Explicit Programming

Avi Bryant, Andrew Catton, Kris De Volder and Gail C. Murphy
Department of Computer Science
University of British Columbia
2366 Main Mall
Vancouver BC Canada V6T 1Z4
abryant,catton,kdvolder,murphy@cs.ubc.ca

ABSTRACT

Many design concepts can be expressed only indirectly in source code. When this occurs, a single concept at design results in a verbose amount of code that is scattered across the system structure. In this paper, we present explicit programming, an approach that enables a developer to introduce new vocabulary into the source to capture a design concept explicitly. An introduced vocabulary item modularizes the implementation details associated with a design concept, reducing the scattering of code needed to express the concept. The vocabulary item appears in the code where the concept is needed; uses of the vocabulary may thus remain distributed through the code. We believe explicit programming provides a useful engineering point, balancing modularization and separation in (at least) two cases. First, when a design concept is tightly coupled with particular constructs in a program, separation is unlikely to lead to any benefits of reusability or comprehensibility. Second, concepts that emerge as a system evolves can be encapsulated and recorded, paving the way for later separation when conditions warrant it. We introduce ELIDE, a tool that supports explicit programming in Java, and describe several cases showing the utility of the explicit programming approach.

1. INTRODUCTION

Many design concepts do not map directly into source code written in a general-purpose programming language. All too often, a single concept in the design requires verbose amounts of code to express. Sometimes this code is localized within a small part of the system's source; more often, it is scattered. Scattered, verbose code introduces at least two problems for software developers who must evolve the system. First, it is difficult for developers to understand the design of the system, leading over time to a degradation in the structure and conceptual integrity of the system. Second, even when a design concept is rediscovered, it is difficult for developers to change the concept consistently.

Aspect-oriented approaches attempt to address these problems by modularizing the scattered pieces of source code that implement a concept [15]. At one end of the spectrum of aspect-oriented approaches are language mechanisms, such as AspectJ [14, 2] and Hyper/J [18, 12], that allow the scattered code to be localized and separated. The developer must state how a separated unit composes with the rest of the system. At the other end of the spectrum are approaches, such as information transparency [11] and concern graphs [20], that allow the scattered code to be identified and reasoned about as a unit. The developer must still manipulate the scattered and verbose code.

In this paper, we propose an approach, called explicit programming, that allows a developer to introduce new vocabulary into the source. The definition of a vocabulary item modularizes the implementation details associated with representing the concept in general-purpose source code. The use of a vocabulary item in the source makes the design concept explicit. Explicit programming falls into the middle of the spectrum outlined above. Although the definition of a vocabulary item localizes previously scattered code, the uses of the item remain distributed through the code.

We believe explicit programming provides a useful engineering point balancing modularization and separation in (at least) two cases. First, when a design concept is tightly coupled with particular constructs in a program, separation may not improve the reusability or comprehensibility of the code. Second, concepts that emerge as a system evolves can be encapsulated and recorded, paving the way for later separation when conditions warrant the separation. We expand on these points in Section 2.

To investigate this approach, we have built a tool, called ELIDE, to support explicit programming in Java^{TM}. We outline explicit programming in Section 3, we describe the ELIDE tool in Section 4, and we present several uses of the tool in Section 5. In Section 6, we discuss the relationship between our approach and aspect-oriented programming. We compare explicit programming to related approaches, including macros and domain-specific languages in Section 7, before summarizing the paper in Section 8.

Our paper makes three contributions.

1. We identify and describe two situations in which it is helpful to modularize a scattered concern without fully separating the concern from the rest of the system.

2. We characterize the explicit programming approach, which provides a useful set of engineering tradeoffs for supporting these two situations.

3. We introduce the ELIDE tool to support explicit programming in Java.

2. TO SEPARATE OR NOT?

The concept of separating concerns is fundamental in software development. Although the separation of a concern can bring many benefits, such as making it easier to understand a concern or to reuse a component, separation can also entail costs, such as making it more difficult to understand the behavior of the system as a whole. In this section, we discuss two situations in which there are benefits to modularizing, but not fully separating, a concern.

2.1 Coupled Concepts

Some design concepts are strongly coupled to programming language constructs. An example of such a concept is the design-by-contract (DBC) style of programming [16] that advocates the specification of run-time checkable method pre-conditions, method post-conditions, and class invariants.

Since Java does not directly support design-by-contract concepts, programming in this style requires the checks of the specifications to be encoded as assertion tests. This approach litters the code with similar assertion tests. The management of these similarities can be eased by capturing the coding idiom in an aspect, perhaps even separating the aspect from the main class code. But, since individual conditions and invariants are tightly coupled to the classes and methods to which they are applied, the separation of the checks may not be beneficial; for instance, when a method is changed, its pre- and post-conditions must be re-examined. Moreover, code is less likely to be shared between the tests by separating the assertion testing code.

