= Aspect weaver =

An aspect weaver is a metaprogramming utility for aspect @-@ oriented languages designed to take instructions specified by aspects (isolated representations of a significant concepts in a program) and generate the final implementation code . The weaver integrates aspects into the locations specified by the software as a pre @-@ compilation step . By merging aspects and classes (representations of the structure of entities in the program) , the weaver generates a woven class .

Aspect weavers take instructions known as advice specified through the use of pointcuts and join points , special segments of code that indicate what methods should be handled by aspect code . The implementation of the aspect then specifies whether the related code should be added before , after , or throughout the related methods . By doing this , aspect weavers improve modularity , keeping code in one place that would otherwise have been interspersed throughout various , unrelated classes .

= = Motivation = =

Many programming languages are already widely accepted and understood . However , the desire to create radically different programming languages to support the aspect @-@ oriented programming paradigm is not significant due to business @-@ related concerns ; there are risks associated with adopting new technologies . Use of an entirely new language relies on a business 's ability to acquire new developers . Additionally , the existing code base of a business would need to be discarded . Finally , a business would need to acquire a new toolchain (suite of tools) for development , which is often both an expense in both money and time . Primary concerns about roadmaps for the adoption of new technologies tend to be the need to train new developers and adapt existing processes to the new technology .

To address these business concerns , an aspect weaver enables the use of widely adopted languages like Java with aspect @-@ oriented programming through minor adaptations such as AspectJ which work with existing tools . Instead of developing an entirely new language , the aspect weaver interprets the extensions defined by AspectJ and builds " woven " Java code which can then be used by any existing Java compiler . This ensures that any existing object oriented code will still be valid aspect @-@ oriented code and that development will feel like a natural extension of the object @-@ oriented language . The AspectC + + programming language extends C + + through the use of an aspect weaver , offering the additional efficiency over AspectJ that is necessary for embedded systems while still retaining the benefits of aspect @-@ oriented programming .

= = Implementation = =

Aspect weavers operate by taking instructions specified by aspects , known as advice , and distributing it throughout the various classes in the program automatically . The result of the weaving process is a set of classes with the same names as the original classes but with additional code injected into the classes ' functions automatically . The advice specifies the exact location and functionality of the injected code .

Through this weaving process , aspect weavers allow for code which would have otherwise been duplicated across classes . By eliminating this duplication , aspect weavers promote modularity of cross @-@ cutting concerns . Aspects define the implementation code which would have otherwise been duplicated and then use pointcuts and join points to define the advice . During weaving , the aspect weaver uses the pointcuts and join points , known as a pointcut designator , to identify the positions in candidate classes at which the implementation should be injected . The implementation is then injected into the classes at the points identified , thus permitting the code to be executed at the appropriate times without relying on manual duplication by the programmer .

= = = Weaving in AspectJ = = =

In the programming language AspectJ , pointcuts , join points , and the modularized code are defined in an aspect block similar to that of Java classes . Classes are defined using Java syntax . The weaving process consists of executing the aspect advice to produce only a set of generated classes that have the aspect implementation code woven into it .

The example at right shows a potential implementation of an aspect which logs the entry and exit of all methods . Without an aspect weaver , this feature would necessitate duplication of code in the class for every method . Instead , the entry and exit code is defined solely within the aspect .

The aspect weaver analyzes the advice specified by the pointcut in the aspect and uses that advice to distribute the implementation code into the defined class . The code differs slightly in each method due to slight variances in requirements for the method (as the method identifier has changed) . The aspect weaver determines the appropriate code to generate in each situation as defined by the implementation advice and then injects it into methods matching the specified pointcut .

= = = Weaving to bytecode = = =

Instead of generating a set of woven source code , some AspectJ weavers instead weave the aspects and classes together directly into bytecode , acting both as the aspect weaver and compiler . While it is expected that the performance of aspect weavers which also perform the compilation process will require more computation time due to the weaving process involved . However , the bytecode weaving process produces more efficient runtime code than would usually be achieved through compiled woven source .

= = = Run @-@ time weaving = = =

Developments in AspectJ have revealed the potential to incorporate just @-@ in @-@ time compilation into the execution of aspect @-@ oriented code to address performance demands . At run @-@ time , an aspect weaver could translate aspects in a more efficient manner than traditional , static weaving approaches . Using AspectJ on a Java Virtual Machine , dynamic weaving of aspects at run @-@ time has been shown to improve code performance by 26 % . While some implementations of just @-@ in @-@ time virtual machines implement this capability through a new virtual machine , some implementations can be designed to use features that already exist in current virtual machines . The requirement of a new virtual machine is contrary to one of the original design goals of AspectJ .

To accomplish just @-@ in @-@ time weaving , a change to the virtual machine that executes the compiled bytecode is necessary . A proposed solution for AspectJ uses a layered approach which builds upon the existing Java Virtual Machine to add support for join point management and callbacks to a Dynamic Aspect @-@ Oriented Programming Engine . An alternative implementation uses a weaving engine that uses breakpoints to halt execution at the pointcut , select an appropriate method , embed it into the application , and continue . The use of breakpoints in this manner has been shown to reduce performance due to a very large number of context switches .

= = Performance = =

Aspect weavers ' performance , as well as the performance of the code that they produce , has been a subject of analysis . It is preferable that the improvement in modularity supplied by aspect weaving does not impact run @-@ time performance . Aspect weavers are able to perform aspect @-@ specific optimizations . While traditional optimizations such as the elimination of unused special variables from aspect code can be done at compile @-@ time , some optimizations can only be performed by the aspect weaver . For example , AspectJ contains two similar but distinct keywords , thisJoinPoint , which contains information about this particular instance of woven code , and thisJoinPointStaticPart , which contains information common to all instances of code relevant to

that set of advice . The optimization of replacing thisJoinPoint with the more efficient and static keyword thisJoinPointStaticPart can only be done by the aspect weaver . By performing this replacement , the woven program avoids the creation of a join point object on every execution . Studies have shown that the unnecessary creation of join point objects in AspectJ can lead to a performance overhead of 5 % at run @-@ time , while performance degradation is only approximately 1 % when this object is not created .

Compile @-@ time performance is generally worse in aspect weavers than their traditional compiler counterparts due to the additional work necessary for locating methods which match the specified pointcuts . A study done showed that the AspectJ compiler ajc is about 34 % slower than the Sun Microsystems Java 1 @.@ 3 compiler and about 62 % slower than the Java 1 @.@ 4 compiler .