LINGI2252 – PROF. KIM MENS

* Slides partly based on presentations by Dr. M. D’Hondt, Dr. W. De Meuter & Dr. J. Brichau

ASPECT-ORIENTED PROGRAMMING

*
OVERVIEW OF THIS TALK

Modularity
Crosscutting concerns
Scattering and Tangling
Aspects
Conclusion
AspectJ
Worked-out example
OVERVIEW OF THIS TALK

Modularity

Crosscutting concerns

Scattering and Tangling

Aspects

Conclusion

AspectJ

Worked-out example
Modularity

How to construct “good” software systems?
that can be used over an extended period of years
that are easy to understand
of which parts can be reused in other software systems
that are easy to modify, maintain and evolve

“Modularity” is the key

Divide et impera
Structured Programming

Tangled code due to explicit goto statements

\[
i = 1
\]
TEST: if \( i < 4 \)
    then goto BODY
    else goto END
BODY: print(i)
    i = i + 1
    goto TEST
END:

Recognised common control structures

\[
i = 1
\]
while \( i < 4 \) {
    print(i)
    i = i + 1
}

capture program logic in a more explicit form
resulting code more clear, easier to write, maintain, debug, …
main () {
    draw_label("Haida Art Browser");
    m = radio_menu(
        {"Whale", "Eagle", "Dogfish"});
    q = button_menu{"Quit"};
    while ( ! check_buttons(q) ) {
        n = check_buttons(m);
        draw_image(n);
    }
}

draw_label (string) {
    w = calculate_width(string);
    print(string, WINDOW_PORT);
    set_x(get_x() + w);
}

draw_image (img) {
    w = img.width;
    h = img.height;
    do (r = 0; r < h; r++)
        do (c = 0; c < w; c++)
            WINDOW[r][c] = img[r][c];
}

radio_menu (labels) {
    i = 0;
    while (i < labels.size) {
        radio_button(i);
        draw_label(labels[i]);
        set_y(get_y() + RADIO_BUTTON_H);
        i++;
    }
}

button_menu (labels) {
    i = 0;
    while (i < labels.size) {
        draw_label(labels[i]);
        set_y(get_y() + BUTTON_H);
        i++;
    }
}

draw_circle (x, y, r) {
    %%primitive_oval(x, y, 1, r);
}

radio_button (n) {
    draw_circle(get_x(), get_y(), 3);
}

7
main() {
  draw_label("Haida Art Browser");
  m = radio_menu(
    {"Whale", "Eagle", "Dogfish"});
  q = button_menu({"Quit");
  while (!check_buttons(q)) {
    n = check_buttons(m);
    draw_image(n);
  }
}

radio_button(n) {
  draw_circle(get_x(), get_y(), 3);
}

draw_circle(x, y, r) {
  %primitive_oval(x, y, 1, r);
}

button_menu(labels) {
  i = 0;
  while (i < labels.size) {
    draw_label(labels[i]);
    set_y(get_y() + BUTTON_H);
    i++;
  }
}

draw_image(img) {
  w = img.width;
  h = img.height;
  do (r = 0; r < h; r++)
  do (c = 0; c < w; c++)
    WINDOW[r][c] = img[r][c];
}
... into Modules

```c
main () {
    draw_label("Haida Art Browser");
    m = radio_menu(
        {"Whale", "Eagle", "Dogfish"});
    q = button_menu({"Quit");
    while ( ! check_buttons(q) ) {
        n = check_buttons(m);
        draw_image(n);
    }
}
```

```c
draw_label (string) {
    w = calculate_width(string);
    print(string, WINDOW_PORT);
    set_x(get_x() + w);
}
```

```c
draw_image (img) {
    w = img.width;
    h = img.height;
    do (r = 0; r < h; r++)
        do (c = 0; c < w; c++)
            WINDOW[r][c] = img[r][c];
}
```

```c
draw_circle (x, y, r) {
    %primitive_oval(x, y, 1, r);
}
```

```c
radio_menu(labels) {
    i = 0;
    while (i < labels.size) {
        radio_button(i);
        draw_label(labels[i]);
        set_y(get_y() + RADIO_BUTTON_H);
        i++;
    }
}
```

```c
radio_button(n) {
    draw_circle(get_x(), get_y(), 3);
}
```

```c
button_menu(labels) {
    i = 0;
    while (i < labels.size) {
        draw_label(labels[i]);
        set_y(get_y() + BUTTON_H);
        i++;
    }
}
```