Specific tools, such as iContract [13], capture preconditions, postconditions, and class invariants *explicitly* in the code by embedding the checks in stylized comments. These stylized comments localize and hide important details about the design-by-contract concept, such as the precise points at which the checks must be made. The uses of these statements remain scattered, appearing relatively close to the constructs to which they are coupled. One benefit of retaining these scattered uses is that a developer reading the code can use the statements as documentation about the class.

Other common design concepts also exhibit this close coupling to language constructs. For example, test code is strongly coupled to the code it tests. We expand on this example in the context of JUnit [9] in Section 5.1. The concept of properties in the JavaBean^{TM} standard, such as the vetoable property, are strongly associated with a particular field (or fields) of the Bean class. A more general solution than creating a specific tool for each such design concept is desirable.

2.2 Concept Evolution

Some useful design concepts emerge only as a software system evolves. For example, a designer decides that some objects need to be shared, as in the flyweight design pattern [10], to overcome performance problems. If such sharing is introduced at different times in multiple parts of the system, it may be difficult to separate the design concept of sharing until the regularities between the different uses are understood. In fact, the regularities may only become apparent through deliberate restructuring and refactoring. An approach that allows the capture of such a concept *in place* may make it easier for a developer to proceed in smaller steps towards further modularization. Some concepts captured this way may evolve towards being fully separated; for instance, they may be expressed as an aspect in AspectJ. Others may be better left in place because they may have strong local coupling. In Section 5.2 we present a case study of a small object-oriented system in which it was helpful to incrementally evolve towards the more explicit representation and modularization of concepts.

3. EXPLICIT PROGRAMMING

We use the term *explicit programming* to describe approaches that support the modularization, without separation, of a design concept. Using explicit programming, a developer can make design concepts explicit in their source code by *incrementally* introducing new *vocabulary* that can be used *where the concept occurs* in the code, and whose definition *modularizes* the implementation details of the concept.

By modularization, we mean that the mechanism must localize code associated with the implementation of the design concept. This localization is intended to help clarify the code and to ease the evolution of the concept. Simple naming conventions used to mark otherwise indistinguishable language constructs, such as the use of special names for classes or methods, are not sufficient to be considered a form of of explicit programming because they do not meet this last criterion.

4. ELIDE

To provide a cost-effective approach for making non-local design concepts explicit, we have developed the ELIDE tool that supports explicit programming in Java. This section describes the ELIDE tool. A complete example of using ELIDE appears in the Appendix.

4.1 Overview

ELIDE provides a single syntactic extension to the Java language: it permits a developer to introduce new modifiers for classes, fields, and methods. This approach balances the need to express a wide range of design concepts with familiarity: ELIDE modifiers are used identically to the standard Java modifiers such as public, private, and synchronized, and can be freely intermixed with them.

Each ELIDE modifier triggers a transformation on the source code. The transformation may add new classes, insert fields, modify existing methods, or generate new ones. As a simple example, consider marking a particular field as a JavaBeans property.

```
private property<> String name;
```

The transformation associated with this ELIDE modifier would generate the required mutator and accessor methods needed for the field to be a JavaBeans property.

ELIDE modifiers must be followed by a list of (possibly empty) parameters enclosed in angle brackets.[1] The

[1]Our syntax will likely not be compatible with Java 1.5. We are contemplating a change in syntax to be compatible. A simple transformation tool will suffice for moving from the existing syntax to the new syntax we are investigating.

presence of the parameter list, even if empty, helps distinguish ELIDE modifiers from ordinary Java modifiers for both parsers and programmers. The ability to specify parameters adds flexibility to the introduced vocabulary. For example, the `property<>` modifier might take an optional `read_only` parameter, that causes the mutator method to be omitted.

```
private property<read_only> String name;
```

A single modifier can take a varying number of parameters; these can be symbols, type names, string literals, or even code snippets. To make it easier to include verbatim Java code, the snippets are enclosed in an extended string literal syntax. Extended string literals are delimited by `%{` and `}%` and can span multiple lines of text. Additionally, inside the literal, the delimiters `#{}#` can be used to provide gaps in which arbitrary Java expressions can be evaluated and inserted. For example, the code below shows how an ELIDE modifier can be used to state a design-by-contract class invariant that ensures the number of elements in the set, indicated by `count` is never less than zero.

```
public invariant< %{count >= 0}% > class Set
{
        ...
}
```

4.2 Defining the semantics

The transformational semantics of ELIDE modifiers are specified by implementing specific Java classes. We chose this approach to make the tool accessible: Developers do not need to learn a special transformation language. This approach also allows the tool to be applied to itself. Several built-in ELIDE modifiers are used to ease the implementation of other modifiers. Further details are provided in Section 4.4.

