# A Simple Edit-Time Metaobject Protocol

Andrew D. Eisenberg, Gregor Kiczales
Department of Computer Science
University of British Columbia
{ade, gregor}@cs.ubc.ca

## ABSTRACT

We present a simple edit-time metaobject protocol (ETMOP) which runs as part of a code editor and enables metadata annotations to customize the rendering and editing of code. The protocol is layered, so that simple render/edit customizations are easy to implement, while more substantial customizations are still manageable.

Experiments with a prototype implementation of the protocol as an Eclipse plug-in show that the ETMOP is flexible enough to allow easy customization of the rendering and editing of a number of annotations being used in current practice. The flexibility and performance of the prototype suggest that the ETMOP approach is viable and warrants further study.

## 1. INTRODUCTION

For many years, languages like Pascal, various dialects of Lisp, and High Performance Fortran have used metadata annotations to make programs more expressive [10, 16]. Java and C# have recently introduced annotations [2, 13] and practicing programmers now have fairly good support for using annotations as an open-ended extension point for compile- and runtime semantics [5, 9, 14].

Just as compile- and runtime metaobject protocols (and similar technologies) can enable annotations to extend the compile- and runtime semantics of the language, we have been exploring an *edit time metaobject protocol* (ETMOP) that enables annotations to extend the way code is rendered and edited. Our goal in this work is to complement the state of the art with mechanisms that enable annotations to serve as an extension point for making programs more visually expressive. Annotations that are an extension to a base language *still look like annotations* , but using our ETMOP, there is considerable flexibility in how the annotations and the code attached to them are rendered and edited.

By default, code with or without annotations is displayed and edited in the normal way. But an ETMOP programmer can define special metaclasses that customize the display and editing of code with corresponding annotations. The ETMOP is layered, so that simple render/edit extensions are easy to implement, while more substantial extensions are still manageable. We have implemented a prototype of the ETMOP as an Eclipse plug-in, and used it to validate the basic viability of the approach.

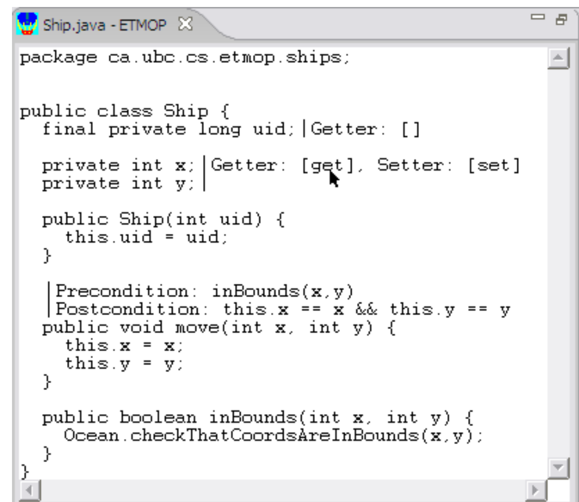


**Figure 1: A screenshot of how the ETMOP renders the code with typical annotations using the ETMOP. In this example the user can click on the getter method prefix to edit it, or click on the other parts of the code to edit them.**

The next section of the paper describes related work. Section 3 describes the ETMOP and its prototype implementation as an Eclipse plug-in. Section 4 presents the implementation of several examples using the ETMOP. This is followed by future work and a summary.

## 2. RELATED WORK

While the distinction is not crisp, the general approaches of tailoring languages to application domains tend to fall into three broad categories: *language-based*, in which the extensions define programming languages ranging from integrated extensions to the main general-purpose language to stand-alone domain-specific languages; *model-based*, which involves not just a shift to a higher level of abstraction, but also explicit manipulation of those models [8, 11, 17]; and editor-based. Our current ETMOP and editor falls into both the language- and editor-based categories.

In the language-based category, many meta-level mechanisms have been developed to support language extension and tailorability, including syntactic and hygienic macros [1], reflection [15] and metaobject protocols [3, 12, 15], generative programming [4] and staged computation [6].

©2006 Andrew Eisenberg and Gregor Kiczales. All rights reserved. Accepted for publication at OAL'06

```
class Ship {
  @Getter("")
  final private long uid;

  @CombineRight @Getter("get") @Setter("set")
  private int x;
  @Previous
  private int y;

  public Ship(long uid) {
    this.uid = uid;
  }
  @CombineTop
  @Precondition("inBounds(x, y)")
  @Postcondition("this.x == x && this.y == y")
  public void move(int toX, int toY) {
    x = toX;
    y = toY;
  }
  private boolean inBounds(int x, int x) {
    ..check that coords are within bounds..
  }
}
```

**Figure 2: The raw source text of the screenshot.**

The editor-based tailoring of languages has primarily focused on development environments that are open and customizable. The Eclipse plugin architecture, metaprogramming in the Squeak IDE, and ELisp for Emacs have all been used to provide extended rendering and editing for programming languages such as syntax highlighting and alternate syntaxes. However, these systems do not provide a specific protocol to extend the display and editing of programming languages.

