Explicit Programming: Improving the Design Vocabulary of Your Program

[Demonstration]

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

Object-oriented systems are frequently built around idioms, design patterns, and other abstractions that can be captured only indirectly in source code. The loss of design information in code often impedes later development activities. Explicit programming attacks this information loss by enabling developers to introduce project-specific design vocabulary into existing general-purpose languages. Using explicit programming, developers can cost-effectively encode design information in a concrete, encapsulated, and reusable way.

In this demonstration, we introduce the principles of explicit programming and demonstrate the ELIDE tool, which supports explicit programming in Java™. ELIDE allows developers to introduce new, parameterized modifiers into the Java language at the class, field, method and block levels. Newly defined modifiers trigger a series of transformations on the source code. Transformations are defined in Java code and can create or modify code anywhere in the source tree. We also describe several applications of explicit programming to increase the design content of systems.

1. THE PROBLEM

As a system is constructed, software developers build up a vocabulary to discuss the design of the system. Unfortunately, most design concepts identified with such vocabulary cannot be mapped directly onto implementation-level constructs offered by general-purpose programming languages. As a result, developers spend a lot of time encoding design concepts into, and decoding design concepts from, verbose and often scattered pieces of code. At best, this encoding and decoding wastes the time of developers. At worst, developers are unable to rediscover the design concepts used in the initial creation. This loss can lead to a degradation in the conceptual integrity of a system [3].

One way to solve this problem is to introduce languages that provide the right vocabulary. This approach can be successful for particular domains as demonstrated by work on domain-specific languages [2]. However, designing and implementing domain-specific languages requires both significant experience with the domain and considerable expertise in programming language implementation. Solving the problem this way is impractical and costly.

In this short presentation, we introduce Explicit Programming as a low-cost way to tackle this problem. We describe the concept of explicit programming(EP), present ELIDE (Extension Language for Incremental Design Encoding), which supports Explicit Programming for Java, and discuss initial uses of the tool.

2. Explicit Programming

Much of the vocabulary used by developers to discuss a system describes consistencies in the code: vocabulary for design patterns, vocabulary for sets of naming conventions, and vocabulary for protocols and idioms within the code, amongst others. Explicit programming allows a developer to gradually and economically extend a general-purpose programming language with such vocabulary. In explicit programming, new vocabulary items are defined generatively: the definitions describe effects on the codebase similar to the code the developers would have written if explicit programming was not available.

To meet the goals of explicit programming, a supporting mechanism must demonstrate several characteristics.

- Vocabulary items must be definable in terms of the general-purpose language concepts familiar to the developer. This constraint ensures that explicit programming is accessible to developers lacking specialized language design knowledge, such as grammars and syntax trees.
- The output of the generative process must be purely in the constructs of the general-purpose language. It must be readily readable by developers and correspond closely to what an implementation would normally look like without explicit programming. This constraint ensures that explicit programming is *non-disruptive*: a developer can apply explicit programming to code that is their responsibility without affecting other members of the development team.
- It must be possible to add new vocabulary items *incrementally*. This constraint ensures that design concepts emerging as development progresses can be refactored and made more explicit.

As an example, consider a group of developers implementing a compiler. When discussing the compiler, the developers may speak in terms of visiting the nodes of the parse tree. If the Visitor design pattern is used to represent this concept, each node in the parse tree will have an `accept` method and each Visitor class will have a

`visit` method for each kind of node in the parse tree. Adding a new node to the parse tree will necessitate the addition of multiple methods in multiple places. Explicit programming allows the developer to describe this concept such that it can be coded directly. For instance, a developer can mark node types with a new vocabulary item, `visited`, and the details of the Visitor pattern can then be generated according to the vocabulary item's definition. Another developer can rely on the generated Visitor infrastructure without needing to know how its implementation was provided.

3. ELIDE

We have built the ELIDE tool (Extension Language for Incremental Design Encoding) to support Explicit Programming in Java. ELIDE allows Java's vocabulary to be extended with new, parameterized modifiers that can be attached to classes, methods, fields, and code blocks. For example, the compiler developer who wishes to capture explicitly the Visitor design pattern can express that the `AssignmentNode` class can be visited by the `NodeVisitor` interface as shown.

```
public visited<NodeVisitor> class AssignmentNode
```

The developer defines the `visited` modifier by writing a Java class that describes the effects of the modifier. In the Java class, the developer uses a library provided by ELIDE to programmatically add the `accept` method to the modified class, `AssignmentNode`, and to add the `visit` method for `AssignmentNode` to the `NodeVisitor` interface.

ELIDE exposes the program being modified to the programmer as a coarse-grained tree of source elements. The developer manipulates this representation using an ELIDE API that builds on the `java.lang.reflect` API, adding additional methods for extending, wrapping, and transforming source elements. ELIDE thus allows developers to specify in normal Java code, the series of changes they would perform to manually encode the design information: operations such as adding new classes to packages, adding new methods and fields to classes, and adding new code to methods and blocks.

ELIDE takes as input Java source using newly defined modifiers and the definitions of those modifiers. It outputs Java source on which the modifiers have been applied. ELIDEgoes through some effort to keep the output code human readable. For example, comments from the input sources are preserved and additional comments are inserted to indicate the application of transformations defined by modifiers.

4. SOME APPLICATIONS

We have applied ELIDE in several cases.

JUnit. We used ELIDE to simplify the writing of tests using JUnit [4]. We defined a `test` modifier that can appear before blocks of test code inside the body of a class definition. This modifier allows a developer to express the test code as close to the code under test as possible. The modifier also automates the creation of test suites for particular classes and packages. Amongst the other modifiers defined was an `assertThrows` modifier, which captures a common but verbose idiom used to test exceptions, and reduces it to a form consistent with the rest of the JUnit API.

JavaBeans. To make JavaBean compliance [1] both explicit and convenient to write, we defined `property`, `boundProperty`, and `constrainedProperty` modifiers. Bean writers can add these modifiers to fields to achieve the desired bean-compliant behavior, including adding appropriate bean-compliant accessors and mutators, event listener registration methods, and introducing supporting fields to the parent class. In addition, the modifiers ensure the parent class implements Serializable.

Refactoring a Visualization System. We used ELIDE to introduce 9 new design vocabulary items into the source code comprising one subsystem of the AVID Java visualization system built at the University of British Columbia. The new design vocabulary makes it simpler to extend the system to add new execution events to visualize. Event type classes that used to contain approximately 100 lines of source code can be described with 5 or 6 lines of code with the new vocabulary available.

5. REFERENCES

[1] Javabeans 1.01 specification, 1997.

[2] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. In *Proc. of ICSR*, 2000.

[3] F.P. Brooks Jr. *The Mythical Man-Month*. Addison Wesley, 1982.

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

Acknowledgements

This work was funded, in part, by the Natural Sciences and Engineering Research Council of Canada (NSERC).