Each introduced modifier corresponds to a subclass of `Transform`, the base class for all ELIDE extensions. The implementation of the modifier consists of one or more atomic transformations, each of which is represented by a separate method in the class. For example, the `property<>` modifier adds both a mutator and an accessor method. To define this property, we could introduce two separate atomic transformations, `addMutator` and `addAccessor`, as shown below.

```
public class Property extends Transform
{
     public satisfies<methodDefinition>
     void addMutator(FieldNode target) {...}

     public satisfies<methodDefinition>
     void addAccessor(FieldNode target) {...}
}
```

An atomic transformation can perform many actions on the source. We use the term atomic because we treat such a transformation as an indivisible operation when stating and resolving dependencies (Section 4.4). A method is designated as being an atomic transformation method by the `satisfies<>` and `requires<>` modifiers. A class defining a modifier may have additional private helper methods.

The first argument to each of these methods is an object that represents the target—the entity in the source code to which the modifier is attached—of the transformation. In the example above, the target is a field: The argument of type `FieldNode` is the node in the abstract syntax tree representing the declaration of the field. The remaining arguments, if any, correspond to parameters that appeared within the angle brackets of the modifier.

4.3 The ELIDE API

The body of an atomic transformation describes the effect that a portion of the modifier has on the structure and behavior of the system. To ease the definition of these transformations, the tool provides an API. This ELIDE API provides classes and methods for inspecting and modifying the parse tree of the program.

To make the API intuitive and familiar to Java programmers, the core of this API is modelled after the java.lang.reflect API, with accessor methods, such as `getDeclaringClass()`, and `getDeclaredFields()`, amongst others. This introspective API is complemented with basic mutator functionality, including `add(someNode)` and `makePublic()`.

The basic API enables simple transformations to be implemented easily. For example, the following transformation method makes all fields of its target class private.

```
public void mkAllFieldsPrivate(ClassNode target)
{
       FieldNode[] fields = target.getDeclaredFields();
       for(int i = 0; i < fields.length; i++)
          fields[i].makePrivate();
}
```

To ease the definition of more complex transformation methods, the API offers a number of methods that accept (and parse) strings containing Java code. These methods are intended to be used with ELIDE's extended string literal syntax to pass in blocks of code. This mechanism eases the assembly of larger chunks of code. For example, a method to add an accessor for a field can be written as follows.

```
public void addAccessor(FieldNode target)
{
         String uppercaseName =
             Utils.methodCase(target.getName());
         target.getDeclaringClass().extend(
         %{
             public #{target.getType()}#
             get#{uppercaseName}#()
             {
                 return #{target.getName()}#;
             }
         }%);
}
```

Support for API methods to accept code from strings is beneficial for two reasons. First, this mechanism eases the writing of more complex transforms. Second, the mechanism enhances readability of the transforms because the implementation of the transform method *visually* resembles the code that is produced.

4.4 Managing Transformation Dependencies

To meet the goals of explicit programming, ELIDE supports the incremental introduction of modifiers into a system. As modifiers are introduced, interactions between modifiers are likely to occur. Consider adding a DBC-like `invariant<>` modifier, which adds assertions at the beginning and end of each public method, to a class that makes use of `property<>`. If the `property<>` transformations occur first, then invariant<> will find the newly introduced accessor methods and will add the proper assertions to them. However, if invariant<> occurs first, then any methods introduced by property<> will not contain the assertions.

Since many interactions between different modifiers manifest themselves as constraints on the transformation ordering, ELIDE provides a simple dependency system for managing the transformation order. Each transformation method is marked with a user-defined list of dependencies that it satisfies, and a list that it requires. These lists define a partial order on the transformations, ensuring that a transformation satisfying a dependency is always executed before any transformation requiring it. For instance, using the built-in modifiers, satisfies<> and requires<>, the example given above could be disambiguated as follows.

```
//in Invariant
requires<methodDefinition>
public addAssertions() { ... }

//in Property
satisfies<methodDefinition>
public addAccessorMethod() { ... }
```

These modifiers and their parameters ensure that the addAccessorMethod(), as well as any other transformation method marked as satisfying a methodDefinition, executes before addAssertions().

ELIDE will check for circular dependencies before executing any transformations. If it detects a deadlock, it will report an error to the user. The user must alter the dependencies to resolve the deadlock before the transformations can be applied.

4.5 Implementation

ELIDE is implemented in Java as a Java two-pass preprocessor. A typical run cycle of ELIDE involves the following five steps:

- process modifier definitions with ELIDE,
- compile the resulting source,
- load the compiled modifiers into ELIDE,
- process the target source files, and
- compile the output source files.

ELIDE outputs code that looks as much as possible like the code a developer would expect if they were not using the tool. This transparency helps minimize the conceptual distance between the versions of a system before and after capturing a concept.