# 3. ARCHITECTURE

Our ETMOP enhanced editor has a simple hybrid text/structure architecture. Programs are stored as plain text in the traditional way. Annotations are written using the Java 5 syntax [2]. When a file is loaded into the editor, it is parsed into an abstract syntax tree (AST), which is then rendered and edited in a variety of ways. The AST is serialized back to text in order to save the file.

Parsing is itself a two phase process. In the first phase, the text is parsed to produce an AST for the program. We use the `org.eclipse.jdt.core.dom.ASTParser`, which produces a typical AST structure. In this AST, annotations appear are attached to declarations. Comments are stored and can be retrieved as strings. We have extended the `ASTParser` class slightly, to provide a few additional fields and methods used by the ETMOP.

The second phase of parsing is to walk the AST in order to produce *render edit metaobjects* (REMOs) for each AST node. Nodes without annotations get an instance of `DefaultREMO`. When annotations are encountered, a registry is consulted to determine whether a REMO metaclass has been registered for that annotation. If so, an instance of that metaclass is created for the node and the annotation itself is flagged to indicate that a REMO was created for it. This is called the REMO binding phase.

The ETMOP supports a restricted form of REMO sharing between AST nodes. Multiple AST nodes can share a single REMO, as long as they are immediately neighboring siblings. This allows a REMO to group the rendering of sibling AST nodes when that is appropriate. In Figures 1 and 2 this is what enables a single vertical line and Getter/Setter notation to appear next to several field declarations.

To represent this restricted REMO sharing in the source text, we use a special annotation, `@Previous`, which the REMO parsing phase interprets as meaning that the REMO for this AST node should be the same as for the previous sibling.

We also support several simple generic compositions of multiple REMOs. This functionality is provided by several generic annotations and REMO metaclasses, such as `@CombineRight` used in Figures 1 and 2. This is convenient when, for example, a field has both a getter and a constraint annotation.

## 3.1 Logical Layout

Rendering the AST occurs in two stages. The first is *logical layout*, which controls how AST structure will be grouped in the final rendering. Logical layout walks the AST and produces a hierarchical structure of *boxes*, which corresponds to Cartesian rectangles of as yet undetermined size in the final rendering. The second stage is *physical layout*, which traverses the box structure and constructs lower-level *figures* that will be displayed on the screen.

This two-step process simplifies the architecture and helps with the layering of the protocol, because many user-defined REMO metaclasses only need to customize logical layout.

Logical layout is driven by a `LogicalLayout.Visitor` which calls `logicalLayout` methods on REMOs, which in turn call `accept` methods on AST nodes as follows: (see Figure 3)

a `LogicalLayoutVisitor` calls `logicalLayout` on the node's REMO to produce the layout.

b The `logicalLayout` method on `DefaultREMO` simply calls `accept` on the node itself.

c Each of the `accept` methods on subclasses of `ASTnode` classes knows how to layout that kind of AST node into a `JavaBox`. Except for leaf nodes, this of course involves recursive calls back to the `visit` method on the visitor with a sub-tree of the node.

