"Program, Enhance Thyself!" – Demand-Driven Pattern-Oriented Program Enhancement

Eli Tilevich Godmar Back

Department of Computer Science, Virginia Tech
Blacksburg, VA 24061, USA
{tilevich, gback}@cs.vt.edu

Abstract

Program enhancement refers to adding new functionality to an existing program. We argue that repetitive program enhancement tasks can be expressed as patterns, and that the application of such enhancement patterns can be automated. This paper presents a novel approach to pattern-oriented automated enhancement of object-oriented programs. Our approach augments the capabilities of an aspect compiler to capture the programmer's intent to enhance a program. In response to the programmer referencing a piece of functionality that is non-existent, our approach automatically synthesizes aspect code to supply the required functionality transparently. To improve flexibility and facilitate reuse, the synthesis and application of the new functionality is guided by declarative *when-then* rules, concisely expressed using a rule base.

Our extensible automated program enhancement system, called DRIVEL[1], extends the AspectJ compiler with aspect generating capabilities. The generation is controlled using the DROOLS rules engine. To validate our approach and automated tool, we have created a collection of enhancement libraries and used DRIVEL to apply them to the LibX Edition Builder, a large-scale, widely-used Web application. DRIVEL automatically enhanced the LibX Edition Builder's XML processing modules with structural navigation capabilities and caching, eliminating the need to implement this functionality by hand.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming — Program Synthesis, Program Transformation; D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.2.13 [*Software Engineering*]: Reusable Software — Reusable Libraries; D.2.7 [*Distribution, Maintenance,and Enhancement*]; D.3.3 [*Language Constructs and Features*]: Frameworks, Patterns; D.3.4 [*Programming Languages*]: Processors — Compilers

General Terms Design, Experimentation, Languages

Keywords aspect-oriented programming, patterns, program enhancement, rules engines, meta-programming

[1] **DRIVEL**: **D**emand-driven **R**ules-based **I**ntelligent **V**alue-adding **E**nhancement **L**ibraries.

AOSD'08, March 31 – April 4, 2008, Brussels, Belgium.

1. Introduction

Software maintenance constitutes the majority of the total software development effort and cost [5, 39], with program enhancement being its dominant activity [42, 59]. Program enhancement is concerned with adding, changing, or removing functionality to adapt software to meet new or changed technological or business requirements [35] and is inherently difficult, costly, and error-prone. Software development realities that complicate program enhancement are both diverse and pervasive: enhancement programmers are typically not those who developed the system, enhancements are made under time pressure to meet deadlines, the software documentation is often sparse or missing altogether, and portions of the existing code may carry restrictions against being changed.

This paper presents a novel approach that automates software enhancement tasks commonly performed as part of developing and maintaining object-oriented programs. We introduce our approach by means of a concrete software development scenario. Consider a Java program under construction that has the following `useVisitor` method:

```
class SomeClass {
    A a;
    B b;
}

void useVisitor(SomeClass o) {
    o.accept(new SomeClassVisitor() {
        public void visit (A a) {
            System.out.println("visited A " + a);
        }
        ...
        public void visit (B b) {
            System.out.println("visited B" + b);
        }
    });
}
```

The method `useVisitor` above takes an object of type `SomeClass` as a parameter and uses it as a participant in the *Visitor* pattern [18]. The Visitor Pattern provides a flexible approach to extending the functionality of a program by separating its functionality and object structure. By providing `accept` methods, object structures can pass around different `Visitor` objects, implementing a traversal strategy. Using the Visitor pattern is as straightforward as providing a specific implementation of a `Visitor` interface (or an abstract class) and passing it as a parameter to the root object of a structure.

The example above assumes that class `SomeClass` does indeed have a method `accept` that provides the functionality required by the Visitor pattern. It also assumes that an interface or abstract class `SomeClassVisitor` exists, from which anony-

mous subtypes can be derived. If this is not the case, the compiler will report errors such as *Invalid Method: accept* or *Invalid Type: SomeClassVisitor*, until the programmer provides a suitable implementation of the Visitor pattern.

Despite the aforementioned software engineering benefits afforded by employing the Visitor pattern, implementing its core functionality may not be trivial. For one, as is usually the case when using patterns, applying the Visitor successfully requires that certain preconditions be met. Specifically, a hierarchical object structure with a "has-a" relationship must be present, requiring the programmer to examine all classes in the program. In addition, since this pattern encodes an object traversal strategy, each usage scenario is likely to require a different strategy. Because this traversal strategy is effected through a particular implementation of the `accept` method, accomplishing this task requires a thorough understanding of the underlying object structure. Finally, the Visitor interface that specifies the `visit*` methods is application-specific. If implemented as a class or aspect library, the library would have to be updated whenever the underlying object structure changes, which is cumbersome and error-prone, especially if the underlying classes are generated by a code generation tool.

The novel approach described in this paper alleviates many of the difficulties of enhancing object-oriented programs outlined above. Our approach enables program enhancement *on-demand* by augmenting an aspect compiler with sophisticated enhancement capabilities. In the code example above, our augmented aspect compiler intercepts the reporting of the *Invalid Method* and *Invalid Type* errors resulting from compiling method `useVisitor` and evaluates the flagged errors against a set of previously-defined, declarative *when-then* rules. If the rules indicate that the Visitor pattern is indeed desired in this part of the program, an aspect generator synthesizes aspect-oriented code to enhance the program as specified by the rules. In our example, a rule could indicate that an invalid type `SomeClassVisitor` indicates the need for a `Visitor` for `SomeClass`.

The aspect generator executes as part of the compiler, and thus has access to information about the partially compiled program; in the example, it has access to the field declarations of `a` and `b` in the partially compiled class `SomeClass`. It also has access to the compiler's name resolution mechanism, allowing it to disambiguate the meaning of the identifier `SomeClass` based on the specific scope in which the invalid type error occurred. The enhanced code will then be automatically recompiled, and the programmer will be able to use the Visitor pattern exactly as coded in the `useVisitor` method. To the programmer using the enhancement-providing compiler, this process is transparent, making it appear as though the program had been enhanced automatically.

From a software engineering perspective, our approach enables the separation of concerns pertaining to enhancing object-oriented programs by distinguishing between *enhancement providers*, a party supplying prepackaged enhancement libraries, and *enhancement consumers*, a party using the supplied enhancement libraries as part of their software development cycle. Specifically, our approach distinguishes between three separate enhancement concerns: providing general enhancement libraries, defining the logical rules for applying the libraries, and guiding the application of enhancements in a specific software project as defined by the rules.

This separation makes it possible for different parties to tackle those enhancement challenges that are commensurate with their level of programming expertise. For example, a third-party software vendor could develop general enhancement libraries, an expert programmer could define which enhancement libraries will be used, adapt those libraries to the software conventions used in a given software project, and define the conditions under which they should be applied. Finally, rank-and-file developers could simply use the libraries as parameterized by the expert programmer. Each party can express or refine the conditions under which enhancements are applied by contributing declarative *when-then* rules to a rule base. The declarative nature of the rules not only increases flexibility and expressiveness, but also facilitates reuse.

While we believe that our automated approach can be beneficial for many program enhancement scenarios, it proves particularly useful for those cases when it is not desirable or perhaps even possible to modify the source code of a program, such as in situations when the program's code needing enhancement is part of a third-party library or automatically generated. Because our approach employs aspects as the mechanism for adding functionality, it does not require explicit changes to the source code of a program, and yet retains the type-safety and compile-time checking capabilities of the Java language. As a result, our approach not only facilitates program enhancement, but also enables it for a whole class of programs currently deemed not easily amenable to enhancement. In addition to facilitating the enhancement of code we cannot change, we found that the use of aspects helped because many of the enhancements we consider are in fact cross-cutting concerns.