ELIDE's speed is limited primarily by the parser, which is generated by the SableCC [8] system. A typical first run of ELIDE over 17000 kLOC takes approximately 21 seconds on a 1GHZ PC. To improve these times, the internal representation of source files is serialized to disk as they are parsed, and, when possible, this cached information is then used on subsequent runs. With the majority of files cached, the runtime decreases to approximately 9 seconds.

An issue with the current version of ELIDE is that separate compilation is practically impossible because we chose not to limit the scope of transforms. This choice was deliberate because we wanted to investigate the capture of scattered concepts. To date, most of the modifiers we have defined have had localized effects, affecting only the code in their surrounding classes. The few exceptions are modifiers used to help write tests in JUnit (Section 5.1), and those used to implement some design patterns (i.e. Visitor and Flyweight [10]). Although these modifiers affect more than one class, they do so in a predictable way. Based on this experience, we intend to investigate an approach in which modifiers have explicitly declared scope, enabling separate compilation without impeding usability.

5. USING ELIDE

We have used ELIDE on several systems to capture different kinds of design concepts. We describe two of these uses in detail: the use of ELIDE to support unit testing with JUnit, and the use of ELIDE to capture design concepts in an evolving visualization system. These uses demonstrate the useful range of design concepts that can be captured in a cost-effective way.

5.1 Using the JUnit Framework

We used ELIDE to simplify the writing of tests using JUnit [9]. JUnit is structured around TestCase classes. Typically there is one TestCase class per Java class being tested, each such class has a number of test methods that makes various assertions about the tested class.

Adding JUnit tests to a system involves a fair amount of overhead. Each TestCase class must follow a number of conventions; for example, the class must provide a constructor with a particular form. Each test that will be run requires a main() method that invokes JUnit's TestRunner, and running more than one test at once requires a TestSuite class to be created that lists the TestCases for a particular package.

To automate and standardize this process, we created a test modifier. The test modifier can be attached to blocks of code inside a class body that contain testing code.

```
class SomeClass
{
        test<%{ ... }%>
        void method1() { ... }

        test<%{ ... }%>
        void method2() { ... }
}
```

The transformation defined for this modifier extracts the test code into methods of a newly created TestCase class, and registers that class with a TestSuite for the package. There are two benefits to using this modifier. First, a developer may be more likely to keep tests up-to-date when a separate set of test classes need not be maintained. Second, the test code, when kept close to the method being tested, may provide additional documentation about the method.

5.2 The AVID Visualization System

To understand whether ELIDE could capture useful design concepts in an existing system, we applied ELIDE to the AVID Java tool developed at the University of British Columbia.

AVID Java supports the visualization of the dynamic execution of object-oriented Java-based systems in terms of user-defined architectural entities [25]. The version of the system with which we started consisted of approximately 17,000 lines of commented Java code or 7000 non-commenting source statements (NCSSs). We focused on a subset of the system that supported the manipulation of the execution trace. This subset consisted of approximately 4200 lines or 1100 NCSSs. The bulk of this code defined different events that might appear in a trace, such as `InstanceMethodEntryEvent`. These classes formed a single complex inheritance hierarchy (24 classes and depth 7). In addition, there were various reader and writer classes that were used to provide persistence to the trace data types.

We approached the task of increasing the design content of the code incrementally. Small changes and brief analysis lead to new understanding which, in turn, led to further changes. The developer performing this task was familiar with both AVID and ELIDE.

5.2.1 Refactoring and Design Rediscovery

Initially, the developer focused on the use of ELIDE to remove straightforward redundancies and boilerplate code. This analysis resulted in the definition of three class-level ELIDE modifiers: `type<>`, `equals<>`, and `allAccessors<>`. The `type<>` modifier generates appropriate factory classes for all classes to which it was attached. The `equals<>` modifier generates appropriate equals behavior given the fields on a class. The `allAccessors<>` modifier generates accessors for all fields on a class.

After the addition of these modifiers, the code output of ELIDE was the same as the original code, but it was easier to modify and understand. For example, the fact that all classes used all fields for equality testing was explicit in the definition of the `equals<>` modifier, not something that would have to be recovered by examining all of the classes `equals()` methods at a later point.

5.2.2 Persistence and Efficiency

The original code contained reader and writer classes to access the trace files. An examination of these readers and writers revealed that they were tightly coupled to the structure of the trace data types. This coupling was needed to support efficient encoding of the data types. The intent of the reader and writer classes was clear: The classes encapsulated persistence support. The problem was that the data type encapsulation was broken in the process.

The developer introduced ELIDE modifiers to support a better factorization of the program: Each class became responsible for its own persistence, but the similarities of persistence that crosscut those classes were captured in ELIDE modifiers. Specifically, we defined `read<>` and `write<>` modifiers to introspect on trace data type classes and to generate the appropriate `read()` and `write()` methods to support persistence. The reader and writer classes were essentially simplified to iterators that called these methods. We decided not to remove these classes outright in order to maintain the current interface to the rest of the system.

