Introduction to Aspect Oriented Programming and Aspect Matlab

Motivation for Aspect Oriented Programming

```
void transfer(Account from, Account to, int amount, User user, Logger logger)
  throws Exception {
  logger.info("Transferring money...");
  if (!checkUserPermission(user)){
    logger.info("User has no permission.");
    throw new UnauthorizedUserException();
  }
  if (from.getBalance() < amount) {
    logger.info("Insufficient funds.");
    throw new InsufficientFundsException();
  }
  from.withdraw(amount);
  to.deposit(amount);
  logger.info("Successful transaction.");
}
```

Motivation for Aspect Oriented Programming

```
void transfer(Account from, Account to, int amount, User user, Logger logger)
  throws Exception {
  logger.info("Transferring money...");
  if (! checkUserPermission(user)){
    logger.info("User has no permission.");
    throw new UnauthorizedUserException();
  }
  if (from.getBalance() < amount) {
    logger.info("Insufficient funds.");
    throw new InsufficientFundsException();
  }
  from.withdraw(amount);
  to.deposit(amount);
  logger.info("Successful transaction.");
}
```

The basic functionality is simply transferring money from one account to another, but other interests get tangled together with this simple functionality.

Cross-Cutting Concerns

Certain portions of a program cannot be neatly represented using objects, and are scattered throughout the code. These portions of the program are referred to as cross-cutting concerns, problems which require cutting across multiple abstractions of a program.

These concerns raise a few problems:

Cross-Cutting Concerns

Certain portions of a program cannot be neatly represented using objects, and are scattered throughout the code. These portions of the program are referred to as cross-cutting concerns, problems which require cutting across multiple abstractions of a program.

These concerns raise a few problems:

- Code will be difficult to maintain or modify.

Cross-Cutting Concerns

Certain portions of a program cannot be neatly represented using objects, and are scattered throughout the code. These portions of the program are referred to as cross-cutting concerns, problems which require cutting across multiple abstractions of a program.

These concerns raise a few problems:

- Code will be difficult to maintain or modify.
- Code will be redundant.

Cross-Cutting Concerns

Certain portions of a program cannot be neatly represented using objects, and are scattered throughout the code. These portions of the program are referred to as cross-cutting concerns, problems which require cutting across multiple abstractions of a program.

These concerns raise a few problems:

- Code will be difficult to maintain or modify.
- Code will be redundant.
- Code will be less clear.

Aspect Oriented Programming

- Aspect oriented programming solves inherent issues with cross-cutting concerns by separating them into stand alone modules called aspects.
- Aspect oriented languages weave aspect code into existing object oriented code
- Aspect oriented languages define join points, which are well-defined points in a program where code can be meaningfully inserted.
- Specific groups of join points can be specified using pointcuts. A pointcut is used to determine if a given join point matches some specification.
- Programmers can specify code to be run at join points. This code can be inserted when a desired pointcut matches a portion of code in the main program.

Aspect Oriented Programming - Example

```
void transfer (Account fromAcc, Account toAcc, int amount) throws Exception {
  if (fromAcc.getBalance() < amount) {
    throw new InsufficientFundsException ();
  }
  fromAcc.withdraw(amount);
  toAcc.deposit(amount);
}
```

AOP allows for us to separate out cross-cutting concerns from our main program, leaving only the basic functionality. Cross-cutting concerns are added into their own aspect modules which specify what code should be added (woven) into the main program and where.

AspectMatlab

- Extension of Matlab
- Intended to be easy to use
- Would like a library of predefined useful aspects
- Supports patterns (pointcuts), actions
- Focused more on profiling/checking functionality
- Designed to have patterns which deal with constructs frequently encountered in Matlab

AspectMatlab

- An aspect is named and contains a body
- Aspects are designed to be similar to object-oriented MATLAB classes, which allow for properites blocks and methods blocks.
- In addition, includes patterns blocks and action blocks. Patterns are used to pick out specific join points, and actions are the blocks of code intended to be executed at specific join points.