The rest of this paper is structured as follows. Section 2 introduces Enhancement Patterns, a methodology for describing and classifying common program enhancement tasks. Section 3 details our approach to demand-driven automated enhancement and an automated tool implementing this approach. Section 4 discusses the challenges of enhancing automatically generated code using the example of Java XML binding code. Section 5 presents a case study of applying our automated enhancement tool to the development of a real-world Web application. Section 6 reviews related work. Section 7 presents future work, and Section 8 concludes.

2. Enhancement Patterns

Before presenting the enabling concepts of our approach to automated program enhancement, we clarify the kinds of enhancements that our approach aims to support. Because our ultimate objective is to enable *automated* program enhancement, we must ask which types of enhancements are amenable to automation. To be amenable to automation, a group of programming tasks should be repetitive and easy to categorize. In that regard, the enhancements concerned with adding arbitrary new functionality to an existing program are likely to be non-repetitive and may defy categorization. In other words, such enhancements are application- and case-specific.

Thus, as a means to meet the stated objectives of automation, our approach is concerned not with adding new arbitrary functionality but with common program enhancements. Furthermore, the machinery for applying these common enhancements should be expressible as aspects in a mainstream aspect-oriented language such as AspectJ [36]. To ensure greater flexibility, the aspects are automatically generated to better specialize them for individual enhancement scenarios. As concrete examples, consider enhancement tasks such as adding a GoF [18] design pattern to an existing code base or enforcing an implicit program invariant (e.g., all the objects inserted into a hash table must provide consistent `equals` and `hashCode` methods). As examples of larger-scale program enhancements, consider the long-standing challenges of adding program capabilities such as persistence and distribution [56], adapting existing program code for reuse in a different software product [3, 46], or integrating a COTS component with an in-house software system [4, 16]. The program enhancements represented by the examples above do occur in modern software development on a regular basis. Furthermore, the repetitive nature of these en-

hancement tasks lends itself to their categorization and subsequent automation by observing the following:

- Such enhancements are accomplished in common ways rather than randomly.
- These common ways represent accumulated programming experience/wisdom in program enhancement.
- These common program enhancements are amenable to automation through programming tools.
- These enhancements cannot generally be expressed as aspect libraries alone, but require code generation.

The above observations have led us to explore common enhancement strategies that programmers follow to enhance existing programs with additional functionality. To reflect the pattern-based nature of these strategies, we call them *Enhancement Patterns*. An enhancement pattern captures common modifications made to an existing program to enable its operation in an additional, and usually unforeseen, context. Unlike classical GoF Design Patterns [18], whose primary purpose is to guide the initial stages of object-oriented software development (i.e., design), Enhancement Patterns describe common ways of adding functionality to pre-existing or partially complete programs.

Patterns in software development have been advocated as an approach for disseminating the shared knowledge of experienced programmers in solving common problems. The concept of Enhancement Patterns shares the objective to capture and organize the accumulated experience and best practices of common program enhancements, creating an extensive and ready-to-use catalog of Enhancement Patterns. The benefits of cataloging Enhancement Patterns include providing a valuable resource for developers in search of an effective and elegant enhancement solution, creating a shared vocabulary for communicating about common enhancement tasks, and guiding the creation of automated program enhancement tools.

Example Patterns

Software developers commonly enhance programs in ways specific to their application domain. The benefits of expressing such enhancements as patterns depend on their range of applicability and ease of automation. The following catalogue of enhancement patterns represents those enhancements that we have found recurring in our own software development practices and believe to be particularly amenable to be captured as patterns.

2.1 Shapeshift

The Shapeshift pattern describes modifications to the external interface of a collection of objects for use in a new context. It enables treating a group of unrelated classes in a uniform fashion or using them in some other fashion for which they were not originally written. The Shapeshift enhancement patterns occurs in many flavors and implementations.

As a concrete example, consider a group of Java classes (possibly automatically generated) that provide a textual representation for their instances via a method called `asText` rather than via the standard method `toString`. This precludes these classes from being used in all the multiple standard contexts that rely on the `toString` method to get a textual representation of a Java object. Shapeshift can enhance these classes by introducing a `toString` method that invokes the original `asText` method.

As a second example, consider linking (or binding) of properties of Java Beans, a technique frequently used in component-based applications. Suppose a property p_1 of bean b_1 should be linked to property p_2 of bean b_2. Such a pattern can be implemented by adding an instance (or aspect method) to b_1 that observes property changes on b_2 and sets $b_1.p_1 = f(b_2.p_2)$ whenever the value of $b_2.p_2$ changes, based on some predefined function f. When coded manually in Java, these methods clutter the code with large amounts of cut-and-pasted code, which can be hard to maintain. In addition, manual coding requirings anticipating all possible properties to which an application may create bindings. Alternatively, binding can be implemented using runtime reflection. However, runtime reflection cannot detect misnamed or non-existing properties until runtime, and it is significantly slower than invoking getter and setter methods by direct method call. By applying the Shapeshift pattern on demand, this repeated pattern can be expressed in generative aspects, created only when used, and whose output is subject to compile-time checking for early error detection.

As a Shapeshift example from a different domain, consider an Inversion-of-Control container [30] that manages components of a certain type. Specifically, the components are expected to provide methods that the container can invoke (e.g., `init`, `service` and `destroy` for Java Servlets [12]). Nevertheless, recent designs of container-based frameworks enable the use of POJO's (Plain Old Java Objects) as components managed by a container to improve flexibility and ease maintenance. Frameworks such as Spring [29] perform shapeshift using bytecode engineering at class load time to add the required methods to the loaded POJOs.

An implementation of the Shapeshift pattern should provide convenient support for introducing new methods and fields to a class. One motivation for expressing these transformations as a pattern is that the added convenience of applying Shapeshift enhancements eliminates the need to emulate the required functionality at run-time (i.e., using reflection facilities) and allows for compile-time checking.

2.2 Navigatable Structure

The Navigatable Structure pattern describes the process of enhancing an existing structure that is composed of multiple objects to facilitate its navigation or traversal. An example was presented in Section 1, in which client code was enhanced with a hierarchical Visitor Design pattern. The Visitor Design pattern is widely used, such as in compilers and other code analysis and transformation tools. Its implementation follows a strategy that is specific to the structure being navigated. For instance, for an object representing an XML document, the navigation strategy may be determined by the XML schema describing the XML document's elements and attributes, including their order and constraints on their occurrence. Writing visitors requires a correct understanding of the pattern, and a reasoned strategy on how to apply the pattern to the given situation. Once this strategy is identified, coding the pattern by hand becomes time-consuming and error-prone.

Not all structures are amenable to navigation using the Visitor pattern. For instance, navigation could be provided via the `Iteratable` interface for a structure. Imagine being able to say

```
for (Apple a : fruitBasket.apples())
    process(a);
```

to iterate over the apples in a heterogeneous set of fruits that originally did not provide a way to do that.

Existing object structures are often enhanced with such navigation capabilities, and these enhancements tend to follow common strategies. This observation justifies capturing and expressing these strategies in the Navigatable Structure enhancement pattern.

2.3 Equivalence Relations, Factor Sets, and Ordering

An equivalence relation is a reflexive, symmetric, and transitive relationship that creates some notion of equivalence among objects. In Java, classes may override the inherited `equals` method to test for membership in this equivalence relation. A common way to implement this equivalence relation is by using a conjunction of field-wise equivalence test for all or a subset of fields. Though com-

paratively trivial, implementing `equals` correctly requires careful coding to ensure robustness and consistency with the `hashCode` method, and is repetitive. The equivalence relation pattern captures this strategy of implementing `equals` and `hashCode`.

