Abstract

The ability for software products to adapt to future changes is essential. Whether the change, the delta, is prefixed to subclasses or postfixed to super classes, class hierarchies grow vertically, upward or downward. The horizontal expansions (infix changes) of these hierarchies have been restricted or neglected. The infix changes are better understood as crosscutting concerns. However, these concerns shall not be confused with mixins in OO technology. Although OO and AOP technologies have addressed the issue of the incremental addition to software products, the mix-and-match of software features often results in a redundant code and class hierarchies that are difficult to integrate and maintain. In this paper we present an approach that combines the advantages in both AspectJ for its join semantics and HyperJ for its merge and provides an XML-based weaving rules to incorporate the delta changes into the core class hierarchies both dynamically and statically.

Keywords: Aspect-Oriented, Weaving Rules, Join Points, functionality slice, Software Adaptation.

1. Introduction

One of the main rational behind object-oriented programming is to support the notion of programming by extension. However, current object-oriented languages do not support the notion of on-demand composable software products. Software products often consist of a set of core functionalities and a set of optional features that can be added or removed based on customer requests. Software features are functionality slices that can be added to the base class hierarchies statically or dynamically. Current object-oriented technologies depend on inheritance and composition mechanisms which do not scale well when there are a large number of optional features that need to be cooperated into the core functionalities of the software products. Software change requests may result in new implementation in a number of classes. If the number is small, we can use either prefix, i.e., parameterized types, or postfix, i.e., inheritance, to add the changes to the core functionality class hierarchies. In general, postfix is implemented through inheritance while prefix is implemented through the use of mixins [20, 21, 22, 23]. However, if the implementation of a software change request results in a scattered implementation across the class hierarchies, such implementation will be difficult to integrate and expensive to maintain without the use of aspect-oriented techniques [6,7,8,9,10,18,19,27,28]. Aspect-oriented technologies support the infix changes by weaving the infix changes’ class into the base class hierarchy in a non-invasive but crosscutting manner; invasive changes will affect the base class implementation and consequently change the initial type of the base class for all heir classes. Adding a functionality slice, a feature, to the software product statically shall be possible without the unnecessary and invasive rework on base classes and without the need to shutdown the running system for reconfiguration. However, the ability for a software product to adapt in order to meet change requests either statically or dynamically relies heavily on underlying supporting technologies. A change request, whether it is an addition or removal of a functionality slice, may require more than a simple inheritance mechanism in order to append the changes to the base class. In fact, as we will see later, inheritance can be treated as a special kind of merge with the override semantics. Unfortunately, to our knowledge, none of the current object-oriented languages support the general notion of class merges and large scale integration of class hierarchies based on external configuration rules specified separately from base class hierarchies.

To better understand the shortcomings of these technologies, consider the possible features that a shape class might have: the ability to draw a shape, change a color, change a style, or change a dashed pattern, or weight. By encapsulating each one of these features in its own class, we could build a shape class by creating a class hierarchy that contains a base class and selected feature classes in linear order. The result would be a fixed class hierarchy that supports the selected shape features. However, different feature combinations would require different class hierarchies. In some cases these new hierarchies would require
existing feature classes to have different superclasses, which would lead to a replication of code that quickly becomes unmanageable as the number of new feature combinations increases. Multiple inheritances would not help because what is needed is a careful weaving of different classes that have different supertypes.

Mixins can be used to solve the code repetition problem. However, there are some features that may crosscut a component(s) vertically or horizontally. One may argue that vertical crosscutting can be easily avoided through inheritance. However feature combinations may limit the scalability of this approach and produce a huge number of class hierarchies in order to accommodate the different feature combinations. Similarly, the combinations of horizontal crosscutting may produce monolithic class hierarchies that would severely impact the maintenance and evolution of the software product. Perhaps the modular representation of crosscutting concerns may subdue the code tangling problem. However, dynamic software evolution and reconfigurability are important as well since rebuild and reboot are prohibited in some situations.

