Incomplete Resolution of References in Eclipse

Joseph J. C. Chang and Robert J. Walker
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
{jjcchang, walkerr}@ucalgary.ca

ABSTRACT
In the Eclipse JDT, the Java reference resolution rules are applied as fully as possible, thereby either determining the unique target for a given reference or signalling that the reference cannot be resolved. However, a variety of development tasks require the manipulation of code for which incomplete resolution of references is both possible and useful. This paper motivates the need for incomplete resolution during a software reuse-and-integration task and the difficulties that result. A proof-of-concept implementation is described that is used as a basis for reuse tool support and that can be used for other transformation tools.

1. INTRODUCTION
Modern software development depends heavily on the use of references, e.g., to types and to methods. In programming languages like Java, references can possess a degree of ambiguity when considered on a localized basis. For example, method names can be overloaded; type names may lack full qualification; polymorphism can render the target of an invocation unobvious. Java provides rules so that any reference can be resolved to its target. These rules require that a complete program be present; unresolved references signal (semantic) programming errors. However, a variety of development tasks—such as software reuse—require the manipulation of code for which complete resolution of references is either not possible or otherwise inappropriate.

Despite its problems, software reuse via copy-and-modify remains prevalent [10]. While research continues into modularization-based approaches for software reuse (e.g., component-based systems [13], aspect-oriented software development [9,4], or Batory's feature-oriented programming [3]), pragmatism suggests that improved tool support for copy-and-modify is needed, at least in the short-term.

A major obstacle to large-scale software reuse is the dependencies that exist between a given module and the original environment where it was located [6]. When developers perform copy-and-modify reuse manually, they replace references to targets in the old context, which have become invalid, with the most appropriate ones in the new context. Tool support can help to ensure that this is done consistently and accurately, similar to how refactoring tool support can help to ensure that tedious and error-prone refactoring modifications are done consistently and accurately. While the Eclipse Java Development Tools (JDT) support many development tasks effectively, its support for copy-and-modify tasks is insufficient. Ultimately, the JDT treats references as being bound to particular targets; in the presence of code fragments, references may not be resolvable or one may desire to rebind references to other targets. Section 2 examines this issue in greater detail through a motivational example.

Tool support for a copy-and-modify task can be provided via an initial step of *incomplete resolution of references*. Incomplete reference resolution takes into account only a specified portion of a project, considering references that remain unresolvable within that portion to be unbound. Section 3 details the technical problems involved in providing tool support for incomplete reference resolution, our conceptual approach for overcoming these problems, and our proof-of-concept implementation of an incomplete resolution mechanism as an Eclipse plug-in.

Of course, incomplete reference resolution is solely a foundation upon which tools can be constructed. In the context of copy-and-modify support, references that are determined to be unbound from within a reused section of code can subsequently be bound to targets in its new environment. There are a variety of ways in which such rebinding can take place. In addition, other tasks and their supporting tools can utilize incomplete reference resolution. Section 4 describes these issues (and related work) in brief, but details of the particular approach that we are pursuing [15,16] largely lie beyond the scope of this position paper.

The contributions of this position paper are the identification of incomplete resolution of references as a useful foundation for non-meaning preserving transformations and the description of a proof-of-concept implementation as an Eclipse plug-in.

2. MOTIVATION
A great deal of code reuse is still performed using a copy-and-modify approach. Existing code is first copied into a new system, and then modified to fit the new environment. This is an invasive, error-prone and tedious process, especially in large and complex systems.

The following example is intentionally simple to motivate how the copy-and-modify approach is difficult to avoid given the current reference resolution capabilities of Java and Eclipse. It will also demonstrate a situation where incomplete reference resolution can aid in the reuse of code between systems.

Suppose you have a `ScrollingPanel` compound widget that is composed of two simple widgets, as shown in Figure 1.

To appear at the OOPSLA 2005 Eclipse Technology Exchange (eTX) Workshop.

```
public class ScrollingPanel {
    private Panel panel;
    private ScrollBar scrollBar;

    public ScrollingPanel() {
        panel = new Panel();
        scrollBar = new ScrollBar();
    }

    // ...
}
```

Figure 1: Code snippet from a compound widget for scrolling panels.