AspectMatlab - An Example Aspect

```
aspect myAspect
 properties
   count = 0;
end
methods
   function incCount(this)
   this.count = this.count + 1;
end
end
patterns
   callFoo : call(foo);
end
actions
   actCall : before callFoo
   this.incCount();
end
end
```

Patterns

- Patterns are contained inside blocks, a particular aspect can have any number of blocks of patterns.
- Patterns are identified by a unique name, and consist of a pattern designator which identifies which MATLAB constructs should be targeted by the pattern.
- In addition, includes patterns blocks and action blocks. Patterns are used to pick out specific join points, and actions are the blocks of code intended to be executed at specific join points.

```
            Name of pattern
patterns         /            ____Pattern designator
         callFoo : call(foo);
end
```

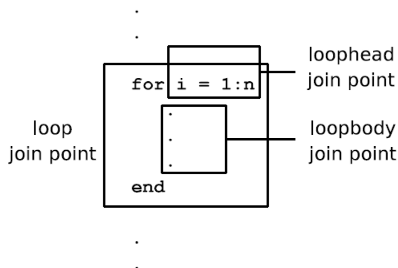
Patterns - Join Points in Matlab

AspectMatlab provides patterns to match a variety of join points

- Loops (loop,loopbody,loophead)
- Function calls (call)
- Function executions (execution)
- Array accesses (get)
- Array assignments (set)
- Annotations (annote)

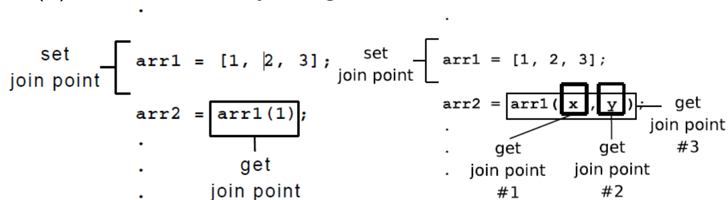
Patterns - Loops

- loop(i) - Matches the outside of the loop that iterates over i
- loopbody(i) - Matches the body of the loop that iterates over i
- loophead(i) - Matches the head of the loop that iterates over i



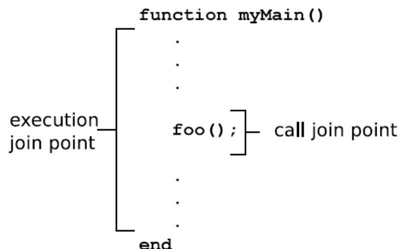
Patterns - Arrays

- get(x) - Matches array accesses of x
- set(x) - Matches array assignments to x



Patterns - Functions

- call (foo) - Matches calls to the function foo
- execution (foo) - Matches the entire body of function foo
- mainexecution() - Matches the execution of the first function/script executed.



Patterns - Selective Matching

Specialization of patterns for arrays and function calls can based on the indices/arguments used for assignments/function calls

call(foo)	matches all calls to `foo` (function or script)
call(foo())	matches calls with no arguments (function or script)
call(foo(*))	matches calls with exactly one argument (function only)
call(foo(..))	matches calls with 1 or more argument(s) (function only)
call(foo(*,..))	matches calls with 2 or more arguments (function only)
	...and so on
set(arr)	matches all assignments to `arr`
set(arr())	matches assignments with no indices
set(arr(*))	matches assignments with exactly one index
set(arr(..))	matches assignments with 1 or more index/indices
set(arr(*,..))	matches assignments with 2 or more indices
	...and so on

Patterns - Annotation

In order to make AspectMatlab easier to use, we introduce annotations to the base Matlab language. Annotations are Matlab comments which the AspectMatlab compiler recognizes and considers to be join points. Annotations take the form %@annotationname. Optionally, a list of arguments may follow the annotation name, which can be used in the woven aspect code.

```
function [F, V] = nbody3d(n, R, m, dT, T)

%@type n "double" [1,1]
%@type R "double" [n,3]
%@type m "double" [n,1]
%@type dT "double" [1,1]
...
```

Patterns - Annotation

The annote pattern can be used to match annotations. The pattern which will be matched can be restricted by specifying expected arguments. Recognized argument types are var, char and double, as well as arrays of these types.

```
patterns
  annoteEx: annote( plot );
  annoteAdd: annote(add(double,double));
  typeAnn : annote(type(var, char, [*]));
end
```

Patterns - Compound Patterns

Compound patterns can be made of primitive patterns using | (or) and & (and)

```
patterns
  pCallFoo :  call (foo) & within(loops, *);
  pGetOrSet : (get(*) | set (*)) & within(function, bar);
  pCallExec : pCallFoo | execution(foo);
end
```

Patterns - Scope Restriction

The within pattern can be used to restrict the scope of pattern application. It matches all join points that occur within a function, script, class, or loop. It can be meaningfully applied using compound patterns to only match within certain constructs.

```
patterns
  pWithinFoo : within(function, foo);
  pWithinBar : within(script, bar);
  pWithinMyClass : within(class, myClass);
  pWithinLoops : call(foo)&within(loops, *);
  pWithinAllAbc : get(x)&within(*, abc);
end
```