Software products are expected to evolve and run in a wide range of platforms and environments. For software products to fit tomorrow’s requirements and run effectively across a wide range of platforms and environments, it requires a new approach to reconfigure and adapt these products in a non-invasive manner. For instance, as more and more inexpensive, small and light-weight computer devices become available, it is desired that what was formerly thought of as in-office computer related activities be carried out anywhere and anytime. The challenge is how we can adapt traditionally distributed applications to a mobile environment without invasively re-writing the existing applications. Consider the multi-site video game playing application. In order to play multi-site video games under a mobile environment the computer resource, network resource and the location of the application itself are constantly changing, the application must be able to adapt to such changes. In this scenario, dynamically reconfiguring its QoS requirements based on the current resource context becomes necessary [31].

It is clearly desirable that these changes take place in a pervasive but non-invasive manner. Software design methodologies have gone through consistent and rigorous improvements for the past few decades. Object-oriented technology [5] has evolved based on the information hiding principle, and aspect-oriented technology [6, 8, 9, 10, 11] has evolved from the object-oriented technology based on the principle of modular representation of crosscutting concerns. Whether the software design is based on flowcharts, modules, objects, or aspects, the goal is always to achieve better design to fit tomorrow’s requirements, to achieve a software system that is easy to evolve with change requests, maintain, and comprehend. In [15, 16], it has well been argued that considering the reason for change will always yield to a better design.

In next section we articulate the essence of software changes and the existing approaches that tackled this issue. We also present a better categorization scheme for software changes and how aspect-oriented techniques can better complement previous approaches in building software products that can be easily extended or contracted. In section 3, we discuss the technical details of our approach. Section 4 presents the related work. In the final section, we present the concluding remarks and our future work.

2. Software Changes

There are a number of approaches [1, 3, 20, 21, 22, 23, 34, 35] that can be used in order to build software products which can accommodate changes. A majority of these approaches support the postfix and prefix changes. Current practices for software engineers tend to focus on software design that meets current requirements, and then revise the design to engineer proper hooks in order to accommodate future requirements. Indeed, this approach may work for a few releases of software products but afterwards the software products will become a monster of class hierarchies that are difficult to change and expensive to maintain. The essence of this problem stems from the fact that the engineers attempt to use common techniques that are adequate to accommodate only few changes but inefficient for streams of changes. As we will articulate later in the paper, reflection is a powerful capability if it is used as a first class entity in the programming languages.

Software adaptation is one of the fundamental design requirements for modern software systems. The ability to adapt in order to meet future change requests is a necessary condition in order to prolong the lifespan of a software product. Software adaptation can be categorized as either static (adaptable) or dynamic (adaptive). An adaptable software system is one where software components are selected, customized, and only executed after compilation. In contrast, adaptive software systems can be altered while the system is running, without the need to recompile and reboot the running system. Changes to the software system are in
the form of adding, removing, or replacing software components or services. Static adaptability can be engineered in order to support the structural and behavioral changes. Existing OO programming languages and design patterns are sufficient to meet these needs, though recent aspect-oriented approaches have significantly improved and simplified this process. However, dynamic adaptability to support structural changes would require reflection support from the underlying programming language model [24, 26].

Software changes may come in many forms and could be originated from different sources. A software change request may impact the behavioral or structural properties of the software product. Hence, software reconfigurability is important for classes of applications where runtime environment changes may bring the application into a panic mode. Decomposing the software product into its perspective components is often a challenging task since the requirements in the specification document are not segregated into their perspective orthogonal components. Separation of concerns is essential to separate the core functionality components from the add-on concerns like security, logging, synchronization, and distribution transparency; some of these concerns can be isolated in one component while others may touch a number of different components. The problem will be overwhelming when considering feature-driven development, where numerable software features are added to software products in different releases.