The `ScrollingPanel` widget is currently part of a system where both the `Panel` and the `ScrollBar` widgets are concrete classes. Suppose that we wish to reuse this code in a different system, where the widgets are not obtained by instantiation from concrete classes but rather, are obtained through the use of the Abstract Factory design pattern [5]. If we were to copy the `ScrollingPanel` from `WidgetLibraryA` to `WidgetLibraryB`, as shown in Figure 2, we would have to modify the `ScrollingPanel` to make use of the Abstract Factory pattern to retrieve the two widgets.

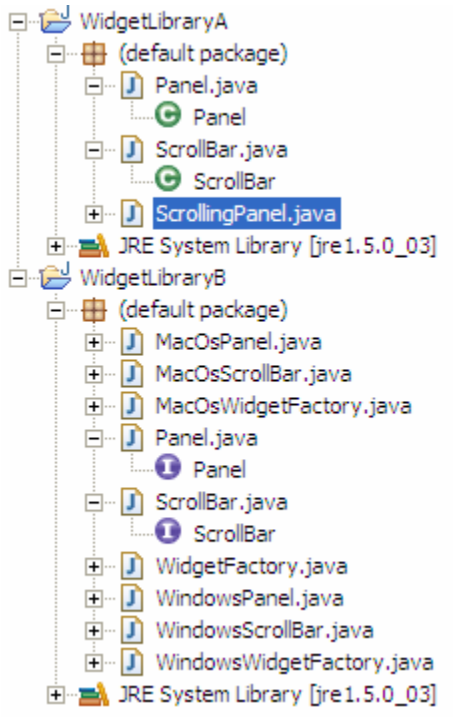

Figure 2: `ScrollingPanel.java` from `WidgetLibraryA` is to be reused in `WidgetLibraryB`.

If the scrolling panel is moved without modification from `WidgetLibraryA` to `WidgetLibraryB`, Eclipse displays `ScrollingPanel.java` as shown in Figure 3. As indicated by the lines underneath the `Panel` and `ScrollBar` object instantiations, Eclipse and Java do not consider the code to be valid within the context of `WidgetLibraryB`. In addition, Eclipse does not provide automated refactoring support to make the code valid by converting the existing code to use the Abstract Factory pattern, according to the desired result shown in Figure 4. As a result, the only way to make the `ScrollingPanel` work in `WidgetLibraryB` is by modifying the code manually.

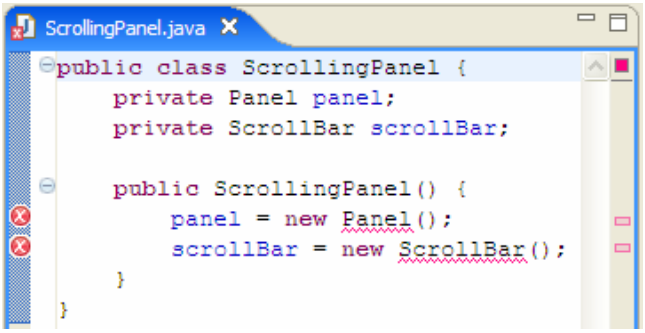

Figure 3: `ScrollingPanel.java` after it is moved without modification from `WidgetLibraryA` to `WidgetLibraryB`.

```
public class ScrollingPanel {
    private Panel panel;
    private ScrollBar scrollBar;

    public ScrollingPanel(WidgetFactory factory) {
        panel = factory.createPanel();
        scrollBar = factory.createScrollBar();
    }

    // ...

}
```

Figure 4: `ScrollingPanel.java` after it has been modified to fit the environment of `WidgetLibraryB`.

Invasively modifying code in order to reuse parts of another system is a problematic way to effect code reuse [10]. While the manual modifications required to allow `ScrollingPanel` to conform to its new environment can be performed with little difficulty, such an approach would not scale well when dealing with increasingly large systems. In the next section, we consider how incomplete resolution of references can aid in such a copy-and-modify task.