5.2.3 The Flyweight Pattern

The original system employed a version of the Flyweight design pattern [10] to save space at run-time. The use of this pattern was mentioned in the design document. However, its implementation was non-obvious. The developer added a `flyweight<>` modifier to capture the particular implementation of flyweights used. This modifier made the `type<>` modifier added earlier obsolete: It had referred to characteristics of the entire hierarchy collectively, but the addition of the `flyweight<>` modifier necessitated a division of the hierarchy into non-flyweight and flyweight types. Since introducing `type<>` was not costly, it could be disposed of without hesitation. The full implementation of a version of `flyweight<>` similar to the one used in AVID can be found in the Appendix.

5.2.4 Summary

Quantitatively, applying ELIDE to the tracing portion of AVID reduced the original 1100 NCSSs to 300 NCSSs of AVID code and about 250 NCSSs of modifier definition code. Much of this reduction is due to the replacement of boilerplate code in trace data type classes. For example, the ELIDE output for the following ELIDEd class is about 100 lines of Java code.

```
read<>
write<>
equals<>
flyweight<>
allAccessors<>
public abstract class EventEncoding
extends Encoding
{
         private ThreadID threadID;
         private ClassID classID;
}
```

This example shows that ELIDing the AVID code helped make design concepts explicit in the code. At a glance, a developer can see that this class provides persistence, has accessors for all fields, supports equals, and is used as a flyweight. Just as important is what the developer does not see: Lines and lines of boilerplate code that might obscure the only thing differentiating this encoding class from others, the two ID fields. In addition to being more comprehensible, the system is more modifiable and extensible. A change in the way persistence is handled can be achieved through a change to the definition of the persistence modifiers. Adding a new trace data class is also much easier, requiring only a dozen of lines of code instead of about a hundred. Since ELIDE produces (readable, commented) Java code, a client of the EventEncoding class can still get access to the class' full interface description.

6. DISCUSSION

Modifiers in ELIDE play a similar role to pointcut declarations in AspectJ: each is used to designate a set of joinpoints. There are two differences between these constructs.

One difference is the target of the constructs. In AspectJ, joinpoints are dynamic execution points. In ELIDE, joinpoints are points in the program text.

A second difference is the placement and form of the constructs. In AspectJ, pointcuts designate sets of joinpoints from a place conceptually *outside* of the affected (part of the) program by means of expressions that *match* joinpoints. In contrast, ELIDE modifiers are placed *in* the affected program and are *attached* to joinpoints, or to regions of code where the joinpoints are to be found. For example, a modifier attached to a class can implicitly designate all of the class' methods to be joinpoints.

Since the intent in ELIDE is to modularize a crosscutting concept without separating the concept, associating the modifiers directly with the joinpoints is reasonable. However, in the course of making these direct associations, broader patterns may become clear. For example, in the

AVID case study, we introduced two modifiers, `read<>` and `write<>`, to make explicit similarities in the persistence of event classes. These modifiers were used in a regular way within a group of classes: They were applied to each class within a particular package. This regular usage raises the question of whether the application of these modifiers should be placed into a separate crosscutting transform. For such cases, it might be desirable to introduce a method of specifying joinpoints through patterns, more akin to the joinpoint model of AspectJ.

The code below specifies such a crosscutting transform where '`@`' is used as a wildcard to apply the modifiers to all classes in the encodings package.

```
package encodings;

read<>
write<>
public class @ {}
```

An experimental extension to ELIDE permits such specifications, as well as more complicated forms. The following code, for example, specifies that the `log<>` modifier should be applied to all public methods of all public classes.

```
public class @ {
    log<> public void @();
}
```

A disadvantage of such transforms is that they reduce the explicitness of the source with respect to particular encodings. On the other hand, these transforms make it more explicit that all classes in the package provide persistence, or that all public methods are logged. A developer using ELIDE can choose whether or not to take this step of separation. The developer applying ELIDE to AVID never did perform the separation. The knowledge that a clean path to separation exists, provided the developer the confidence to postpone the actual decision until the benefits become clear.

7. RELATED WORK

Explicit programming and ELIDE are intended to increase the amount and kinds of conceptual information that can be captured in source code. Broadly, there are three approaches that have been taken to this problem: domain-specific languages, macros, and static meta-programming. In this section, we compare explicit programming and ELIDE to representative work in these areas, and we compare the role of ELIDE modifiers to UML stereotypes in a design-driven code generation approach.

7.1 Domain-Specific Languages

The development of a language to express the concepts of a specific domain can yield many advantages. SQL, for example, is a domain-specific language for querying a database; in a few lines, a query may be stated that might take a screenful or two otherwise. Creating a domain-specific language typically requires a substantial investment: a domain analysis must be performed, a language must be designed, and tools to support the language must be created. To lessen the cost of providing a domain-specific language, numerous generators and toolkits have been developed [23]. For example, the Jakarta Tool Suite (JTS) [5] supports the extension of Java with new language constructs.