2.1 Class Hierarchy Extensions and Expansions

Whether we are extending the class hierarchies vertically or expanding them horizontally, additive software changes (Δ) can be classified as postfix, prefix, or infix, Figure 1; our definition of the delta software change (Δ) has similar semantics to the one presented in [20]. The traditional inheritance mechanism is an example of postfix changes in which changes are appended to the base class. Mixin is an example that supports the prefix changes. Current aspect-oriented technology [6,7,8,9,10,18,19,27,28 ] is an example that provides limited support for infix changes and modular representation for changes that crosscut a number of classes; the infix changes can be better implemented if the programming model utilizes the join and merge weaving mechanisms along with the weaving rules that are separately specified in XML-based configuration file. In the following sections we describe the essence of these changes.

2.1.1 Postfix Changes

Object-oriented technology relies primarily on inheritance in order to support programming by extension, where new changes are added to the newly created class and previous code is inherited from the base class. The inheritance mechanism in object-oriented languages is a special kind of merge where subclass methods override the same method definitions in the superclass. Although merging two classes with the same name through the inheritance mechanism in current OO languages may produce a syntax error and semantically may look like a recursive definition, in subject-oriented programming [25] it is possible to merge two classes with the same name in order to produce the composed functionality under the same class name. In object-oriented programming, once a class hierarchy is constructed, changes to superclass(es) definitions in the hierarchy are prohibited since it is considered as invasive and may impact the heir class(es) or the client classes.

2.1.2 Prefix Changes

To better articulate the essence of the prefix changes, such as mixins, we present a class hierarchy for the publisher application. The publisher maintains a list of authors who belong to the industry track or the academic track. Every author may have published a book, or a conference paper, or a journal paper, or a magazine paper, but not every author has written a book and a journal paper and a conference paper and a
magazine paper. However, it is possible that some authors have written magazine papers and published a number of books, while others have published a number of conference and journal papers. The mix-and-match of these features may produce redundant code and giant class hierarchies that are inflexible for extension and expensive to maintain. These are mixin [20, 21, 22, 23] type of compositions and considered as upward extensions or prefix. Analogous to inheritance by which the extension is downward or postfix, however, there is another class of changes that require changes to expand horizontally across the class hierarchy in a none-invasive manner. We call such changes *infix* changes. AOP [6,7,8,9,10,18,19,27,28] is an attempt in that direction.

Most mixin based approaches [20, 21, 23, 24] are based on linguistic extensions to common OO languages; C++ supports mixins by means of templates, Java (JDK1.5) supports the notion of generic types, and C# is considering this feature in future releases.Mixin approaches have one phase (growing phase), by which a mixin can be passed to another mixin. However if there is a need to remove a certain mixin from the mixin hierarchy at run time, current approaches fall short of meeting such a requirement.

### 2.1.3 Infix Changes

Changes to an existing class, by means of adding, removing, or modifying, the definition of methods or fields without subclassing is considered invasive and may have rampant side effects on the heir classes or the client classes. There is certain kind of concerns that may cut across a class hierarchy and often causes maintenance changes to be invasive. Aspect-oriented technology is an attempt to resolve this issue by modular representation of the crosscutting that is called an aspect by which maintenance is localized and injected to the base class hierarchy in a non-invasive manner. For example, adding the log capability before a client can invoke the deposit method on the checking account class, without aspect-orientation, may require changing the base class and the client class. Aspect-oriented technologies provide programmers with the capability to weave the crosscutting concerns with their corresponding classes in an explicit way that will protect the base class and client class from such changes. However, aspect-oriented technology is better utilized if we allow the weaving rules to be specified in isolation of aspects and their corresponding classes. This approach is taken by HyperJ [27] as well.

### 2.2 Weaving Rules

As illustrated in previous sections change requests and integration may come in different forms. We may add a functionality slice to a base class at build time through subclassing or runtime through the use of the decorator design pattern. However, merging two classes with the same class name to produce a unified class is not supported in current object-oriented languages. Parameterized types are possible through the use of mixins [20, 21, 22, 23] and a combination of mixins as well, but once the mixin class hierarchy is constructed, a mixin can not be removed. While Java has recently introduced the notion of generic types in its 1.5 release, currently C# does not support mixins by means of templates like C++, however, Java and C# have sufficient support for reflection by which developers can exploit in order to intercept method calls and exceptions and reroute the method call to another object. We will elaborate on this capability in the next section. Crosscutting concerns can be modularized by means of aspects and at a later point weaved with their corresponding classes through the use of aspect-orientation.

