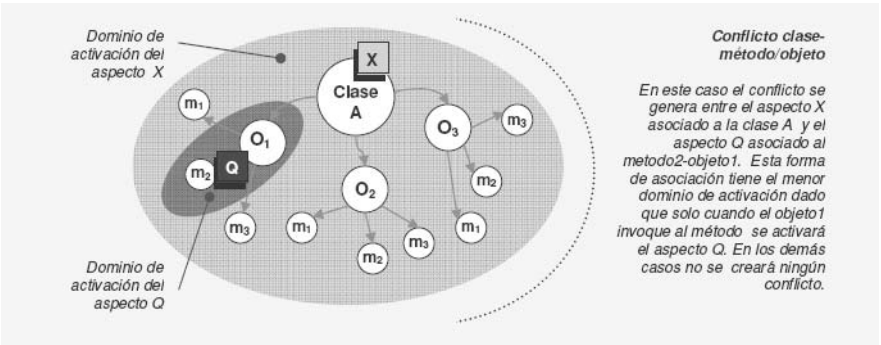
Conflicts between Aspects related to the Composition



Conflicto clase-método

El dominio de activación del aspecto Z se encuentra incluido en el dominio de activación del aspecto X, por lo tanto cada vez que cualquier objeto de la clase A invoque al método m2 se generará un conflicto entre ambos aspectos. Este tipo de conflicto puede ser detectado en tiempo de diseño. Si cualquier objeto invoca a otro método no se genera conflicto dado que el aspecto esta asociado a uno de ellos específicamente.

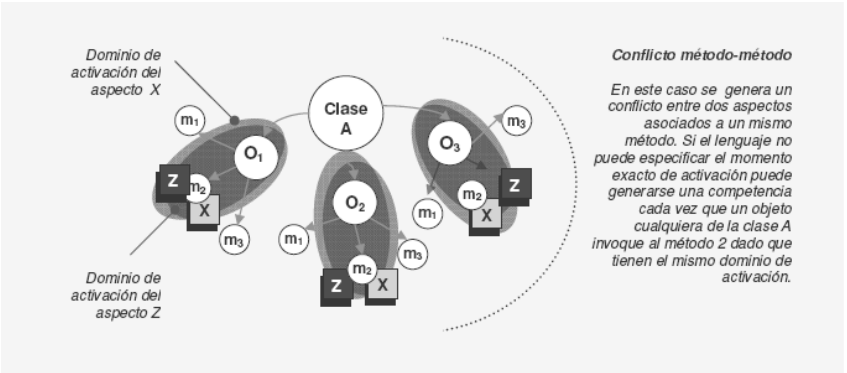
Conflicts between Aspects related to the Composition



Conflicto clase-método/objeto

En este caso el conflicto se genera entre el aspecto X asociado a la clase A y el aspecto Q asociado al metodo2-objeto1. Esta forma de asociación tiene el menor dominio de activación dado que solo cuando el objeto1 invoque al método se activará el aspecto Q. En los demás casos no se creará ningún conflicto.

Conflicts between Aspects related to the Composition



Conflicto método-método

En este caso se genera un conflicto entre dos aspectos asociados a un mismo método. Si el lenguaje no puede especificar el momento exacto de activación puede generarse una competencia cada vez que un objeto cualquiera de la clase A invoque al método 2 dado que tienen el mismo dominio de activación.

Conflicts between Aspects

- **Static conflict**: **detected when the association is established; similar to system restrictions**
- **Dynamic conflict**: **detected at run-time**

Detection and Solving Conflicts

- **Which aspects are to be executed?**
- **Activation order**

Which aspects are to be executed?

- **They can be defined at design time or if it depends on the execution time information it can be at execution time**
- **Null: Neither aspect will be activated**
- **Exclusive: Only one aspect**
- **Partial: An specific amount of aspects will be activated**
- **Total: All aspects**

Activation Order

- **Priority: Each aspect has a qualification**
- **Access to the data: Read, Write, Read/Write. Like in databases first the read aspects, the read/write and at the end the write ones**
- **Specificity: Aspects with the execution domain most specific. Method-Object composition are executed first**
- **Precedence: Order of aspects activation**

Conflict Taxonomy

- InOrder: **aspects are activated in the specified order**
- ReverseOrder: **aspects are activated in reverse order**
- Optional: **the system decides the order of activation (pre-established or random)**
- Exclusive: **only one aspect is activated**
- Null: **neither aspect is activated**
- Context-dependent: **the developer codes the specific activation policy of the aspects**

Potential Conflicts

Given two aspects, asp1 and asp2, the following cases could be conflictive:

- asp1 **and** asp2 **are associated to the same base class.**
- asp1 **and** asp2 **are associated to the same method and base class.**
- asp1 **and** asp2 **are associated to the same base object.**
- asp1 **and** asp2 **are associated to the same base object-method pair.**
- asp1 **is associated to a class and** asp2 **is associated to a method of that same class.**
- asp1 **is associated to a class and** asp2 **is associated to an object of that same class.**
- asp1 **is associated to a class and** asp2 **is associated to an object-method of an instance of that class.**
- asp1 **is associated to a method and** asp2 **is associated to an instance of the class to which that method belongs.**
- asp1 **is associated to a method and** asp2 **is associated to an object-method of the same method and instance of the class.**
- asp1 **is associated to an object and** asp2 **is associated to an object-method of that same object.**

Levels of Conflict

- Aspect-Aspect: **between two specific aspects**
- Aspect-Plane: **between an aspect and a plane (all the aspects of this plane)**
- Plane-Plane: **between two specific planes (all aspects of one plane with respect to all aspects of another)**
- Aspect-All: **between a specific aspect and all the others (in any plane)**

Aspect-Oriented Applications Implementation

- **Current programming languages extensions**
  - **A language to define the functional components**
  - **A language to define the aspects**
  - **An aspect weaver**
- **Frameworks**
- **Specific tools**

Aspect-Oriented Paradigm Benefits

- **Concerns can be added and deleted without modify the functional components**
- **Concerns are encapsulated in a same place**
- **Concerns can be designed/implemented independent to the functional components**
- **Aspects can be plug-in and plug-out without problem**

Aspect-Oriented Paradigm Drawbacks

- **Performance problems**
- **Language difference between the functional programming language and the aspectual programming language**
- **Aspects incompatibility. They work alone well but not together**