Based on this equivalence relation, a factor set can be computed. If objects are mapped to their class representatives in their respective equivalence class, they can be compared via the reference equality operator (i.e., ==), which improves performance and expressiveness and can reduce memory consumption. For Java strings, the `String.intern` method provides a mapping of a string to its class representative. The Factor Set enhancement pattern creates an `intern` method for any class wishing to use it. Like `equals`, implementing `intern` is repetitive, and subject to subtle mistakes by non-experts (e.g., maintaining a factor set using strong references rather than weak references is likely to result in a memory leak).

For some applications, such as insertion into a sorted map or set, the ability to compare two objects according to a predefined partial or total order is required. In Java, this *natural ordering* is provided by implementing the `Comparable` interface and its method `compareTo`. A naive, but correct implementation could simply return the cumulative result of comparing individual fields. Alternatively, the implementation could be refined to include a subset of fields, and determine the order in which fields are being compared. The natural ordering of a class should be consistent with equals, e.g. the comparator should signal equality if and only if the `equals` method does so as well. An enhancement pattern providing both ordering and equivalence can guarantee such consistency.

2.4 Instant Cache

The Instant Cache pattern describes the functionality required to cache the results of an expensive computation. For example, the results of parsing a textual representation of an object structure may need to be cached. If the textual representation does not change between different requests to retrieve the object structure, then a cached version can be returned, saving the time required to re-parse and recreate the structure. However, if the textual representation is updated, then the object structure has to be recreated. This scenario exemplifies a typical case in which caching can be employed to improve performance.

In many applications, caching follows a common template in which a cache is implemented as a lookup table. A *key* is used to look up a *value*, which is returned on a cache hit. On a cache miss, the value is recreated from the key. In addition, caches should be kept to a limited size, or use weak references. The "Instant Cache" enhancement captures this pattern by automatically building a complete cache based on a cache miss handler. Any Java method can be labeled a cache miss handler - the enhancement pattern infers the key from its formal parameters and its value from its return value. It creates aspect methods that maintain the cache, allow for the invalidation of individual or all elements, and provides convenient accessor methods if the cached values are themselves compound types.

2.5 Audible Model

The Audible Model pattern enhances an arbitrary object model to be used as the Model entity of the Model-View-Controller architecture. The adjective *audible* refers to the added ability to "listen" to changes in the Model's state. As a concrete example, this pattern can solve a common problem in GUI programming arising when a model must notify graphical views of state changes.

Audible Model works by assigning model classes the roles of an `Observer`, an `Observable`, or both. `Observable` classes select sets of field changes and method calls that result in mutating a part of an object's state (i.e., field sets and mutator method calls). All changes to the state raise notifications to the registered `Observers`. `Observer` classes, in turn, provide the notification events handling logic. In addition, `Observer` classes might also have `Observable` fields for which they might need to receive notifications.

The implementation of Audible Model exhibits many variations and requires careful attention to detail. For instance, an `Observer` may wish to filter out certain fields. In other instances, the notification must be delayed such that `Observers` do not see an inconsistent state of an object. Finally, Audible Model may be combined with Navigatable Structure, for instance, in order to provide hierarchical notifications enabling `Observers` to listen to state changes in a connected object structure.

2.6 Veto

A companion pattern to the Audible Model enhancement pattern is the Veto pattern, in which the invocation of a method or a set of methods, or updates to instance or static fields can be disallowed in an enforceable fashion. For example, a model can be rendered read-only by applying Veto to all of its mutator statements or methods.

3. Demand-Driven Program Enhancement

We detail our approach to demand-driven automated enhancement by illustrating the implementation of our extensible program enhancement system, called DRIVEL, which is an acronym for **D**emand-driven **R**ules-based **I**ntelligent **V**alue-adding **E**nhancement **L**ibraries.

3.1 Overview

Figure 1 shows the main steps of our approach. An enhanced AspectJ compiler (EAJC) integrates enhancements into the compilation process by acting as a host for enhancement plugins. Enhancement providers package *enhancement libraries* as .jar files, and EAJC loads them from locations specified as command line arguments. Typically, an enhancement library provides an implementation for a single enhancement pattern. An enhancement library jar contains aspect and Java code generators and a set of logical rules. These rules express the conditions that must be fulfilled to correctly apply an enhancement pattern contained in the library to *any* code base. In addition, the users of an enhancement library (typically lead programmers) can control the specific application of enhancements to their code by adding additional sets of rules to the ones supplied with the libraries. They can also customize how the enhancements are applied.

Applying an enhancement entails generating AspectJ and Java code files that are then added to the compilation set. Compiling such an enhanced compilation set may trigger the application of additional enhancements. This process is repeated until no more enhancements can be applied. Finally, all remaining unresolved errors are displayed to the user.

3.2 Rules Engine

The application of enhancements is controlled by a rules engine. We use the DROOLS engine [52], a Java implementation of the RETE algorithm [17]. The rules engine maintains a rule base (a set of rules) and a working memory (a set of asserted facts). A rule consists of an antecedent, expressed as a predicate in first-order logic, and a list of consequences, triggered if the current set of facts makes the antecedent true. Consequences may add new facts to the working memory, retract facts from the working memory, or modify facts. The RETE algorithm propagates these changes through the working memory by forming and processing a network of consequence-antecedent relationships. This process is also referred to as truth maintenance.

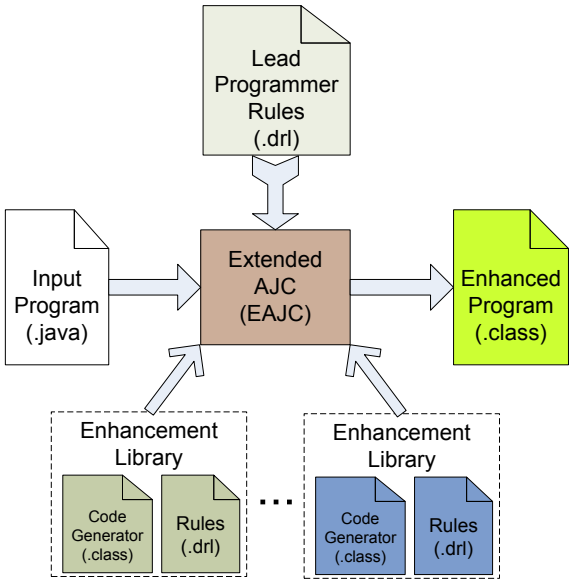

Figure 1. Enhancement Programming Model.

3.2.1 Facts.

Our implementation uses the following types of facts:

Code entity facts. As the compiler processes input source code, all code entities are represented as facts; these include packages, classes, methods, fields, and annotations. To facilitate the convenient use of code entities in antecedent expressions, we ensure that their properties have corresponding sets of bean-style accessor methods.

Compile error facts. All errors raised during compilation are represented as facts. Examples of such errors include invalid or missing types, methods, or incompatible assignments. Error facts often contain context information inferred by the compiler, which is used to guide the enhancement process. For example, when reporting a missing method error, the compiler might have the information about the expected return type of the method. Each reported compilation error is stored along with its occurrence scope. This enables scope-specific identifier lookup during the application of enhancements.

Enhancement-provided facts. We employ facts as the coordination mechanism between different enhancements, user rules, and the host compiler. For instance, an enhancement can signal to the host engine that a recompilation is needed by inserting the fact "Need.Recompile" into the working memory. Enhancements can also express dependencies on each other by inserting facts that express those dependencies. For instance, a factor set enhancement requires an equivalence relationship; it may express it by inserting a fact "NeedEquivalence" for a particular class. Enhancements can also use facts to 'remember' whether they have been already applied to a class (e.g., a fact "HaveEnhancedWithX" may signal that a particular enhancement X has already been applied).

DROOLS enables the expression of facts by using any Java object. Furthermore, equality assertion (i.e., determining whether facts are identical) is expressed by using the `equals` method of the objects used as facts. This design feature of DROOLS has helped to streamline our implementation. Since DROOLS calls the `equals` method on each reported compiler error message before inserting it as a fact, we are able ensure that multiple, but identical errors appear only once in working memory. Therefore, the corresponding enhancement is triggered only once.