Our approach as shown in Figure 2 presents a solution by which components can be postfixed, prefixed, or infixed with changes, through the use of an XML schema that states the join and merge rules for the respective components either statically or dynamically. The merge in our approach has similar semantics to mixins, where *merge* operation may generate new class type while the *join* maintains the initial type.

### 2.3 Merge verse Join
It is important to emphasize the difference between the merge and the join. The merge may produce a new type, class, while join may never produce a new type, class. Table 1 shows the relationship between the weaving type and the weaving result:

<table>
<thead>
<tr>
<th>Type Retention</th>
<th>New Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join</td>
<td>✓</td>
</tr>
<tr>
<td>Merge</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Table 1: Weaving Mechanisms’ Results**

A merge that uses the named type merge, class, can be used to build a new named type, class, merge or unnamed type merge, for example, merge(C5,C4, merge(C3,C1,C2)); where C5 and C3 are the new named type, classes, merge. Table 2 shows the relationship between the weaving type and the weaving unit.

<table>
<thead>
<tr>
<th>Join</th>
<th>Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>✓</td>
</tr>
<tr>
<td>Interface</td>
<td>✓</td>
</tr>
<tr>
<td>Construct</td>
<td>✓</td>
</tr>
<tr>
<td>Method</td>
<td>✓</td>
</tr>
<tr>
<td>Field</td>
<td>✓</td>
</tr>
<tr>
<td>Access Specifier</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Table 2: Weaving Mechanisms’ Unit**

For instance, given two classes C1 and C2 with C1 having methods m1 and m2 and C2 having methods m3 and m4, then the result of the merge(C1,C2) will be as a new type, C3, which has the union of C1’s and C2’s methods, the set \{m1, m2, m3, m4\}. However, if C2 has methods m1 and m3, then there will be a name collision and the result of the merge will be ambiguous. Therefore, the weaving rules shall specify how to resolve this collision. There are four cases to consider: override, rename, exclude and concatenate. The override will replace m1 method of C1 with m1 of C2. The semantics of the override is that the latter class methods will override the methods of the predecessor. The rename will rename m1 of C1 to C1m1 and m2 of C2 to C2m2, only if these methods are declared as private members in C1 and C2. This kind of merge will not preserve C1 and C2 types in the resulting type C3. Exclude will remove the method with name collisions. In our example, m1 will be removed from the merge assuming that neither m2 nor m3 makes an explicit call to m1. Similar to rename, exclude changes the definition of source types and generates a new type C3. If the intent of the merge is to preserve the identity of either C1 or C2, concatenation is the only one type will result from this kind of merge. Concatenate sequentially appends the common methods together (in our example, the C1.m1 and C2.m1) without changing the class definition of C1.

Table 3 summarizes the different name collision resolutions in the merge.

<table>
<thead>
<tr>
<th>Override m1 =</th>
<th>Rename m1 =</th>
<th>Exclude</th>
<th>Concat m1 =</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1 C2.m1</td>
<td>C1m1</td>
<td>✗</td>
<td>C1.m1@C2.m1</td>
</tr>
<tr>
<td>C2 ✗</td>
<td>C2m2</td>
<td>✗</td>
<td>C2.m1@C1.m1</td>
</tr>
<tr>
<td>C3 C2.m1      ✗</td>
<td>✗</td>
<td>✗</td>
<td>C1.m1@C2.m2</td>
</tr>
</tbody>
</table>

**Table 3: Name Collision Resolutions in Merge**

To join C1 with C2, the weaving rules shall specify the joint points. Furthermore, in our framework, classes are the basic building blocks (interfaces are considered as special classes with no implementations and functions/methods are treated as classes with no states). In order to maintain the encapsulation of objects, join points are limited at the method level, and advices are given as methods with enclosing class contexts.