But... variations on modules remain incredibly complex
Object Orientation

- **Menu**
  - display: <promise>
  - click: <promise>

- **ButtonMenu**
  - display: boxes, labels
  - click: highlight

- **ToggleButtonMenu**
  - click: set/clear

- **CheckBoxMenu**
  - display: boxes, maybe checks, labels
  - click: set/clear

- **RadioButtonMenu**
  - display: circle, maybe dots, labels
  - click: set/clear
  - AND clear others
Tangling => modularity

structured control constructs

modules with narrow interfaces

classification & specialisation of objects

main () {
  draw_label("Haida Browser");
  m = radio_menu({"Whale", "Eagle", "Dogfish"});
  q = button_menu({"Quit"});
  while ( ! check_buttons(q) ) {
    n = check_buttons(m);
    draw_image(n);
  }
}
draw_label (string) {
  w = calculate_width(string);
  print(string, WINDOW_PORT);
  set_x(get_x() + w);
}
draw_image (img) {
  w = img.width;
  h = img.height;
  do (r = 0; r < h; r++)
    do (c = 0; c < w; c++)
      WINDOW[r][c] = img[r][c];
}

radio_menu(labels) {
  i = 0;
  while (i < labels.size) {
    radio_button(i);
    draw_label(labels[i]);
    set_y(get_y() + RADIO_BUTTON_H);
    i++;
  }
}

button_menu(labels) {
  i = 0;
  while (i < labels.size) {
    draw_label(labels[i]);
    set_y(get_y() + BUTTON_H);
    i++;
  }
}

radio_button (n) {
  draw_circle(get_x(), get_y(), 3);
}

classification & specialisation of objects

AbstractButton

ToggleButton

RadioButton

AbstractButton

Button

ToggleButton

RadioButton

Checkbox
OVERVIEW OF THIS TALK

Modularity

Crosscutting concerns

Scattering and Tangling

Aspects

Conclusion

AspectJ

Worked-out example
Sometimes still tangled code
Good modularity:
A specific concern is handled by code in a single class

org.apache.tomcat ➔ XML parsing

[Picture taken from the aspectj.org website]
**OO : good modularity (2)**

org.apache.tomcat  ➔  URL pattern matching

[Picture taken from the aspectj.org website]

**Good modularity:**
A specific concern that is handled by code in two different classes related by inheritance
Bad modularity:
A specific concern that is handled by code that is “scattered” over almost all classes
Crosscutting Concerns

Concern

‘Something the programmer should care about’

Ideally implemented in one single module

The “crosscutting” phenomenon

Implementation is spread across other modules

Difficult to understand, change, maintain, etc…

Tyranny of the Dominant Decomposition

Given one of many possible decompositions of the problem…

(mostly core functional concerns)

…then some subproblems (concerns) cannot be modularised!

non-functional, functional, added later on, …
Crosscutting Concerns

Crosscutting is inherent in complex systems

E.g., logging code in the code of the Apache Web Server
not in a single place; not even in a small number of places;
it “cuts across” the “dominant decomposition”

Nevertheless, such crosscutting concerns often do

have a clear purpose  \textit{What}
have some regular interaction points  \textit{Where}

AOP proposes to capture crosscutting concerns explicitly...

in a modular way
with programming language support
and with tool support
Tangling => modularity

structured control constructs

modules with narrow interfaces

classification & specialisation of objects

aspects
OVERVIEW OF THIS TALK

Modularity

Crosscutting concerns

Scattering and Tangling

Aspects

Conclusion

AspectJ

Worked-out example
Bad modularity: scattering & tangling

- **scattering**
  code addressing one concern is spread throughout the entire program

- **tangling**
  code in one region addresses multiple concerns

- **scattering** and **tangling** tend to appear together – they describe different facets of the same problem
Cost of scattered and tangled code

• Redundant code
  • Same (or similar) fragment of code in many places

• Difficult to reason about
  • The big picture isn’t clear

• Difficult to change
  • Difficult to find all the code involved...
  • ...and be sure to change it consistently
Good modularity:
clean separation of concerns

• **separated**
  implementation of a concern can be treated as separate entity or module