3.2.2 Rules.

When an enhancement library is loaded by the host environment, it uses its corresponding set of rules that guide its application into the host's rulebase. The rules are expressed in a domain-specific language in a *when/then* format. Antecedents are expressed in MVEL (a Java-based scripting language) or Java and must not have side-effects, whereas consequences can consist of arbitrary MVEL or Java code.

Enhancement rules. Enhancement rules use a declarative approach to describe the assumptions under which enhancements should be applied. In response to the assumptions holding true, the corresponding enhancements rules "fire," typically adding an instance of an enhancement to the working memory. The actual application of the enhancement, however, is delayed until later.

For instance, consider an enhancement that adds a method `public String toStringLong()` to a class. This method returns a detailed printable representation of an object such as one containing all the object's fields and their values. Such an enhancement could be triggered if the compiler encounters an invalid method error, whose method name value is "toStringLong." If this error is encountered during compilation, the toStringLong enhancement will create an enhancement object. Below, the variable 'm' is bound to any encountered 'InvalidMethod' facts meeting the condition that the name of the missing method is "toStringLong." `AddToStringLong(m)` returns an enhancement object, and the `insert` method adds it as a new fact into the working memory.

```
rule "Provide a toStringLong() aspect"
when
    m : InvalidMethod ( methodName == "toStringLong" )
then
    insert (Enhancements.AddToStringLong(m));
end
```

A second example shows how rules can be triggered even in the absence of compile time errors using annotations. The annotation `@NaturalOrdering` triggers an enhancement that adds a natural ordering to a class, allowing it to be used in `java.util.*` containers without requiring the use of a comparator.

```
rule "Provide a natural ordering"
when
    t : Clazz ( annotations['NaturalOrdering'] != null )
    not ( Enhancements.HaveNaturalOrdering ( clazz == t ) )
then
    insert (Enhancements.AddNaturalOrdering(t));
end
```

The 'not HaveNaturalOrdering' construct prevents the application of this rule if the class already defines a natural ordering. Applying the enhancement will assert a "HaveNaturalOrdering" fact for the class that is being enhanced. If the class already provides a natural ordering via a `compareTo` method, a rule can add this fact as follows:

```
rule "Disallow natural ordering annotation
     if compareTo() is present."
salience 10
when
    m : Method (name == "compareTo",
                signature == "(Ljava/lang/Object;)I")
then
    insert(new Enhancements.HaveNaturalOrdering (m.getClazz()));
end
```

The 'salience 10' argument gives this rule higher priority than the "Provide a natural ordering" rule, thereby ensuring that its an-

tecedent is falsified before it fires, preventing the accidental application of the enhancement (which would result in a compile error).

Rules can be used to achieve *composition* of enhancement patterns. For instance, the factor set enhancement depends on the equivalence enhancement. A rule set that expresses this dependency is shown below.

```
rule "Request factor set enhancement"
when
    m : InvalidMethod ( methodName == "intern",
                        expectedReturnType == receiverClass
                      )
    t : Clazz ( name == m.receiverClass )
then
    insert (new Enhancements.NeedEquivalence(t));
    insert (new Enhancements.NeedIntern(t));
end

rule "Build equivalence enhancement if equals()
     is not overridden"
when
    n : Enhancements.NeedEquivalence ()
    not ( Method (name == "equals", clazz == n.clazz) )
then
    insert (Enhancements.AddEquivalence(n.getClazz()));
end

rule "Provide an intern() aspect if needed and
     if we have equivalence"
when
    n : Enhancements.NeedIntern()
    Enhancements.HaveEquivalence(clazz == n.clazz);
then
    insert (Enhancements.AddIntern(n.getClazz()));
end
```

In this example, if the compiler encounters a call to `intern` in a context in which the expected return type matches the receiver's type, and if no `intern` method is defined, the first rule will insert a fact indicating the need for the equivalence relationship and factor set enhancements. The second rule is triggered by the presence of a fact indicating the need of an equivalence relationship enhancement. If the class does not already define an `equals` method, the equivalence enhancement fact is inserted, but the enhancement is not applied immediately. Once DRIVEL applies this enhancement to the given class, it will insert a fact "HaveEquivalence," parameterized with the enhanced class. This fact, in turn, will make the antecedent of the third rule true, resulting in the insertion of the factor set enhancement.

User rules. Simply including an enhancement library does not apply its enhancements. Instead, user rules specify which of the enhancements to apply. The following rule, for example, applies all available enhancements:

```
rule "Apply All Enhancements"
when
    e : Enhancement();
then
    e.apply();
end
```

When fired, this rule invokes the `apply` method for each enhancement found in working memory, which includes all enhancements inserted by the consequences of the detection rules provided by all loaded enhancement packages. This simple rule works because enhancements are required to subtype `Enhancement`. An excerpt of a more realistic rule is as follows:

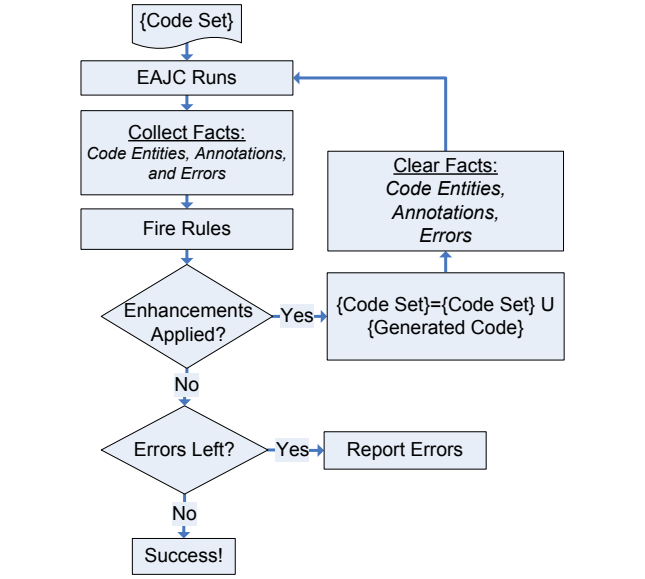

Figure 2. DRIVEL Flowchart.

```
rule "Add Visitors"
when
    e : AddVisitor(
      enhanceClass matches "org\.libx\.xml\..*"
    );
then
    e.apply();
end
```

This rule applies the visitor enhancement if the class being enhanced is in the `org.libx.xml.*` package. In this way, the programmer can determine exactly when compile time errors, annotations, or other code entity facts should lead to enhancements, and when such enhancements would be spurious, unnecessary, or downright wrong. The programmer can also provide supplementary facts that guide or customize the application of enhancement patterns. For example, for the factor set enhancement described in the previous subsection, a programmer could declare that an already provided `equals` method is suitable for the factor set enhancement by inserting a *HaveEquivalence(...)* fact for the type in question.

3.3 Extended Aspect Compiler

We have extended the AspectJ compiler, version 1.5.3. Our changes are confined to two modules of the compiler: the main driver and the error reporting facilities. The addition of code entity facts takes place upon the completion of the code generation phase. Compile error facts are added immediately when they are signaled, but the display of errors to the user is suppressed.

