Aspect Oriented Programming

Aspect-oriented programming

- Some components of system may occur in many different places
  - login/security
  - logging
  - validation

- Not surprising, but potentially painful
  - how many classes or methods should start with

```
def doSomething
  login-verify
  really do something
end
```

- Suppose we want to add this code later? – that's also a problem

AOP simplifies this process

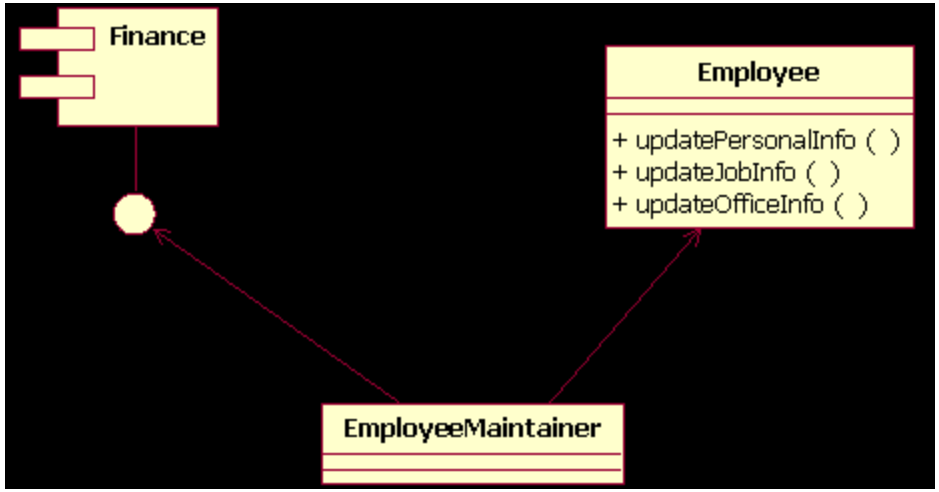
- cross-cutting concern
  - comes up in multiple classes in multiple places
  - adding or removing can be difficult
    - what if you forgot adding authorization in one class

- Provide an aspect - to code cross-cutting concerns - incorporate them into the system.

- Join point – spot in the code where the concerns crosscut

AOP Examples

- Say we want to log every time we update employee info.
  - in both the employee class and the finance class



  - example methods in finance class
    - updateFederalTaxInfo (Employee e)

Pollice, G. "A look at aspect-oriented programming"

Adding logging code: three steps

1. Identify places in the code where we want to insert the logging code
2. Write the logging code
3. Compile the new code and integrate it into the system

Examples here are using AspectJ

Identify the join points

```
pointcut employeeUpdates(Employee e):
  call(public void Employee.update*Info()) &&
  target(e);
```

- any methods called update*Info in Employee

```
pointcut employeeFinanceUpdates(Employee e) :
  call (public void update*Info(Employee)) &&
  args(e);
```

- all methods called update*Info that take Employee as an argument

Pollice, G. "A look at aspect-oriented programming"

Writing the aspect (logging) code

```
public aspect EmployeeChangeLogger {
   pointcut employeeUpdates(Employee e) : call(
         public void Employee.update*Info()) && target(e);

   pointcut employeeFinanceUpdates(Employee e) : call(
         public void update*Info(Employee)) && args(e);

   after(Employee e) returning : employeeUpdates(e) ||
                        employeeFinanceUpdates(e) {
         System.out.println("\t>Employee : " +e.getName() +
                              " has had a change ");
         System.out.println("\t>Changed by " +
                              thisJoinPoint.getSignature());
   }
}
```

advice

can be before, after, around

Pollice, G. "A look at aspect-oriented programming"

Compiling new code and integrating

- Once the aspects are written, we can use the AspectJ compiler (ajc).
  - AJC will *weave* the aspect into our compiled code
  - AJC does it at the Java bytecode level (can also do source)

Pollice, G. "A look at aspect-oriented programming"

Output

- Before weave:

```
Updating job information
Updating federal tax information
```

after weave:

```
Updating job information
  >Employee : Chris Smith has had a change
  >Changed by void employee.Employee.updateJobInfo()

Updating federal tax information
  >Employee : Chris Smith has had a change
  >Changed by void
  employee.EmployeeFinance.updateFederalTaxInfo(Empl
  oyee)
```


Pollice, G. "A look at aspect-oriented programming"

Benefits with AOP

- Encapsulate cross-cutting concerns in the system
- Enhance maintainability
  - add new features
  - incrementally improve
- Improve testing, insert debug code automatically

Pollice, G. "A look at aspect-oriented programming"

Issues with AOP

- Problems with code review
  - Can't reason for the code just by looking at the class, got to look at the aspects too
  - Will somebody insert new aspects into the code after you have reviewed it?
- AspectJ
  - requires additional compiler for Java
  - eclipse plug-in
  - before, after as in Rails
- Code harder to read
  - don't know what's going on where
  - but this may be a generally-valid complaint with Rails
- May cause problems with testing

AOP and Rails

- Rails
  - filters, validation, observers can be effective forms of AOP
  - Idiomatically supported
    - Rails programmers should learn how to use it