• **localised**
  implementation of a concern appears in one part of a program

• **modular**
  concern has a well-defined interface to the rest of the system
Scattering & tangling: a first example

every call to foo is preceded by a log call

```java
class Helper {
    public static void foo(int n) {
        System.out.println("foo called");
        Helper.foo(n/3);
    }
}
```

```java
System.out.println("foo called");
Helper.foo(n/3);
```

```java
System.out.println("foo called");
Helper.foo(i+j+k);
```

```java
System.out.println("foo called");
Helper.foo(x);
```

```java
class Helper {
    public static void foo(int n) {
        System.out.println("foo called");
        Helper.foo(i+j+k);
    }
}
```
Scattering & tangling: a first example – solution

procedures can modularize this case (unless logs use calling context)

class Helper {
    public static void foo(int n) {
        System.out.println("foo called");
        ...
    }
}

Helper.foo(n/3);
Helper.foo(i+j+k);
Helper.foo(x);
Scattering & tangling: a second example

all subclasses have an identical method

FigureElement

- moveBy(int, int)

Point

- getX()
- getY()
- setX(int)
- setY(int)
- moveBy(int, int)
- draw()
- refresh()

Line

- getP1()
- getP2()
- setP1(Point)
- setP2(Point)
- moveBy(int, int)
- draw()
- refresh()
Scattering & tangling: a second example – solution

inheritance can modularize this

FigureElement

- moveBy(int, int)
- refresh()

Point

- getX()
- getY()
- setX(int)
- setY(int)
- moveBy(int, int)
- draw()

Line

- getP1()
- getP2()
- setP1(Point)
- setP2(Point)
- moveBy(int, int)
- draw()
several methods that end with a call to:

Display.update();
after():
call(void FigureElement+.set*(..)) ||
call(void FigureElement.moveBy(int,int))
{
  Display.update();
}

Point
- getX()
- getY()
- setX(int)
- setY(int)
- moveBy(int, int)
- draw()

Line
- getP1()
- getP2()
- setP1(Point)
- setP2(Point)
- moveBy(int, int)
- draw()

FigureElement
- moveBy(int, int)
- refresh()

AspectJ

Pointcut expression - describes set of join points
OVERVIEW OF THIS TALK

Modularity

Crosscutting concerns

Scattering and Tangling

Aspects

Conclusion

AspectJ

Worked-out example
AOP History

Reflection

Meta-object protocol (MOP)

Control over method invocation, instance creation, etc…

Often used to implement crosscutting concerns

Considered too powerful and too difficult

Aspect Oriented Programming (AOP)

Provide necessary abstractions to implement crosscutting concerns

“a poor man’s reflection”

Is often implemented through meta programming and reflection
The AOP Idea: aspects

Main idea is to describe crosscutting concerns as separate, independent entities, called aspects.

+ weaving

aspect 1
aspect 2
aspect 3

core application functionality
The AOP Idea: weaving

At software development time, aspects and classes are kept as two, separate dimensions.

At run-time however, both dimensions need to be combined in some way for obtaining the final product.

This process of combining both dimensions is generally referred to as “weaving”.

Typically at compile-time (AspectJ)

Aspects are physically woven into the classes that make up the base application by source-code transformations.
The AOP Idea

Aspect 1

Aspect 2

Aspect 3

Aspect Weaver

core application functionality

woven output code
AOP: Terminology

Base program
- core functionality of your (OO) program

Aspect
- modularisation of a crosscutting concern

Weaver
- composes/compiles aspects into base program

Join Point
- particular point in the base program where/when an aspect can be woven

Pointcut
- concise description of a set of join points
AOP : Typical Examples

Classic Non-Functional Examples
- Synchronisation
- Logging
- Error Handling
- Persistence

Some Functional Examples
- Business-rules
- Language Internationalisation
- Personalisation of an e-commerce application
AOP : concrete example

```java
class Buffer {
    char[] data;
    int nrElements;

    char get() { ... };
    void put(char c) { ... };

    bool isEmpty() {
        return (nrElements==0) }
}
```

Functionality
AOP : concrete example

class Buffer {
    char[] data;
    int nrElements;
    Semaphore threads;

    char get() { ... };
    void put(char c) { ... };

    bool isEmpty() {
        bool result;
        threads.lock();
        result = (nrElements==0);
        threads.unlock();
        return result
    }
}
A better solution ...