The existing AJC implementation already allowed for repeated activation when operating in incremental mode. We modified its incremental mode so that the compiler consults the fact base before each incremental activation. After each activation, we check if an enhancement requires recompilation and trigger a new compiler activation if so. We currently recompile all files, although this could be optimized by allowing the enhancement to point out which files need to be recompiled. We remove all code entity and compile error facts between compiler activations, but keep facts that were inserted by consequences. If no enhancement signals that recompilation is needed, but there are still compile time errors, we switch the driver back into a mode that reports errors immediately to the user. At this

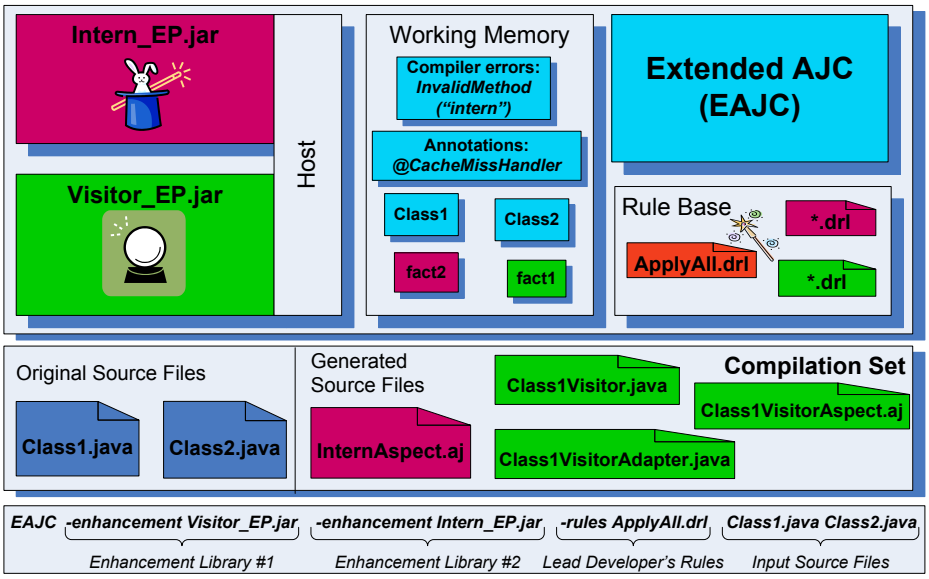

Figure 3. DRIVEL Enhancement Process.

point, the user will find the enhanced program available for further examination. A control flow diagram of this process is shown in Figure 2. Figure 3 shows the interaction of the components of our system.

We have created a set of class libraries that facilitates the extension of our infrastructure. By using these libraries, an enhancement library developer can add enhancements by simply subclassing one of our abstract adapter classes[2], providing a concrete implementation of its `apply` method, and adding a set of DROOLS rules to specify the conditions under which the enhancement should be applied. As an additional resource for creating new enhancement libraries, we provide convenience classes that facilitate the actual generation of the aspect and Java code.

4. Enhancing Automatically Generated XML Data Binding Code

One of the most common approaches for automating menial programming tasks is automatic code generation [14, 43]. A code generator takes a high level description as input and generates lower level code. Because the input specification is simpler and shorter than the code it generates, code generation not only saves time and effort, but also helps avoid programming errors, thereby increasing programmer productivity [1, 55].

Alas, the benefits of automatic code generation can diminish rapidly if the generated code does not fully satisfy the requirements for the task at hand [27]. For example, the generated code may not adhere to the established in-house coding conventions large software organizations typically follow. Generated code may miss important capabilities such as concurrency control, making it unsuitable for safe use in concurrent contexts [57]. Or, generated code may not provide support for applicable design patterns. Even if it does, the specific expression of those design patterns may be presented in an awkward or incompatible way, complicating its integration.

Refining automatically generated code by hand to meet the requirements is not a viable solution: every time a code generator is re-run (e.g., in response to a changed specification), the hand-written changes will likely be lost and need to be re-applied, wasting programming effort. Changing the source code of a code generator to customize its functionality might not be feasible either, as the source code could be unavailable. Open-source code generators, which could be changed, are often intricate and large, making changes to their implementation a prohibitively difficult and time-consuming undertaking. In fact, this process could prove as expensive as developing a custom, in-house code generator, negating the time-saving benefit of using a third-party code generator. By contrast, our approach can enhance the capabilities of automatically generated code without the shortcomings outlined above. Aspect-based enhancement patterns enable the tailoring of such code without explicit changes either to the generated code or to the code generator.

XML [20] has become the de facto standard for representing, storing, and transporting persistent data on disk or in databases. Operating on XML data requires an efficient in-memory representation. For Java, a number of XML data binding conventions describe programmatic ways to represent XML data at runtime. If the XML document's structure is described by a Document Type Definition (DTD) or XML Schema description, code generators such as Castor [11] can be used to generate a Java binding. This binding consists of number of Java classes that represent the XML document's elements and attributes. The generated code has methods that support marshaling and unmarshaling of XML data, in-memory manipulation, and validation. Generators such as Castor significantly reduce the complexity of XML processing by ensuring that the produced XML is well-formed and valid. However, if the XML schema changes, the classes providing the Java binding must be regenerated, which precludes manual changes to the binding code. To adapt Castor for our needs, we applied several enhancement patterns to the XML-related code Castor produces.

4.1 Applying Shapeshift

We implement a number of convenience methods that shift the classes generated by the castor API into shapes which we found easier to work with. For instance, we added a method `xml-Descriptor` to each Castor class that retrieves a descriptor that represents the underlying XML schema at runtime. Even though the

[2] We provide adapter classes for invalid method errors, invalid type errors, and annotations found in the code.

mapping from an object's class to its descriptor instance is known statically at code generation-time, Castor did not provide a method to retrieve it at run time. As a second example, we added a method `toXMLString` to convert an object directly into its XML representation, a piece of functionality that requires several cumbersome steps to implement in vanilla Castor.

4.2 Applying Navigatable Structure

Castor does not provide support for any type of Visitor pattern. However, the hierarchical structure of an XML document often requires traversal, typically in an order that follows the structure defined in the XML Schema or DTD. We implemented a Castor-specific visitor enhancement. The example below shows the code produced by the enhancement for the DTD production shown at the top.

```
/* DTD production:
   <!ELEMENT edition (name,links,catalogs,openurl,proxy,
                      options, searchoptions?, additionalfiles ?)>

    Code generated by enhancement: */

public aspect EditionVisitorAspect
{
  public void org.libx.xml.Edition.accept(EditionVisitor v) {
    v.visit(this);
    this.getName().accept(v);
    this.getLinks().accept(v);
    this.getCatalogs().accept(v);
    this.getOpenurl().accept(v);
    this.getProxy().accept(v);
    this.getOptions().accept(v);
    if (null != this.getSearchoptions())
      this.getSearchoptions().accept(v);
    if (null != this.getAdditionalfiles())
      this.getAdditionalfiles().accept(v);
  }
  ...
}
```

This strategy captures the declaration order of the children of an 'edition' element, and it captures the fact that some children are optional, requiring a null-check. Our enhancement also handles multi-valued elements, which are represented as vectors by Castor. When creating the logic for the `accept` methods, our implementation relies on the descriptors generated by Castor. It also emits a specific `Visitor` interface type and `VisitorAdapter` classes. The application of the pattern is guided by castor-specific rules as described in Sections 1 and 3.

4.3 Applying Audible Model

Castor represents XML elements as Java beans. A configuration option allows Castor to generate support for property change events through a simple variation of the Observer design pattern. However, traditional bean support is insufficient for a number of reasons: first, it does not allow an application to veto a property change since constrained properties are not supported. Second, the `java.beans.PropertyChangeEvent` does not pass a reference to the bean being changed to the listener, requiring custom listener instances for each object. Third, and most importantly, there is no support for propagating events that would allow recursive observers to listen to any change in a given subtree.

We used aspects to generate support for a recursive, vetoable Observer. An example of the code generated by this enhancement is shown in Figure 4. This generated aspect intercepts field assignments to the field `_proxy` in class Edition, representing the DTD elements 'edition' and 'proxy', respectively, from the DTD above.

This aspect vetoes changes if the programmer has invoked a `setReadOnly` method, which this enhancement also introduces. Because the order in which different observers will be notified cannot be guaranteed, it is impossible to ensure that no Observer will have seen the changes that are to be vetoed. Therefore, we have decided to proactively prevent changes, rather than allowing an Observer to request that changes be undone. If the change is allowed to proceed, Observers are notified of the new value. The implementation also passes along the object being changed, allowing a single Observer to observe multiple objects.