The syntax for the new extensions with join and merge is as follows:

\[
O := \text{Methods} \\
C := C | J(O[C], C[O], O[C]) | M(C, C, C) \\
J := \text{before} | \text{after} | \text{replace} | \text{around} \\
M := \text{override} | \text{rename} | \text{exclude} | \text{concert}
\]

Different from merge semantics, \texttt{join} always maintains the identity of its core class. The first and last elements of a \texttt{join} operation specify advice while the second of the element of the operation defines the join point. Examples of \texttt{J(O[C], C[O], O[C])} are:

1. \texttt{before (m1[C1], C[m1], )}
2. \texttt{after (m1[C1], C[m1], )}
3. \texttt{replace (m1[C1], C[m1], )}
4. \texttt{around (m1[C1], C[m1], m2[C2])}

The first expression states that method m1 in the context of class C1 will be executed "before" the method \texttt{m} of class C is called; the second example adds the advice \texttt{m1} in class C1 at the end of \texttt{m} of class C. The \texttt{around} weaving is a combination of \texttt{before} and \texttt{after}. "around (m1[C1], C[m1], m2[C2])" causes method \texttt{m1} in C1 to be executed as soon as a thread of control enters method \texttt{m} of class C and \texttt{m2} in class \texttt{C2} be executed right before the \texttt{m} exits. \texttt{Replace}, on the other hand, changes the implementation of the core in such a way that the core implementation is replaced by an advice while its interface/signature is still maintained. Hence, the example given above in (3) denotes that when a thread of control reaches the method \texttt{m} of class C, the advice \texttt{m1} in class C1 is executed and the thread exits from method \texttt{m} right after the execution of method \texttt{m1}. Since the join does not modify method signatures or state types, it does not change the
definition of class C. Hence, no new types are needed for join operations.

Recent approaches [10, 28] for aspect-oriented technology have inter-mixed the weaving rules with aspect functionality. HyperJ is an exception. However, with HyperJ dynamic weaving is not possible. Our previous work [7] has demonstrated the effectiveness of using aspect-oriented technology in building concurrent object-oriented systems, where the concurrency aspects, like synchronization constraints and scheduling policies are isolated from the core functional components. This approach helps to build a stable software system that can easily adapt to meet future requirements. However that work fell short of supporting weaving rules that can be used at build time or runtime in order to build software products that can be easily extended or contracted. To facilitate engineering extension and contraction into the software product, we present a framework using .NET's reflection that simulates the dynamic proxy capability in java.lang.reflect.Proxy class, part of the Java language reflection package, along with the XML weaving rules in order to allow the component developers to extend an existing class by adding a new functionality slice or inject a new functionality slice into an existing class in a non-invasive manner.

3. Implementation

In this section, we discuss the architecture and implementation of our framework and discuss only the most relevant details due to space limitations; the complete schema definition can be downloaded from http://www.cs.iit.edu/~ren/research/weaver-schema. As discussed earlier, the join is done dynamically and does not produce a new type, while the merge is done statically and produces new types. In both cases, we utilize several namespaces in the .NET Framework. The most important ones are System.Reflection, System.CodeDom.Compiler, System.Xml.Schema, and System.Text.RegularExpressions. In addition, an XML configuration file is supplied by the user (programmer) to specify the weaving rules, which is validated by our weaver against an XML schema.

3.1 Join

The architecture of the framework that implements both joins and merges is depicted in Figure 3. A join may be specified programmatically by calling the static function

```
AspectWeaver.AspectRepository.AddAspect() and passing four parameters: aspect name, pointcut name, pattern string, and an instance of a class that implements the IAdvice interface. This function may be called several times for each join, and it may be called from within any scope and at any point in the execution of the client program. This gives the client the flexibility to add / remove advice dynamically and conditionally. Here is an example that adds all functions, which satisfy the following criteria (1) have name Write, (2) take any number of arguments and (3) belong to any namespace, to pointcut “Writing” within aspect “Email”:

```
AspectWeaver.AspectRepository.AddAspect("Email ", "Writing", @".*\.*Write\(.\)\", new EmailAdvice());
```

An alternative approach to perform a join is to initialize the aspect repository from a programmer specified XML configuration file using static function AspectWeaver.AspectRepository.InitializeFromXml() and passing a single parameter, the XML file name is of string. Type. InitializeFromXml() will in turn call AspectWeaver.AspectRepository.AddAspect() for every rule specified in the XML file. This means that a combination of the two methods is legal and gives a client the flexibility to add / remove advice. Also, the same advice may be reused in several XML join rules with different patterns. Here is an example of an XML fragment that specifies two join rules:

```
<aspect name="WorkFlow">
  <pointcut name="SaveAndLoad">
    <advice>
      <join><after/></join>
      <pattern>
        SalesOrder\..Save(.)
      </pattern>
      <type>
        Client.Send
      </type>
    </advice>
    <advice>
      <join><before/></join>
      <pattern>
        SalesOrder\..Load(.)
      </pattern>
      <type>
        Client.CreateTask
      </type>
    </advice>
  </pointcut>
</aspect>
```
3.1.1 Join Types

The type of join is determined by the actual type passed by the client, as depicted by Figure 3. We have three interfaces supplied by our framework that implement IAdvice: before, after, and replace. The around join can be accomplished by implementing both interfaces before and after.

All three sub-interfaces require a method called Invoke which gets called by the framework appropriately based on the interface being implemented. Here is an example of a class that implements around:

```csharp
public class EmailAdvice : IBeforeAdvice, IAfterAdvice
{
    public string AspectName
    {
        get { return "Email"; }
    }

    void IAfterAdvice.Invoke(object theObject, string theMethodName, object[] theArgs, ref object theResult)
    {
        MessageBox.Show("Email: AfterAdvice.");
    }