Easier to
- understand
- maintain

class Buffer {
    char[] data;
    int nrElements;

    char get() { ... };
    void put(char c) { ... };

    bool isEmpty() {
        return (nrElements==0) }
}

before : reception(Buffer.isEmpty)
{ threads.lock() }
after: reception(Buffer.isEmpty)
{ threads.unlock() }
AOP : concrete example

class Buffer {
    char[] data;
    int nrElements;

    char get() { ... };
    void put(char c) { ... };

    bool isEmpty() {
        return (nrElements==0) 
    }
}

When a Buffer object receives the message isEmpty, first make sure that the object is not being accessed by another thread.

before : reception(Buffer.isEmpty)
{ threads.lock() } 
after: reception(Buffer.isEmpty)
{ threads.unlock() }
AOP: concrete example

**Aspect:**

Pointcut =
when to execute the aspect
+
Advice =

Weaver directive:
composition of
when and what

Aspect functionality:
what to do at join points

---

When a Buffer object receives the message isEmpty,
first make sure that the object is not being accessed

before: reception(Buffer.isEmpty)
{ threads.lock() }

after: reception(Buffer.isEmpty)
{ threads.unlock() }
Other concrete examples

Logging
“write something on the screen/file every time the program does X”

Error Handling
“if the program does X at location L then do Y at location K”

Persistence
“every time the program modifies the variable v in class C, then dump a copy to the DB”

User Interfaces
“every time the program changes its state, make sure the change is reflected on the screen”
OVERVIEW OF THIS TALK

Modularity

Crosscutting concerns

Scattering and Tangling

Aspects

Conclusion

AspectJ

Worked-out example
Conclusion

If you can think about something in a modular way but have trouble moulding it into a module it can probably be modelled as an aspect.

Traditionally, aspects were often non-functional... 
... but aspects can describe crosscutting functionality as well.

Invented in the mid 90’s it gained a lot of attention but seems to have lost momentum now.

Nevertheless it remains an interesting new way of thinking about decomposing a software system.

When AOP was first introduced, many OOP people “recognized” their own work as AOP “avant la lettre”
OVERVIEW OF THIS TALK

Modularity

Crosscutting concerns

Scattering and Tangling

Aspects

Conclusion

AspectJ

Worked-out example
AspectJ

- An aspect-oriented programming language
  - for Java
  - one of the “first” and most mature AO languages
  - created by the inventors of AOP
  - seamlessly integrated with the Eclipse IDE
    - by means of the AspectJ Development Tools (AJDT) plug-in
- Other aspect languages exist
  - JAsCo, CaesarJ, AspectS, Carma, Object Teams, HyperJ, JBOSS AOP, Compose*, DemeterJ, AspectC++, ...
  - they differ in the advice models, join point models and pointcut languages they offer
Eclipse

Universal tool platform
Open extensible IDE
Language-independent
Open-source
Very popular in Java community
Plug-in architecture

for example the AJDT plug-in (AOP plug-in for Java)

www.eclipse.org
www.eclipse.org/ajdt
AspectJ Development Tools (AJDT)
public void setRelations(List relations) {
    if (relations.size() > 0) {
        this.relations = relations;
    }
}
Compatible extension to Java:

- upward compatibility (Java program => AspectJ)
- platform compatibility (use regular JVM)
- attempt to make a small addition to Java

General-purpose rather than domain-specific

- not dedicated to a specific kinds of aspects (like security)
- can handle all kinds of aspects

Balance of declarative & imperative constructs

- pointcuts are a mixture of java fragments and declarative wildcards

Statically typed, uses Java’s static type system
OVERVIEW OF THIS TALK

Modularity

Crosscutting concerns

Scattering and Tangling

Aspects

Conclusion

AspectJ

Worked-out example
Change notification

Update some “view” whenever the state of some “customer” object is updated
A Simple Java Program

Customers in some business application

```java
public class Customer {
    private Address address;
    private String lastName;
    private String firstName;
    private CustomerID id;
    ...

    public Customer() {...}

    public Address getAddress() { return this.address; }

    public String getLastName() {
        return this.lastName;
    }

    public void setLastName(String name) {
        this.lastName = name;
    }
}
```
A Simple Concern

Change notification

Update some “view” whenever the state of a customer object the view is displaying is updated