To ensure that recursive Observers will continue to receive notifications even when updates result in the replacement of subtrees of the document, a helper method adds all current, recursive observers to the new `_proxy` child. To add recursive observers, the enhancement pattern uses a hierarchical Visitor that traverses the document structure and adds observers to each node. To generate the necessary code, the pattern triggers the Hierarchical Visitor enhancement pattern via its rule set. Finally, we note that the rule set allows the Audible Model enhancement to be triggered independently of the Veto enhancement, depending on which methods are used in an actual application.

5. LibX Case Study

Many of our enhancements were driven by concrete needs in the LibX edition builder application, whose development one of the authors supervised. LibX [2, 40] is a browser plugin that provides users with direct access to electronic library resources through a toolbar, context menu, and by providing web localization facilities. Since library resources are local to a particular user community, the plugin is available in heavily customized editions. There are currently over 200 editions, built by academic or public libraries in over 6 countries. A LibX edition is represented by an XML configuration file. These configuration files are described by a reasonably complex DTD with currently 29 elements and over 120 attributes, and their grammar changes *frequently* as new features are added to LibX, whose configuration the edition builder must immediately support.

The LibX Edition Builder[3] is an AJAX [19] application that allows librarians to maintain the configuration of their library's edition. The LibX Edition Builder is written almost entirely in Java, it uses the ZK [9] toolkit for communicating with a front-end JavaScript library in the client's browser. The application is composed from ZK widgets, which are rendered as HTML at the client. It uses a model/view/controller (MVC) approach [18]. The model is composed of a hierarchy of castor-generated objects, representing an edition's configuration in memory. Controller classes process client-side events and update the model accordingly. The view is represented by ZK widgets which listen to changes in the model. We have been applying several enhancement patterns to this application, resulting in the generation of over 2,200 lines of (uncommented) AspectJ code and 250 lines of Java code. This number compares to 38,600 lines of (commented) code produced by Castor, and 8,600 lines of programmer created Java code in the core edition builder application.

5.1 Shapeshift.

We used the convenience methods discussed in Section 4 to consult the XML schema at runtime. For example, the schema specifies that configuration attributes are required, whereas others are optional, which influences the UI's display and handling of these attributes. For instance, telling whether a DTD attribute property was listed as `#REQUIRED` or `#IMPLIED` became as simple as calling `bean.xmlDescriptor().isRequired()`. We use the enhanced `toXMLString()` methods when inserting XML elements into a database we keep to allow different libraries to share

[3] `http://libx.org/editionbuilder`

```
/**
 * Automatically generated code by DRIVEL;
 * Castor enhancements library.
 */
privileged aspect EditionObserverAspect {
  declare parents : org.libx.xml.Edition implements Observer.Observable;

  org.libx.xml.Observer.ObserverList org.libx.xml.Edition.__observer_support = new org.libx.xml.Observer.ObserverList();

  public void org.libx.xml.Edition.addObserver (org.libx.xml.Observer ob) {
    if (ob instanceof org.libx.xml.Observer.Recursive) {
      this.accept(new org.libx.xml.EditionObserverVisitor((org.libx.xml.Observer.Recursive)ob));
    } else {
      this.__observer_support.addObserver(ob);
    }
  }

  public void org.libx.xml.Edition.removeObserver (org.libx.xml.Observer ob) { ... }

  private boolean org.libx.xml.Edition.__readonly;

  public void org.libx.xml.Edition.setReadOnly(boolean value) {
    this.__readonly = value;
  }

  void around(Edition obj, Proxy newvalue) :
      set(Proxy Edition._proxy) && target(obj) && args(newvalue)
  {
    if (obj.__readonly) {
      throw new Observer.VetoException("cannot change field '_proxy'");
    }

    proceed(obj, newvalue);

    obj.__observer_support.notify(obj, "_proxy", newvalue);
    if (newvalue instanceof Observer.Observable) {
      obj.__observer_support.propagateRecursiveObservers((Observer.Observable)newvalue);
    }
  }
  ...
}
```

Figure 4. Observer Aspect Generated by Audible Model Enhancement.

catalog configurations, such as for national resources like OCLC's WorldCat.

5.2 Audible Model.

We used the recursive observer enhancement described in Section 4 to drastically simplify the notification logic of our MVC implementation. The edition builder needs to save a configuration file to disk whenever *any* of its contents change. We chose this approach to ensure immediate persistency, even in the face of client or network failures. It eliminates the need for a "Save" button in the user interface, because any changes the user enters are immediately saved.

Prior to using this enhancement, we used a depth-first traversal based on runtime reflection to recursively add listeners to all nodes in the document. This logic was error prone (for instance, blindly adding observers to static fields resulted in memory leaks, and forgetting to add observers when elements were replaced resulted in lost notifications), and slow due to the use of reflection. Implementing this logic in an enhancement aspect avoids the use of reflection, and allows re-using the complicated logic in other software that uses Castor.

5.3 Veto.

The edition builder provides a simple revision control system for editions. If a user makes an edition "live" (akin to a commit), its configuration settings are frozen. We did not want to require that the user discard the current revision's in-memory object when making an edition live - therefore, we used the Veto enhancement pattern. We catch the VetoException instead and display a message to the user. Because the Veto pattern is implemented as an aspect that vetoes all field assignments (and does not trigger any observers), we can be certain that read-only revisions of an edition are not changed as a result of user action. Without the Veto pattern, we would have to disconnect all controllers from the model, which would have resulted in a perceptible impact on the usability of the interface.

5.4 Navigatable Structure.

When building an edition, we need to apply a number of consistency checks, beyond what can be expressed by applying constraints to individual elements in the schema describing the XML document. For instance, if a librarian uses a custom search field in a catalog that is part of the configuration file, she must have defined that field as a search option, which is stored elsewhere in the configuration file. These semantic checks involve a traversal of the XML document and the examination of certain elements and attributes. We used the Castor-specific hierarchical visitor to implement a number of consistency checks like this one, cleanly separating each concern.

5.5 Instant Cache.

The edition builder contains a search facility that allows edition maintainers to search for and browse through different editions. This browsing requires the retrieval of elements from those editions' configuration files, requiring them to be parsed. In addition, we need to examine an on-disk directory structure for the presence of revisions. Because the search facility appears on the start page of the edition builder application, we use the instant cache enhancement to cache the results of these expensive operations. The instant cache enhancement allowed us to reduce the code we have to maintain to a single method `createRecord(String id)` that maps an edition id to a cacheable edition record.

This case study demonstrates the usability of the enhancement patterns we have created by successfully enhancing the automatically generated data binding functionality used in an AJAX [19] application. Though we have requested that recursive observers, immutability, visitors, and various convenience methods be added to Castor, its developers have been hesitant to do so, possibly because of a lack of development resources. Our enhancement patterns allow us to express these cross-cutting concerns in a robust and targeted fashion nevertheless.

6. Related Work

Multiple prior research efforts share our goals of facilitating program enhancement either directly or indirectly. Our approach builds upon prior work on using *Aspect Oriented Programming (AOP)* [36, 37] to express GoF patterns [8, 10, 24, 25] and concurrency constructs [13] as aspects. Our approach makes it possible to divide the task of program enhancement between different parties (i.e., enhancement providers and enhancement consumers) to a much greater extent than prior efforts. Our approach accomplishes that by providing automated tools that enable the different parties to deal with individual enhancement concerns effectively. The use of generative aspects, for example, allows enhancement consumers to enjoy the benefits of using powerful aspect-oriented machinery, without getting into its lower-level implementation details.