Patterns - Type Restriction

The isclass pattern and dimension pattern can be used to restrict the matlab class and dimension of matches to array accesses and assignments. The isclass pattern matches all array assignment and array access join points which operate on data of the specified matlab type, and the dimension pattern matches those which have the specified size.

```
patterns
   isint32pat  :  isclass (int32)  ;
   isbsinglepat  :  (get(x)| set (x))& isclass (numeric)  ;
  dimp : dimension (2,*,*)  ;
  dimx2by2 : (get(x)| set (x))&dimension(2,2);
end
```

Actions

- Actions are contained inside blocks, a particular aspect can have any number of blocks of actions.
- Actions are pieces of code that can be executed at certain points in source code when matched by specified patterns.
- Actions are named, and are linked to an existing pattern in the patterns block.
- When more than one action of the same class is triggered by the same join point, the actions are applied in the order they are defined

Actions

There exist 3 types of actions, before, around and after actions. Before actions are executed before a matched pattern, after actions are matched after a matched pattern. Around actions are a bit more complicated and are executed around a matched pattern.

```
                Where code should be executed with respect to matched pattern
actions                  /         __which pattern to match
       aCountCall : before pCallFoo
               this.count = this.count + 1; --Code to be executed
       end

       aExecution : after executionMain
               total = this.getCount();
               disp(['total calls: ', num2str(total)]);
       end
end
```

Actions - Context Exposure

- It is often important to have some information about a matched join point to be used in action code. This is done using context exposure.
- In AspectMatlab, context exposure is done by specifying selectors along with an action definition.

Selectors to be used for the action

```
actcall : before call2args : (name, args)

      disp(['calling ', name, ' with arguments(', args , ')']);

end
```

Actions - Context Exposure

The selectors that are applicable depend on the join point type. Certain selectors will have different meanings depending on the join point on which they are used.

	set	get	call	execution	annote	loop	loopbody	loophead
args	indices		arguments passed		arguments	loop iteration space		
obj	old value	value	handle	-	-	-	iterator variable	-
newval	new value	-	-	-	-	-	-	loop range
counter	-	-	-	-	-	-	current iteration	-
name	name of entity matched					-	-	-
pat	name of pattern matched							
line	line in the source code							
loc	enclosing function/script name							
file	enclosing file name							
ainput	-	input var name(s)			-	-	-	-
aoutput	-	-	-	output name(s)	-	-	-	-

Around Actions

Around actions are executed in the place of the join point they match. A special call, proceed, carries out the join point matched by the pattern instead of it simply being executed before or after the action, as with before or after advice. The proceed function can be used several times, including not at all.

```
actions
  actcall : around call2args : (name, args)
    disp (['before call of ', name, 'with parameters(', args , ')']);
    proceed();
    disp (['after call of ', name, 'with parameters(', args , ')']);
  end
end
```

Around Actions

A special variable, varargout, is used to return arguments. When proceed is used, returning arguments is dealt with automatically. In the event that proceed is not used varargout should be set to contain as many values as the original join point would return

```
actions
   actcall : around callFoo : (args)
     % proceed not called, so varargout is set
     varargout{1} = bar(args{1}, args{2});
  end
end
```

Back to the Example

```
aspect myAspect
 properties
   count = 0;
end
methods
   function incCount(this)
      this.count = this.count + 1;
   end
end
patterns
   callFoo : call(foo);
end
actions
   actCall : before callFoo
   this.incCount();
end
end
```

Another Example

```
aspect myAspect
 properties
   count=0;
end
methods
   function out = getCount(this)
     out = this.count;
   end
   function incCount(this)
     this.count = this.count + 1;
   end
end
patterns
   call2args : call (*(*,..));
   executionMain : mainexecution();
end
```

Another Example

```
actions
  actcall : around call2args : (name, args)
    this.incCount();
    disp([' calling ', name, 'with parameters(', args , ')']);
    proceed();
  end
  actexecution : after executionMain
    total = this.getCount();
    disp([' total calls : ', num2str(total)]);
  end
end

end
```

AspectMatlab Compiler - How to Use

After obtaining the AspectMatlab jar file, you can compile aspects by executing the jar,with a list of Matlab files and the AspectMatlab files that should be woven into them.

```
java −jar amc.jar myFunc.m myAspect.m

java −jar amc.jar −main myFunc.m myAspect.m
```

Code generated by the AspectMatlab compiler can be found in a "weaved" directory, which is created in the current working directory. This code can then be executed by running the woven Matlab file in any Matlab environment.

End