Typical Java implementation

“Listeners” notify the view of updates that have occurred

• **Customer** class has methods to add and remove listeners;
• Calls `notifyListeners` method after every state-changing operation;
• Idem for other classes in the Customer hierarchy.
public class Customer {
    ...
    private CustomerID id;
    ...
    public Address getAddress() { return this.address; }
    public void setLastName(String name) {
        this.lastName = name;
    }
    public void setFirstName(String name) {
        this.firstName = name;
    }
    ...
}

public class CustomerListener extends Listener {
    public void notify(Customer modifiedCustomer) {
        System.out.println("Customer " + modifiedCustomer.getID() + " was modified");
    }
}

Customer Change Notification
public class CustomerListener extends Listener {

    public void notify(Customer modifiedCustomer) {
        System.out.println("Customer "+ modifiedCustomer.getID() + " was modified");
    }
}

public class Customer {

    ... private CustomerID id;
    ... private Collection listeners;
    ...
    public Address getAddress() { return this.address; }
    public void setLastName(String name) {
        this.lastName = name;
        notifyListeners();
    }
    public void setFirstName(String name) {
        this.firstName = name;
        notifyListeners();
    }

    ... public void addListener(CustomerListener listener) { listeners.add(listener); }
    public void removeListener(CustomerListener listener) { listeners.remove(listener); }
    public void notifyListeners() {
        for (...) {
            ... listener.notify(this); ...
        }
    }
    ...
}
public class CustomerListener {
    public void notify(Customer modifiedCustomer) {
        System.out.println("Customer "+ modifiedCustomer.getID() + " was modified");
    }
}

class Customer {
    private CustomerID id;
    private Collection listeners;

    public Address getAddress() { return this.address; }
    public void setLastName(String name) {
        this.lastName = name;
        notifyListeners();
    }
    public void setFirstName(String name) {
        this.firstName = name;
        notifyListeners();
    }

    public void addListener(CustomerListener listener) {
        listeners.add(listener);
    }
    public void removeListener(CustomerListener listener) {
        listeners.remove(listener);
    }
    public void notifyListeners() {
        for (...) {
            ... listener.notify(this); ...
        }
    }
}

Change Notification: Tangling

tangling
code in one region
addresses multiple concerns
Change Notification: Scattering

Code addressing one concern is spread around the code.

```java
public abstract class Customer {
    ...
    private CustomerID id;
    private Collection listeners;
    ...
    private CustomerID id;
    ...
    public void setCustomerID(String id) {
        this.id = id;
        notifyListeners();
    }
    ...
}

public class CorporateCustomer {
    ...
    private String companyName;
    private CompanyName taxNumber;
    ...
    public void setCompanyName(String name) {
        this.companyName = name;
        notifyListeners();
    }
    ...
}

public class PrivateCustomer {
    ...
    private String lastName;
    private String firstName;
    ...
    public void setLastName(String name) {
        this.lastName = name;
        notifyListeners();
    }
    public void setFirstName(String name) {
        this.firstName = name;
        notifyListeners();
    }
    ...
}
```
A Simple Concern

Change notification

Update some “view” whenever the state of a customer object the view is displaying is updated

Typical Java implementation

Listeners which notify the view of updates that have occurred

AspectJ implementation

Now let us refactor the traditional solution into an AspectJ solution
AspectJ Implementation (Step 1)

```java
public class Customer {
    private CustomerID id;
    private Collection listeners;

    public Address getAddress() { return this.address; }
    public void setLastName(String name) {
        this.lastName = name;
        notifyListeners();
    }
    public void setFirstName(String name) {
        this.firstName = name;
        notifyListeners();
    }

    public void addListener(CustomerListener listener) {
        listeners.add(listener);
    }
    public void removeListener(CustomerListener listener) {
        listeners.remove(listener);
    }
    public void notifyListeners() {
        for (...) {
            ... listener.notify(this); ... }
    }
    ...
}

public aspect ChangeNotification {
    pointcut stateUpdate(Customer c) :
        execution(* Customer.set*(..)) &&
        this(c);
    after(Customer c) : stateUpdate(c) {
        c.notifyListeners();
    }
}
```