The need to design systems that are easier to enhance has long been recognized as a staple of good software engineering [6, 22, 48, 50]. Our approach treats enhancements as a separate concern, enabling greater flexibility during both the design and implementation stages. Our work follows upon multiple prior efforts aimed at providing automated tools [23, 47] with the goals of mitigating the challenges of program restructuring and enhancement.

Automated program adaptation has been advocated as an approach that can facilitate program enhancement, particularly its component-based and code-based varieties. Specific examples include component adaptation techniques with adaptable component interfaces [26], delegation [38], binary adaptation at runtime [34], and architectures [44, 45, 51]. We assume that the base program is amenable to our enhancement approach and only verify this fact by using a rule base before applying enhancements. Combining automated program adaptation techniques with our approach presents an interesting future work direction.

Compile-time, class-based, meta-object programming systems such as OpenJava [58] enable structural reflection, an ability to alter the definition of data structures such as classes, methods, and fields. Structural reflection enhances programs by generating and incorporating new program elements. Compared to traditional structural reflection systems, our approach allows multiple enhancements to be applied to the same code entity, supports flexible composition of enhancements, and allows enhancements to be triggered using multiple types of observed facts about the code. Finally, our approach effectively separates enhancement roles: enhancement providers can implement enhancements even for not-yet-generated classes whose names are unknown.

Several novel software development paradigms such as *Adaptive Programming*, *Strategic Programming*, and *Intentional Programming* also aim at facilitating program enhancement among their other objectives. *Adaptive Programming* [28, 41] enables the expression and reasoning about object structure traversals as a higher-level strategy, separate from the core functionality of a program. As a specific implementation of the adaptive programming paradigm for Java, the DJ library [49] uses a dynamic form of the Visitor design pattern through the use of reflection. Our *Navigatable Structure* enhancement pattern is closely related to Adaptive Programming, even though our motivation for adding structural navigation abilities to an existing object structure is driven by the needs to enhance a pre-existing program that finds itself missing capabilities due to changed operational contexts or new requirements.

Strategic Programming provides support for a combinatorial style of traversal construction in several programming paradigms. In object-oriented incarnation, it uses generic visitor combinators, which extend the Visitor design pattern. While Strategic programming could be applied to enhancement, its existing applications tend to focus on semantics-preserving transformations such as refactoring and optimization. Strategic programming can be combined with AOP [31], and similarly to our approach, can benefit from being integrated with an open compiler [32].

Intentional Programming (IP) [54] enables programmers to work at a higher level of abstraction (i.e., intentions) by representing a program as a database. This enables greater flexibility when changing a programs, as any change can be kept consistent with the rest of the program by following links in the database. By contrast, our approach relies on programmer-provided enhancement libraries and a rule base rather than a database to control the application of enhancements.

Explicit Programming [7] allows developers to use new declaration modifiers, which refer to syntax transformer classes that exploit an extension of the reflection API to generate additional code. Our approach differs in not requiring changes to the source code, and thus being applicable to third-party libraries and automatically generated code.

The Arcum framework [53] follows an approach that bears similarities with our work. It extends the refactoring paradigm to provide a mechanism for managing crosscutting design idioms. While we share the goals of providing better tool support and greater flexibility to maintenance programmers, our application domains are different: Acrum focuses on refactoring, while our work on enhancement. From the implementation perspective, the Acrum framework uses a domain-specific language, while our approach uses an augmented aspect compiler and a rules engine.

Model-based pointcuts [33] facilitate the evolution of aspect-oriented programs by using a conceptual model of the base program to define the application of pointcuts. Our approach of using first-order logic rules to trigger enhancements bears similarity to model-based pointcuts. We would like to explore the issues of change impact with respect to enhancements as a future work direction.

Rule engines have previously been used for the construction of new aspect weavers for legacy programming languages [21]. Similarly, we could extend our use of rules to base the actual instantiation of an enhancement on rule-based patterns, rather than implementing enhancements as libraries.

7. Future Work

While the main conceptual building blocks of our approach are in place, the DRIVEL infrastructure is a work in progress. In our future work, we will focus our efforts on improving the power, expressiveness, and usability of our infrastructure.

More sophisticated static and dynamic analysis could support more complex and more sophisticated enhancements. Our existing enhancement infrastructure uses only a limited subset of the program analysis functionality of the AspectJ compiler. For example, when handling a "Missing Method" error, DRIVEL uses only the name, receiver type, expected return type (where known), and the scope of the missing method. Although this information is sufficient to synthesize correct code for the currently supported set of enhancement patterns, we could benefit from additional information, such as dataflow information that indicates the intended use of the return value of the method that is missing. Such more advanced analyses would enable DRIVEL to apply more complex enhancements.

As another example of enhancements benefiting from compile-time analysis, consider how the existing *Instant Cache* enhancement could be supplemented with automated cache invalidation capabilities. For instance, dataflow analysis can be used to calculate the input dependencies of a cached object. An aspect inserted at the pointcut of all modification join points could then trigger cache invalidation.

On the expressiveness and usability fronts, we plan to integrate DRIVEL with the Eclipse IDE [15]. We plan to integrate DRIVEL with the existing Intellisense capabilities of the Eclipse framework such that enhancement-provided methods are offered via auto-completion. In addition, an intuitive GUI could provide an interactive mode for creating user rules that guide the application of enhancements.

Finally, we plan to create a Web repository of enhancement patterns that will allow developers to contribute and share their own enhancement libraries. We hope that this repository will become a resource for software developers through which they could find solutions to their program enhancement problems. Statistics from the use of this repository will enable us to assess the benefits of our approach more realistically and to receive constructive feedback from the broader community of software developers.

8. Conclusion

We have presented a novel approach to pattern-oriented automated enhancement of object-oriented programs. We have demonstrated how by augmenting the capabilities of an aspect compiler, an automated tool can capture the programmer's intent to enhance a program. Through automatic synthesis of aspect code and the use of declarative *when-then* rules, our approach enables powerful automated enhancement of real programs, improving productivity and ensuring better quality of the resulting code base. We validated our approach by applying our automated tool to solve the challenges of enhancing automatically-generated code in a real-world AJAX application. Our initial results demonstrate that rule-based, integrated, automated program enhancement has the potential to become a standard part of the professional programmer's toolset.

Acknowledgments

The authors would like to thank the anonymous reviewers, whose useful comments helped improve the presentation of the paper. Cody Henthorne contributed to an earlier prototype of the DRIVEL system. This research was supported by the Department of Computer Science at Virginia Tech.

References

[1] BACK, G. Datascript - a specification and scripting language for binary data. In *Proceedings of the ACM Conference on Generative Programming and Component Engineering Proceedings (GPCE 2002), published as LNCS 2487* (Pittsburgh, PA, Oct. 2002), ACM, pp. 66–77.

[2] BAILEY, A., AND BACK, G. LibX–a Firefox extension for enhanced library access. *Library Hi Tech 24*, 2 (2006), 290–304.

[3] BASILI, V., AND BOEHM, B. COTS-based systems top 10 list. *Computer 34*, 5 (2001), 91–95.

[4] BOEHM, B., AND ABTS, C. COTS integration: Plug and pray? *Computer 32*, 1 (1999), 135–138.

[5] BOEHM, B. W. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, 1981.

[6] BOEHM, B. W. A spiral model of software development and enhancement. *Computer 21*, 5 (1988), 61–72.

[7] BRYANT, A., CATTON, A., VOLDER, K. D., AND MURPHY, G. C. Explicit programming. In *AOSD 2002* (New York, NY, USA, 2002), ACM, pp. 10–18.

[8] CACHO, N., SANT'ANNA, C., FIGUEIREDO, E., GARCIA, A., BATISTA, T., AND LUCENA, C. Composing design patterns: a scalability study of aspect-oriented programming. In *5th International Conference on Aspect-Oriented Software Development* (Bonn, Germany, 2006), ACM Press, pp. 109–121.