    bool IBeforeAdvice.Invoke(object theObject, string theMethodName, object[] theArgs)
    {
        MessageBox.Show("Email: BeforeAdvice.");
        return true;
    }
}
```

3.1.2 Aspect Repository

The AspectRepository class is used by the framework to keep track of all join types. It maintains a collection of Aspect. When AspectRepository.AddAspect() is called, the call eventually gets routed to the pointcut’s Pointcut.AddAdvice(string thePattern, IAdvice theAdvice) function. Class Pointcut maintains three hashtables, one for each type of advice. An instance of System.Text.RegularExpressions.RegEx(thePattern) is the key, and the value is theAdvice. Now, when the Pointcut.BeforeAdvice() function, for example, is called by the framework (discussed in the next subsection), it performs a match of the pattern against the method signature of the method being invoked and if a match is found the advice in question is invoked. The match is tested by calling the function RegEx.IsMatch().

3.1.3 The Interceptor

A client creates an instance of a class by calling the static method AspectWeaver.NewInstance(Type theType, object[] theArgs). AspectWeaver maintains a dictionary of woven types, the first time an instance of a type is requested, the weave creates an in-memory dynamic type derived from the requested type, overrides all virtual methods, inserts calls to the AspectRepository.BeforeAdvice(), base class version of the function, and AspectRepository.afterAdvice(). The System.Reflection namespace is utilized to gather...
information about theType, and the System.CodeDom.Compiler namespace is used to in-memory compile the woven type. Finally, NewInstance() returns an instance of the in-memory sub-class, which acts as an interpreter. This was required to simulate java's dynamic proxy capability found in java.lang.reflect.Proxy.

3.2 Merge

A merge is accomplished statically by using our weave command line tool. Weave is a Microsoft Windows console application that takes one single command line argument, i.e., the XML configuration file name that contains the weaving rules. Here is an example: weave rules.xml.

The architecture of weave is depicted in Figure 3. When invoked on an XML file, weave first validates the rules against our XML Schema; we use a System.Xml.XmlValidatinReader instance to perform the validation. If the rules are valid syntactically and semantically, then weave proceeds to generate the final .NET assemblies as specified by the weaving rules. We use a Microsoft.CSharp.CSharpCodeProvider to produce an instance of System.CodeDom.Compiler.ICodeCompiler, which in turn we use to compile the C# code generated by our code unit. The code unit is simply a document object model that represents our merged classes.

4. Related Work

Software change requests may impact a single concern or a cluster of orthogonal concerns of a software product. The approach presented in this paper can be used to build software products that can adapt to changes statically or dynamically. Such changes are based on two different types of weaving types, i.e., join and merge, which are specified by the XML schema file. There are a number of approaches that have addressed software changes. The first class of these approaches has focused on the adaptability aspect of the software system. These approaches [8, 9, 10, 11, 16] have focused on the adequacy of current programming languages, software frameworks, and software practices in building adaptable software systems. The second class of approaches [1, 2, 3, 4] has focused mainly on software adaptiveness to build reconfigurable software systems.

The aspect-oriented paradigm provides a new mechanism beyond functions and inheritance for isolating the expression of crosscutting concerns that simplifies the extension and contraction of the software systems [6]. The Composition Filters [9] approach intercepts messages exchanged between objects for expressing coordination behavior between objects. The Demeter approach [11] focuses on the extraction and expression of object graph traversals in the implementation of concerns that crosscut a number of object hierarchies. The AspectJ approach [10] extends the Java language with constructs to express aspects as first class entities and provide mechanisms to weave aspects with classes. Multi-Dimensional Separation of Concerns approach [8] has focused on the identification of different concerns that an object must address and then compose classes from a set of semi complete behaviors called hyperslices. A recent stream of AOP approaches, JAC[28] and PROSE[30] have focused on the adequacy of dynamic weaving and providing the linguistics support to build adaptive software systems. Although these AOP technologies simplify the design of the adaptable/adaptive software systems, unfortunately, these AOP technologies have focused mainly on providing linguistic constructs along with their compositional models to ease intermixing aspects and components; however, in these technologies the components are not separated from their weaving rules. JBoss [29] and AspectWerkz [34] are other examples of these approaches that support the dynamic weaving of aspects based on XML specification rules, however, these approaches don’t provide a unified model that supports both the join and merge weaving mechanisms.

Software adaptation can be classified as either static (adaptability) or dynamic (adaptiveness). Software adaptability is the ability of the software product to meet future requirements in a flexible and inexpensive manner. On the hand, software adaptiveness is about enabling the software system to alter its behavior and reconfigure itself at runtime. There are a number of techniques [11, 14, 18, 24, 25] that have been devised in order to aid the engineer in designing software systems that are resilient to changes, whether the changes are static or dynamic in nature. But none of these techniques have provided the necessary mechanisms that simultaneously support the adaptability and adaptiveness of the software system in a non-invasive manner.

5. Conclusion

Software changes are inevitable and current object-oriented and aspect-oriented programming techniques are necessary but not sufficient for building software products that have to cope with rampant of change
requests. These changes come in many forms, postfix, prefix, and infix. Current programming languages provide different mechanisms to accommodate these changes, however, the end results are often class hierarchies that are difficult to extend and expensive to maintain. These languages provide the developer with powerful constructs to extend the class hierarchies. However decoupling these hierarchies or contracting them is difficult.

In this paper we have presented a technique that articulates the essence of the delta changes for software products and showed how to construct these hierarchies for ease of extension and contraction. Reflection is an essential component of most modern programming languages but often neglected by software engineers. Reflection can be used to build software products that can adapt to change requests statically or dynamically. Separation of concerns is essential for a healthy design but separating the components from the weaving rules is indispensable for building software products that are adaptable and adaptive. Every generation of programming languages attempts to raise the level of abstraction for programmers in order to improve the quality and quantity of the software produced. Joining and merging software components can be better abstracted in XML based configuration files if we are willing to accept that joining and merging components as important as inheritance that we use in every program that we write.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments and feedback.

References

16. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-
Oriented Software. Addison-Wesley, Reading, MA, 1995


32. Microsoft Corporation, MSDN Library.