Explicit programming differs from domain-specific languages in two ways. First, explicit programming helps a programmer exploit immediate opportunities to improve a code base by refactoring and capturing design concepts that emerge from, and that are typically tied closely to, the current code base. In contrast, domain-specific languages seek to encode broadly applicable and reusable design concepts. Second, an explicit programming tool is intended to be widely accessible to programmers: A programmer need not have any expertise in language implementation. In contrast, domain-specific language tools, such as JTS, require a developer to understand language parsing to be able to define grammar extensions.

7.2 Macros

Macro systems provide a more economical, but also more limited, means of extending the vocabulary a developer may use in expressing a system than the typical domain-specific language tool. Macros are less flexible in the changes they can introduce into a language, and macro systems do not typically provide any support for handling errors or debugging in terms of the introduced vocabulary. ELIDE makes similar tradeoffs and is thus closely related to macro systems.

The Lisp macro system [19] is the prototypical ancestor of most macro systems in use today. A lisp macro is implemented as a procedure: The procedure accesses the macro's arguments in the form of Lisp lists and computes a suitable substitution syntax for the macro. This mechanism is most suitable to define very local program transformations. Although it is theoretically possible to wrap a macro around a large chunk of Lisp code and to have that macro perform arbitrary transformations on that code, the programmer would need to implement syntax analysis as list manipulation, making it difficult to define the transformation. Wide-ranged transformations are also hard to compose because Lisp provides no mechanism for managing interactions between different macros. ELIDE overcomes these impediments by providing the developer with a suitable interface to define a transformation, and by providing a dependency mechanism to guide the application of interdependent transformations.

More recently, hygienic variants of the Lisp macro system have been developed [6]. These systems improve on earlier macro systems by ensuring names in the code are not accidentally captured. Our current implementation of ELIDE does not support hygienic transformations. In the long run, hygiene may become an important issue for explicit programming, but a simple ad-hoc solution using developer-generated symbols was sufficient for initial exploration.

The Java Syntax Extender (JSE) [3] and Maya [4] are two macro packages for Java. JSE is a hygienic, infix version of Lisp macros for Java. JSE is oriented at defining non-interacting local transformations that require little or no structural information, whereas ELIDE is designed to express interacting non-local structural-based transformations. Maya includes a multi-method-like dispatching mechanism to determine which macro expansion to apply, support for extending the Java grammar, and the ability to redefine existing language elements using macro expansions. This greater flexibility requires more expertise from the user. For example the user has to understand LALR parsing technology and must manage the laziness of Maya's mixed parsing and type-checking algorithm. ELIDE tries to provide less power beyond standard macro expansion packages with the intent of remaining more accessible to programmers.

7.3 Static Meta-Programming

A number of systems are similar to ELIDE in that they are also based on compile-time meta programs that reason about program structure, and that transform it by means of a meta-programming API.

Within this spectrum of compile-time meta programming systems, intentional programming [21, 1] is most similar in motivation to explicit programming. The intentional programming system is an extensible programming environment that defines a programming language as a set of intentions. A developer can add domain-specific language features, including design concepts, by defining new intentions. Although ELIDE and intentional programming both define domain-specific features through transformations on an abstract syntax tree, they differ in the trade-offs made between expressiveness and ease of definition. In intentional programming, complete programming language are implemented as a set of intentions, whereas ELIDE only supports simple modifier definitions for Java. Consequently, an intentional abstract syntax tree is modelled at a fine level of detail, whereas in ELIDE it is coarse-grained. Also since large numbers of intentions are active in a system, given that a whole language is described by intentions, a number of transformation ordering mechanisms are provided, including *guilds* for organizing global transformations, identification of *independent subtrees* for managing locality and *questions* which offer more finer control of transformation order. It has been sufficient in ELIDE to have a simple dependency mechanism based on user-defined categories of transformation effects. Overall, then, a developer using ELIDE can capture less domain-specific features, but the cost of introducing a feature is also lower.

OpenJava is a static meta-programming system that is based on a compile-time, class-based, meta-object protocol [22]. Its compile-time MOP enables OpenJava to address the lack of structural and contextual information in Lisp macros. This compile-time MOP is similar to ELIDE's API in that it essentially provides a `java.lang.reflect` introspection-like API with additional mutation operations. OpenJava meta-classes also share similarities to ELIDE's class-level modifiers. However, the combination of meta-classes is restricted: Two OpenJava meta-classes cannot be applied to the same class. When we applied ELIDE to AVID, we found it useful to have more than one class modifier associated with a single class.