[9] CHEN, H., AND CHENG, R. *ZK: Ajax without the Javascript Framework*. aPress, Aug. 2007.

[10] CLARKE, S., AND WALKER, R. J. Composition patterns: an approach to designing reusable aspects. In *23rd International Conference on Software Engineering* (Toronto, 2001), IEEE Computer Society, pp. 5–14.

[11] CODEHAUS OPENSOURCE SOFTWARE COMMUNITY. The Castor project. `http://www.castor.org/index.html`.

[12] COWARD, D. Java Servlet Specification Version 2.4, 2004.

[13] CUNHA, C. A., SOBRAL, J. L., AND MONTEIRO, M. P. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In *5th International Conference on Aspect-Oriented Software Development* (Bonn, Germany, 2006), ACM Press, pp. 134–145.

[14] CZARNECKI, K., AND EISENECKER, U. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000.

[15] ECLIPSE FOUNDATION. Eclipse Java development tools. `http://www.eclipse.org/jdt`.

[16] EGYED, A., AND BALZER, R. Integrating cots software into systems through instrumentation and reasoning. *Automated Software Engineering 13*, 1 (2006), 41–64.

[17] FORGY, C. Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. *Artificial Intelligence 19*, 1 (1982), 17–37.

[18] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[19] GARRETT, J. J. Ajax: A new approach to web applications, 2005. `http://www.adaptivepath.com/publications/essays/archives/000385.php`.

[20] GOTH, G. XML: The Center of Attention Up and Down the Stack. *Distributed Systems Online, IEEE 7*, 1 (2006), 3–3.

[21] GRAY, J., AND ROYCHOUDHURY, S. A technique for constructing aspect weavers using a program transformation engine. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development* (New York, NY, USA, 2004), ACM Press, pp. 36–45.

[22] GRISWOLD, W. G. Just-in-time architecture: planning software in an uncertain world. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops* (New York, NY, USA, 1996), ACM Press, pp. 8–11.

[23] GRISWOLD, W. G., AND NOTKIN, D. Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Methodol. 2*, 3 (1993), 228–269.

[24] HAMMOUDA, I. A tool infrastructure for model-driven development using aspectual patterns. *Model-driven Software Development - Volume II of Research and Practice in Software Engineering* (2005), 139–178.

[25] HANNEMANN, J., AND KICZALES, G. Design pattern implementation in Java and AspectJ. In *17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Seattle, Washington, 2002), ACM Press, pp. 161–173.

[26] HEINEMAN, G. T. Adaptation and software architecture. In *3rd International Workshop on Software Architecture* (Orlando, Florida, 1998), ACM Press, pp. 61–64.

[27] HENTHORNE, C., AND TILEVICH, E. Code generation on steroids: Enhancing COTS code generators via generative aspects. In *Second International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques IWICS 2007* (2007).

[28] HÜRSCH, W. L., AND SEITER, L. M. Automating the evolution of object-oriented systems. In *International Symposium on Object Technologies for Advanced Software* (1996), Springer Verlag, Lecture Notes in Computer Science, pp. 2–21.

[29] JOHNSON, R. J2EE development frameworks. *Computer 38*, 1 (2005), 107–110.

[30] JOHNSON, R. E., AND FOOTE, B. Designing reusable classes. *Journal of Object-Oriented Programming 1*, 2 (1988), 22–35.

[31] KALLEBERG, K., AND VISSER, E. Combining Aspect Oriented and Strategic Programming. *Electronic Notes in Theoretical Computer Science 147* (2006), 5–30.

[32] KALLEBERG, K. T., AND VISSER, E. Fusing a transformation language with an open compiler. In *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA'07)* (Braga, Portugal, March 2007), A. Sloane and A. Johnstone, Eds., pp. 18–31.

[33] KELLENS, A., MENS, K., BRICHAU, J., AND GYBELS, K. Managing the evolution of aspect-oriented software with model-based pointcuts. In *ECOOP 2006 Object-Oriented Programming* (2006), vol. 4067, Springer Verlag, pp. 501–525.

[34] KELLER, R. Binary component adaptation. In *12th European Conference on Object-Oriented Programming* (1998), Springer-Verlag, pp. 307–329.

[35] KEMERER, C., AND SLAUGHTER, S. Determinants of software maintenance profiles: an empirical investigation. *Journal of Software Maintenance Research and Practice 9*, 4 (1997), 235–251.

[36] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An overview of AspectJ. In *ECOOP* (2001), Springer-Verlag.

[37] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J. M., AND IRWING, J. Aspect-oriented programming. In *ECOOP* (1997), Springer-Verlag.

[38] KNIESEL, G. Type-safe delegation for run-time component adaptation. In *13th European Conference on Object-Oriented Programming* (1999), Springer-Verlag, pp. 351–366.

[39] LEHMAN, M. M., AND BELADY, L. A., Eds. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[40] LIBX TEAM. LibX browser plugin for libraries. `http://libx.org`.

[41] LIEBERHERR, K. J. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[42] LIENTZ, B., AND SWANSON, E. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1980.

[43] MCLAUGHLIN, L. Automated programming: the next wave of developer power tools. *Software, IEEE 23*, 3 (2006), 91–93.

[44] MOREL, B., AND ALEXANDER, P. Automating component adaptation for reuse. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering* (2003), pp. 142–151.

[45] MOREL, B., AND ALEXANDER, P. SPARTACAS: automating component reuse and adaptation. *IEEE Transactions on Software Engineering 30*, 9 (2004), 587–600.

[46] MORISIO, M., SEAMAN, C. B., PARRA, A. T., BASILI, V. R., KRAFT, S. E., AND CONDON, S. E. Investigating and improving a cots-based software development. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering* (New York, NY, USA, 2000), ACM Press, pp. 32–41.

[47] NOTKIN, D., AND GRISWOLD, W. G. Extension and software development. In *Proceedings of the 10th International Conference on Software Engineering* (Singapore, 1988), IEEE Computer Society Press, pp. 274–283.

[48] OREIZY, P., MEDVIDOVIC, N., AND TAYLOR, R. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE)* (1998), pp. 177–186.

[49] ORLEANS, D., AND LIEBERHERR, K. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns* (2001), Springer.

[50] PARNAS, D. L. Designing software for ease of extension and contraction. In *3rd International Conference on Software Engineering* (Atlanta, Georgia, 1978), IEEE Press, pp. 264–277.

[51] PENIX, J., AND ALEXANDER, P. Toward automated component adaptation. In *Proceedings of the Ninth International Conference on Software Engineering and Knowledge Engineering* (1997), Knowledge Systems Institute, pp. 535–542.

[52] PROCTOR, M., NEALE, M., LIN, P., AND FRANDSEN, M. Drools Documentation. Tech. rep., JBoss Inc., 2006.

[53] SHONLE, M., GRISWOLD, W., AND LERNER, S. Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. In *Proceedings of the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)* (2007), ACM Press New York, NY, USA, pp. 175–184.

[54] SIMONYI, C. The Death of Computer Languages, the Birth of Intentional Programming. In *NATO Science Committee Conference* (1995).

[55] SMARAGDAKIS, Y., AND BATORY, D. Application generators. *Encyclopedia of Electrical and Electronics Engineering* (2000).

[56] SOARES, S., BORBA, P., AND LAUREANO, E. Distribution and Persistence as Aspects. *Software: Practice & Experience 36*, 6 (2006).

[57] SUTTER, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal 30*, 3 (2005).

[58] TATSUBORI, M., CHIBA, S., KILLIJIAN, M.-O., AND ITANO, K. OpenJava: A class-based macro system for Java. In *Reflection and Software Engineering*. Springer Verlag, 2000, pp. 117–133.

[59] WILDE, N., MATTHEWS, P., AND HUITT, R. Maintaining object-oriented software. *IEEE Software 10*, 1 (1993), 75–80.