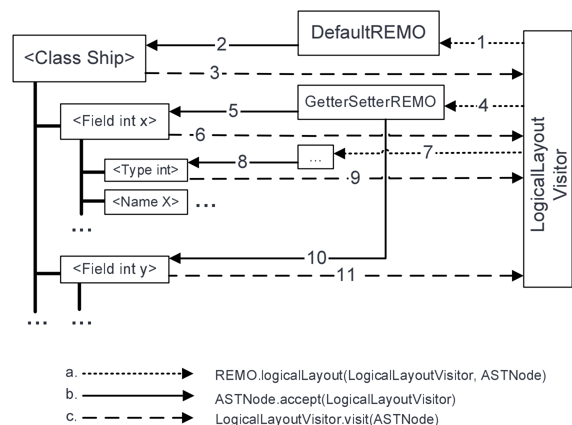


**Figure 3: Flow of the logical layout protocol**

```
public class CombineRightREMO extends DefaultREMO {

  // 1
  private Annotation              annotation;
  @SuppressWarnings({"unused"})
  private List<ASTNode>           asts;
  private List<ICombinableRight> subRemos;

  // 2
  public CombineRightREMO(Annotation annotation,
                          List<ASTNode> asts) {
    this.annotation = annotation;
//  this.asts       = asts;
    this.subRemos   = parseSubRemos(annotation, asts);
  }
  public Box logicalLayout(LogicalLayoutVisitor v,
                           List<AST> asts) {
     List<Box> noteBoxes = new LinkedList<Box>();
     // 3
     for( ICombinableRight subRemo : subRemos ) {
       noteBoxes.add(subRemo.logicalLayoutNoteBox(v, asts));
     }
     // 4
     return makeRow(super.logicalLayout(v, asts),
                    makeVLine(),
                    makeCommaSeparatedRow(noteBoxes));
  }
  ...
}
```

**Figure 4: Fields, constructor and `logicalLayout` method for `CombineRightREMO`.**

The logical layout protocol is also responsible for creating handlers that are called to respond to mouse events on the rendered structure. The behavior of these handlers is described in Section 3.4.1.

## 3.2 Extending Logical Layout

There are several ways user-defined REMOs can customize behavior under the ETMOP. The simplest is that a special class of REMO can override the default `logicalLayout` method with a new method that first allows the default recursion to layout the AST into boxes as normal, but then arranges or rearranges those boxes in its own box structure that includes appropriate layout of the REMO's special behavior. This is the strategy used by `CombineRightREMO` in Figure 4.

To facilitate writing user defined `logicalLayout` methods, the ETMOP includes a library of useful primitive box metaclasses, including `TextBox`, `JavaBox`, `VLineBox`, `HLineBox` and others.

As shown in Figure 4 `CombineRightREMO` mentioned above is actually implemented at user-level, using simple customization. Reading through the members of the class, this code says that:

1. The REMO stores the annotation that gave rise to it, the AST nodes to to which the REMO is attached, and a list of the REMOs it combines. Each of those REMOs must implement the `ICombinableRight` interface, which defines a sub-protocol among combinable REMOs.

   Note that the `asts` field is purposely left unread. The protocol allows the AST nodes received during logical layout to differ from the ones attached during construction. This means that in general, a REMO may need to store the AST nodes to which it is attached.

2. Constructors for REMOs always receive the annotation and a list of the AST nodes which share that annotation. This constructor parses the annotation sub-structure to produce the sub REMOs and stores them.

3. The `logicalLayout` method first invokes the sub-protocol for `ICombinableRight` REMOs, by asking each individual sub REMO for its contribution to the combined layout. This is done by calling the `logicalLayoutNoteBox` method on each sub REMO.

   It then calls `super.logicalLayout` to produce the default logical layout for the declarations themselves (without the contribution from the REMOS.

4. The layout of the declarations and the layout of the individual note boxes is then combined to form the complete result: a single row, starting with the laid out declarations, then a vertical line, then the laid-out note boxes in a comma-separated sub-row. The effective rendering is as shown in Figure 1.

## 3.3 Physical Layout

The physical layout pass is responsible for taking the box structure resulting from the logical layout phase and converting it to a structure of lower-level graphical figures. At this point the layout work becomes dependent on the specific graphics API. This pass is driven by a `PhysicalLayoutVisitor` which traverses the box structure creating figures for each box it encounters. The figures are themselves hierarchical, and they contain additional information specific to the Draw2D [7] graphics API we are using, including exact locations.

By design, physical layout is less extensible than logical layout. Most specialized REMOS can do all their work during logical layout. Exceptions occur when the REMO specifically creates specialized, non-textual graphics.

## 3.4 Editing

Our ETMOP supports editing the rendered program in a way that allows specialized REMOs to define specialized editing commands. The protocol allows REMOS to associate interaction handlers with regions of the rendering. We also provide an API for editing the AST structure as well as a library of command and editor widgets that facilitate writing custom commands and editors.

### *3.4.1 Interaction Handlers*

During logical layout, every box can optionally have an *interaction handler* associated with it. Each handler returns a list of commands available on the box. Using a simple version of the Chain of Responsibility Pattern, boxes without handlers simply defer to their parent box handler.

In this way, whenever a mouse click occurs over a box a list of commands, from generic to specific, is built up for the box. In the canonical way there is a single command for left click, and a list of commands pop up on a menu for right click. The MOP provides a simple API that allows structured editing of the AST. It also provides a number of reusable components from which custom commands and editors can be built.