As another example, the MAGIK system is an open C compiler that can be extended by writing static meta-programs [7]. A developer expresses a desired transformation using an interface to the compiler's intermediate code representation. In contrast to explicit programming and ELIDE where the goal of a developer is to capture and elide design concepts using vocabulary, MAGIK meta-programs are oriented at implementing compiler optimizations and to enhancing error checking.

7.4 Design Pattern Capture

Generic Pattern Implementations (GPI), in which C++ templates are used to generate implementations of design patterns [24], share a similar intent as explicit programming. GPI enables a developer to label the location of a particular design concept in the code where the concept is used. For example, a developer can indicate where the Abstract Factory pattern [10] is used in the code; the code to configure a particular concrete instance of the pattern can then be automatically generated. Describing the code to be generated in the GPI approach requires sophisticated knowledge of the C++ template mechanism. We believe explicit programming, and in particular ELIDE, is more accessible to an individual developer because of the emphasis we have placed on simplifying the definition of a vocabulary item.

7.5 Design-based Generation

Our use of modifiers in ELIDE to tag elements of the source as participating in a particular design concept is similar to a suggested use of UML stereotypes in the OMG Model Driven Architecture (MDA) [17]. In the context of the MDA, which enables an implementation-independent way to describe distributed applications, stereotypes are used as a means of describing the role of a UML model entity. For example, an `Account` class may be tagged as a `BusinessEntity` or it may be tagged as a `CORBAInterface` depending on its role in the design. Code generators can then be built to take advantage of this stereotype information when refining a design into code.

8. SUMMARY

Explicit programming enables a developer to cost-effectively modularize, but not fully separate, useful design concepts. In explicit programming, a developer is able to incrementally make design concepts explicit in their source code by introducing new vocabulary. The definition of a vocabulary item modularizes the implementation details of the concept, but uses of that vocabulary can remain scattered in the code where the concept occurs.

In this paper, we have characterized situations where explicit programming is beneficial, we have described the ELIDE tool that supports explicit programming in Java, and we have described various uses of the tool, including the incremental ELIDing of a non-trivial system. These uses demonstrate the flexibility of ELIDE.

Explicit programming and ELIDE complement existing technologies. ELIDE can pick up where macros leave off, providing support for making crosscutting concepts explicit. With ELIDE, such concepts can be captured at lower-cost than with a domain-specific language. In some cases, capturing a concept with ELIDE could potentially pave the way for later separation of the concept with an aspect-oriented programming language.

Acknowledgements

This work was supported in part by NSERC, in part by the Canadian Consortium for Software Engineering research, and in part by the University of British Columbia. We thank the anonymous reviewers for insightful comments that improved the paper. Thanks also to Yvonne Coady, Brian de Alwis, Gregor Kiczales and Hidehiko Masuhara for insightful comments on drafts of this paper.

9. REFERENCES

[1] William Aitken, Brian Dickens, Paul Kwiatkowski, Oege de Moor, David Richter, and Charles Simonyi.

Transformation in intentional programming. In *Proc. of the First International Conference on Software Reuse*, pages 114–123. IEEE, 1998.

[2] AspectJ web page. http://www.aspectj.org.

[3] Jonathan R. Bachrach and Keith Playford. The Java syntactic extender. In *Proc. of ACM Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 31–42. ACM, 2001.

[4] Jason Baker and Wilson Hsieh. Maya: Multiple-dispatch syntax extension in Java. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2002. To appear.

[5] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for implementing domain-specific languages. In *5th International Conference on Software Reuse*, pages 143–153. IEEE Computer Society, 1998.

[6] William Clinger. Macros that work. In *Proc. of the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 155–162. ACM, 1991.

[7] Dawson R. Engler. Incorporating application semantics and control into compilation. In *Proc. of the First Conference on Domain-Specific Languages.* Usenix, 1987.

[8] Etienne Gagnon and Laurie J. Hendren. SableCC, an object-oriented compiler framework. In *Proceedings of the 26th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS USA 98)*, pages 140–154. IEEE, 1998.

[9] Erich Gamma and Kent Beck. Test-infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, 1995.

[11] William G. Griswold. Coping with crosscutting software changes using information transparency. In *Proc. of Reflection 2001: The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 250–265. Springer Verlag, 2001.

[12] Hyper/J web page. http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm.

[13] iContract web page. http://www.reliable-systems.com/tools/iContract/iContract.htm.

[14] Gregor Kiczales, Erik Hilsdale, Jim Huginim, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer Verlag, 2001.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Verlag, 1997.

[16] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[17] OMG. Model driven architecture (mda). ormsc/01-07-01.

[18] Harold Ossher and Peri Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *Proc of the 2000 International Conference on Software Engineering (ICSE)*, pages 734–737. ACM, 2000.

[19] Kent M. Pitman. The revised MacLisp manual. Technical Report MIT-LCS-TR-295, M.I.T. Artificial Intelligence Lab, 1983.