**pointcut expression** - describes set of join points

**advice code**
public aspect ChangeNotification {

    pointcut stateUpdate(Customer c) :
        execution(* Customer.set*(..)) &&
        this(c);

    after(Customer c): stateUpdate(c) {
        for (Iterator iterator = c.listeners.iterator(); iterator.hasNext();)
            CustomerListener listener = (CustomerListener) iterator.next();
            listener.notify(c);
    }
}
public aspect ChangeNotification {

  pointcut stateUpdate(Customer c) :
    execution(* Customer.set*(..)) &&
      this(c);

  after(Customer c): stateUpdate(c) {
    for (Iterator iterator = c.listeners.iterator(); iterator.hasNext();)
      CustomerListener listener = (CustomerListener) iterator.next();
      listener.notify(c);
  }

  private Collection Customer.listeners = new LinkedList();

  public void Customer.addListener(CustomerListener listener) {
    listeners.add(listener); }

  public void Customer.removeListener(CustomerListener listener) {
    listeners.remove(listener); }
}
Clean Separation Of Concerns

```java
public class Customer {
    private Address address;
    private String lastName;
    private String firstName;
    private CustomerID id;

    public Customer() { ... }

    public Address getAddress() { return this.address; }
    public String getLastName() { return this.lastName; }

    public void setLastName(String name) { this.lastName = name; }
    public void setFirstName(String name) { this.firstName = name; }
}
```

```java
public aspect ChangeNotification {
    pointcut stateUpdate(Customer c) :
        execution(* Customer.set*(..)) &&
        this(c);

    after(Customer c): stateUpdate(c) {
        for (Iterator iterator = c.listeners.iterator(); iterator.hasNext();)
            CustomerListener listener = (CustomerListener) iterator.next();
            listener.notify(c); }

    private Collection Customer.listeners = new LinkedList();
    public void Customer.addListener(CustomerListener listener) {
        listeners.add(listener); }
    public void Customer.removeListener(CustomerListener listener) {
        listeners.remove(listener); }
}
```

**Crosscutting concern:**

**Change notification**

**Base concern:**

**Customer handling**
A Simple Concern

Change notification

Update some “view” whenever the state of a customer object the view is displaying is updated

Typical Java implementation

Listeners which notify the view of updates that have occurred

AspectJ implementation

Let’s refactor the traditional solution into an AspectJ solution.

AJDT tool support

Makes AspectJ development much easier

Especially for Java programmers familiar with Eclipse
AJDT Package Explorer
AJDT Java Editor is Aspect-Aware

```java
public Customer() {
    this.id = new CustomerID(Customer.idCounter++);
    this.address = new Address();
}

public Address getAddress() {
    return this.address;
}

public String getLastName() {
    return this.lastName;
}

public String getId() {
    return (this.id).toString();
}

public void setFirstName(String name) {
    this.firstName = name;
}

public String toString() {
    return "Customer " + id + "\n" + firstName + " " + lastName + "\n" + address + "\n";
}
```
```java
package businessappaspects;

// version 3 : add these lines
import java.util.Collection;

public aspect ChangeNotification {

    pointcut stateUpdate(Customer c):
        execution(* Customer.set*(..)) &&
        this(c);

    after(Customer c): stateUpdate(c) {
        for (Iterator iterator = c.listeners.iterator(); iterator.hasNext();)
            CustomerListener listener = (CustomerListener) iterator.next();
            listener.notify(c);
    }

    private Collection Customer.listeners = new LinkedList();

    public void Customer.addListener(CustomerListener listener) {
        listeners.add(listener);
    }

    public void Customer.removeListener(CustomerListener listener) {
        listeners.remove(listener);
    }
}
```
```java
public String getID() {
    return (this.id).toString();
}

public void setLastName(String name) {
    this.lastName = name;
}

public void setFirstName(String name) {
    this.firstName = name;
}

public String toString() {
    return "Customer " + id + "\n" + firstName + " " + lastName + "\n" + address + "\r";
}

public static void main(String[] args) {
}
AJDT Debugger
Learning objectives:
- Definition and difference between maintenance, evolution, reuse
- Different types of maintenance
- Causes for maintenance and change
- Technic
- Different types of evolution
- Re evolution
POSSIBLE QUESTIONS

✧ Explain, in your own words, what problem aspect-oriented programming tries to solve.

✧ Explain, in your own words, what a crosscutting concern is, and illustrate it with a concrete example.

✧ Explain what the tyranny of the dominant decomposition means, and discuss its relation with aspect-oriented programming.

✧ Explain the notions of tangling and scattering, and illustrate them with a concrete example. What are the problems with having tangled and scattered code?

✧ Explain, in your own words, what an aspect weaver is and how aspect-oriented programming works.

✧ Explain, in your own words, the following concepts from aspect-oriented programming: base program, aspect, join point, pointcut and advice. Illustrate with a concrete example.
CLASS... IS... DISMISSED.