# 4. EXAMPLES

## 4.1 AspectWerkz á la AspectJ

One of the most interesting applications of our ETMOP has been to use it to make AspectWerkz code, written with annotations, look and feel like AspectJ code, which extends the Java Language Syntax.[1] The annotations involved are all defined by AspectWerkz. Using the ETMOP 789 SLOC (source lines of code) define a collection of REMOs that work together to cause AspectWerkz code to be rendered and edited as if it was AspectJ code. So the effect is that code which appears in the source file as:

```
@Before("call(void Point.setX(int)")
void beforeAdviceBody() {
  System.out.println("hello");
}
```

is rendered in the editor, and can be edited as:

```
before(): call(void Point.setX(int)) {
  System.out.println("hello");
}
```

## 4.2 UML State Chart

We have also implemented a REMO that allows editing of state charts. In logical layout, this REMO creates a `GraphicsBox`; then in physical layout the state chart is rendered using various graphical figures like circles and lines. We have used this example to test that the ETMOP layering makes it possible to write very specialized, graphical, domain-specific notations. While this is the largest REMO we have written, 830 SLOC seems like a quite reasonable amount of code for this degree of customization.

# 5. SUMMARY AND FUTURE WORK

We have developed a simple edit-time metaobject protocol that gives programmers significant control over the way programs are rendered and edited. More elaborate examples such as AspectWerkz à la AspectJ are causing us to reassess past answers to questions of the boundary between the language and the tools.

Our work to date shows that an ETMOP such as this one appears useful and viable: we have been able to write REMO metaclasses that control code rendering and editing in ways that we find more expressive. Performance analysis of the prototype puts it within a factor of two to three of Eclipse JDT for space consumption and time, and we already see major possibilities for optimization.

Our main goal in future work is to be able to run two kinds of further studies. In one, we plan to re-work a reasonably large system using this technology, to systematically explore the opportunities for making the code more expressive using REMOs. In the other we plan to make a version of our ETMOP publicly available and evaluate how other users work with it.

Both of these experiments will require significant improvements in our implementation. We will have to achieve tighter integration with Eclipse JDT. To the greatest extent possible the experience of using our tool will have to be that it adds to JDT, not that it partially mimics JDT.

[1]In release 5, AspectJ supports both the traditional AspectJ syntax and the AspectWerkz syntax, and calls them "code style" and "annotation style" respectively. Since this is a new development, we use the older terms AspectJ style and Aspectwerkz style.


# 6. ACKNOWLEDGMENTS

This research was funded in part by IBM (EIG) and by the Natural Sciences and Engineering Research Council of Canada.


# 7. REFERENCES


[1] J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 31–42. ACM Press, 2001.

[2] J. Bloch. JSR-175: A Metadata Facility for the Java™Programming Language. Technical Report Final Release, Sun Microsystems Inc., Sept. 2004.

[3] S. Chiba. A metaobject protocol for C++. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 285–299, New York, NY, USA, 1995. ACM Press.

[4] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[5] J. Darcy. JSR-269: Pluggable Annotation Processing API. Technical Report Early Draft Review, Sun Microsystems Inc., June 2005.

[6] R. Davies and F. Pfenning. A modal analysis of staged computation. In *Proceedings of the 23rd Symposium on Principles of Programming Languages (POPL)*, pages 258–270, 1996.

[7] Eclipse Tools Project. Draw2D, part of the GEF Project. `http://www.eclipse.org/gef/`.

[8] M. P. J. Fromherz and V. A. Saraswat. Model-based computing: Using concurrent constraint programming for modeling and model compilation. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 629–635, London, UK, 1995. Springer-Verlag.

[9] J. D. Herrington. CodeGeneration Network. `http://www.codegeneration.net/`.

[10] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 0.4*, 1992.

[11] G. Karsai, J. Sztipanovits, H. Franke, S. Padalkar, and F. DeCaria. Model-embedded on-line problem solving environment for chemical engineering. In *Proceedings of 1st Conference on Engineering of Complex Computer Systems*, pages 227–233, 1995.

[12] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[13] Microsoft Corporation. C# Version 1.2 Specification. Technical report, Microsoft Corporation, Sept. 2003. `http://msdn.microsoft.com/vcsharp/programming/language/`.

[14] R. Pawlak. Spoon: Annotation-driven program transformation - the aop case. In *Proceedings of the 1st workshop on Aspect-Oriented Middleware Development*, Grenoble, France, Nov. 2005.

[15] B. C. Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, Jan. 1982. MIT-LCS-TR-272.

[16] G. L. Steele. *Common Lisp the Language, 2nd edition*. Digital Press, 1990.

[17] J. Sztipanovits, G. Karsai, and H. Franke. Model-integrated program synthesis environment. In *Proceedings of the 3rd Conference and Workshop on the Engineering of Computer Based Systems*, pages 348–355, 1996.