[20] Martin P. Robillard and Gail C. Murphy. Concern Graphs: Finding and describing concerns using structural program dependencies. In *Proc. of International Conference on Software Engineering*, 2002. To appear.

[21] Charles Simonyi. The death of computer languages, the birth of intentional programming. Technical Report MSR-TR-95-52, Microsoft Research, 1995.

[22] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano. OpenJava: A class-based macro system for Java. In *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 117–133. Springer Verlag, 2000.

[23] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[24] John Vlissides and Andrei Alexandrescu. To code or not to code, part i and ii. C++ Report, March and June 2000.

[25] Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Issak. Visualizing dynamic software system information through high-level models. In *Proc. of ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 271–283. ACM, 1998.

APPENDIX

This appendix presents a complete ELIDE example, showing a simple implementation of the flyweight design pattern via the `flyweight<>` modifier. A similar implementation was used during the AVID study described in Section 5.2. The main differences between the definition shown below and that used for AVID involve details specific to retrofitting legacy code. For example, the existing AVID code base used a more complex set of getter methods. Since ELIDE modifier semantics are by design disposable and flexible, it was easier to have `flyweight<>` generate code consistent with the existing system than to change the usage. There are many possible implementations for any pattern; for the purpose of providing a comprehensive ELIDE example, the AVID-specific details would simply be a distraction and are thus omitted. Note that we also ignore the abstract base classes often present in treatments of flyweight [10].

We begin by presenting the *use* of the `flyweight<>` modifier. Our implementation here expects the name of the corresponding factory class, as well as the type and name of the key used to identify the flyweight via the factory as parameters.

```
/* in regular code base */
flyweight<CharFactory, Character, charRep>
public class Char
{
     private Character charRep;
}
```

Using the modifier as shown above will generate the following code, implementing our version of the flyweight pattern. If the factory class does not already exist in the current package, it is created. Then a get method is added to it. A constructor is added to the flyweight class, parameterized with the key.

```
/* in file Char.java */
public  class Char extends Object
{
     private  Character charRep;

     public  Char(Character charRep)
     {
        this.charRep = charRep;
     }
}

/* in file CharFactory.java */
import java.util.*;

public class CharFactory extends Object
{
     private Map flyweights = new HashMap();

     public Char getChar(Character charRep)
     {
        Char result = (Char)flyweights.get(charRep);
        if(result == null)
        {
           result = new Char(charRep);
           flyweights.put(charRep, result);
        }
        return result;
     }
}
```

The Flyweight class provides the definition of the modifier. The addGet transformation method adds the getter method (getChar above) to the factory. addConstructor adds the required parameterized constructor to the flyweight class. getFactoryClass, a helper method for addGet, creates the factory class if necessary, or returns the existing one. Note that the parameters to the transformation methods correspond to the parameters passed to the modifier.

```
import ca.ubc.cs.elide.nodes.*;
import ca.ubc.cs.elide.*;
import java.util.*;

public class Flyweight extends Transform
{
satisfies<methodDefinition>
public void addGet(
      ClassNode target,
      String factoryName,
      String keyType,
      String keyName)
{
  // retrieve or create the
  // factory class node
  ClassNode factory =
      getFactoryClass(target, factoryName);
    String targetName = target.getShortName();
    // add getter to the factory class using
   // the templating system
   factory.extend(
   %{
      public #{targetName}#
      get#{targetName}#(#{keyType}# #{keyName}#)
      {
         #{targetName}# result =
           (#{targetName}#)flyweights
              .get(#{keyName}#);
         if( result == null )
         {
            result = new #{targetName}#
                           (#{keyName}#);
            flyweights.put(#{keyName}#, result);
         }
         return result;
      }
   }%);
}

satisfies<methodDefinition>
public void addConstructor(ClassNode target,
                           String factoryName,
                           String keyType,
                           String keyName)
{
   String targetName = target.getShortName();
   // add constructor to the flyweight class
   // using the templating system
   target.extend(
   %{
      public #{targetName}#(
            #{keyType}# #{keyName}#)
      {
         this.#{keyName}# = #{keyName}#;
      }
   }%);
}

private ClassNode getFactoryClass(
     ClassNode target,
     String factoryName)
{
   PackageNode parentPackage =
      target.getPackage();
   ClassNode factory =
      parentPackage.getClass(factoryName);

   // check if factory class already existed
   if( factory == null )
   {
      // if factory didn't exist, create one
      factory = new ClassNode(factoryName);
      // give class public access
      factory.makePublic();
      // add a required import statement
      factory.add(new ImportNode("java.util.*"));
      // add class to same package
      // as flyweight
      parentPackage.add(factory);
   }

   // fields can also be added to classes
   // using extend()
   factory.extend(
   %{
      private Map flyweights = new HashMap();
   }%);

   return factory;
}
}
```