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

```java
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:
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.
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.
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.
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
An aspect is named and contains a body. Aspects are designed to be similar to object-oriented MATLAB classes, which allow for properties 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.
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.

```matlab
patterns
    callFoo : call(foo);
end
```

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

```
set
join point
arr1 = [1, 2, 3];
arr2 = arr1(1);
set
join point
arr1 = [1, 2, 3];
arr2 = arr1(x, y);
```

```
get
join point
get
join point
arr1 = [1, 2, 3];
get
join point
arr1 = [1, 2, 3];
```

```
get
join point
get
join point
arr1 = [1, 2, 3];
```

```
get
join point
get
join point
arr1 = [1, 2, 3];
```

```
get
join point
get
join point
arr1 = [1, 2, 3];
```
- 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.

```matlab
function myMain()
    ...
    foo();
    call join point
    ...
end
```
Specialization of patterns for arrays and function calls can be based on the indices/arguments used for assignments/function calls.

- \texttt{call(foo)} matches all calls to \texttt{foo} (function or script).
- \texttt{call(foo())} matches calls with no arguments (function or script).
- \texttt{call(foo(*))} matches calls with exactly one argument (function only).
- \texttt{call(foo(..))} matches calls with 1 or more argument(s) (function only).
- \texttt{call(foo(*,..))} matches calls with 2 or more arguments (function only).

...and so on.

- \texttt{set(arr)} matches all assignments to \texttt{arr}.
- \texttt{set(arr())} matches assignments with no indices.
- \texttt{set(arr(*))} matches assignments with exactly one index.
- \texttt{set(arr(..))} matches assignments with 1 or more index/indices.
- \texttt{set(arr(*,..))} matches assignments with 2 or more indices.

...and so on.
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.

```matlab
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]
...
```
The `annotate` 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
  annotateEx: annotate(plot);
  annotateAdd: annotate(add(double,double));
  typeAnn : annotate(type(var, char, [\*]));
end
```
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
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);
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.
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.

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

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

```matlab
actcall : before call2args : (name, args)
    disp(['calling ', name, ' with arguments(', args, ')']);
end
```
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.

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

```plaintext
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.

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