3. INCOMPLETE RESOLUTION OF REFERENCES

Incomplete reference resolution does not take into account the entire project being developed. Instead, resolution is performed with respect to a *resolution boundary*. Inside a resolution boundary, all references, including names and method invocations, are interpreted and resolved using only the information that resides within that boundary. Incomplete resolution of references allows for the code within a specified resolution boundary to be incomplete or inconsistent from the global perspective (i.e., the perspective of the containing Eclipse project). Unfortunately, neither Java nor the Eclipse JDT directly supports this capability. They cannot perform reference resolution using boundaries of different granularity; instead, their global reference resolution attempts to resolve all elements to the

appropriate construct using whatever information is available in the entire project.

The information provided by an incomplete reference resolver is significantly less detailed than what can be provided by a global reference resolver, whose outputs are associated with some actual construct available in or from the project. However, a global reference resolver may incorrectly tie a reference to an external entity. In contrast, the resolutions produced by an incomplete reference resolver do not falsely constrain the code within the boundary.

3.1 Problems and Approach

Suppose that we have another scrolling panel class, as shown in Figure 5, that was created in an environment similar to `WidgetLibraryA`, except that both `Panel` and `ScrollBar` extend the class `Visual`. Now, suppose we want to move this scrolling panel class to `WidgetLibraryB` (the `Visual` class will not be moved because `WidgetLibraryB` has its own way of displaying widgets). We consider the problems one would encounter in this task, below.

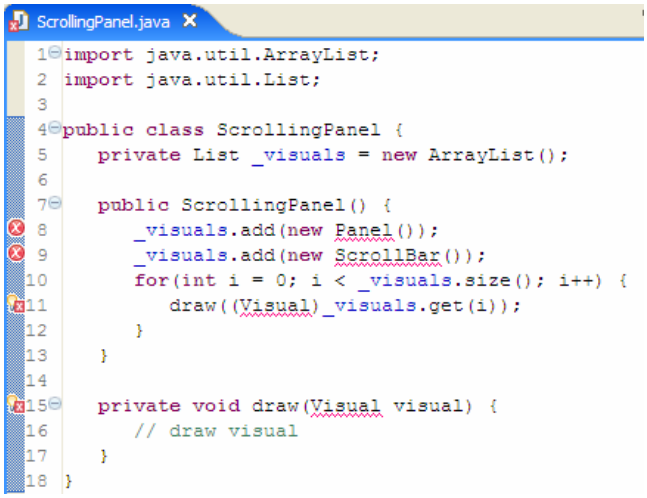

Figure 5: A different scrolling panel widget.

3.1.1 Name Resolution

Name resolution is performed differently by an incomplete and a global reference resolver. For example, in Line 5 of Figure 5, the Eclipse JDT will determine that the line is valid. It will determine that the expression `new ArrayList()` will resolve to the class `java.util.ArrayList`, and that this class implements the `java.util.List` interface, making the assignment statement valid.

The incomplete reference resolver requires a boundary in order to proceed. For the example of Figure 5, let us set the boundary around the class `ScrollingPanel`. Note that the import statements are outside of the boundary, and therefore cannot be taken into consideration when resolving references. The incomplete reference resolver can function with incomplete information; it considers Line 5 to be valid because it does not have the ability to verify whether an assignment of `ArrayList` to a variable of type `List` is invalid within the given boundary. As a result, `_visuals` is merely resolved to the type `List` (not `java.util.List`). The incomplete reference resolver does not and cannot obtain any extra information about `List`, which makes incomplete reference resolution more flexible (it can function with access to less information), but less comprehensive.

The extra flexibility of the incomplete reference resolver shows its value in Lines 8 and 9. Using the global reference resolver of the Eclipse JDT, these lines would be considered invalid, as it is known that `Panel` and `ScrollBar` are interfaces and, therefore, cannot be directly instantiated. In contrast, the line is perfectly valid for the incomplete reference resolver, with our specified resolution boundary, as it does not know any details about `Panel` or `ScrollBar`, and therefore cannot verify the correctness of the line. Instead, it merely resolves these expressions to the best of its ability; `new Panel()` resolves to the type `Panel` and `new ScrollBar()` resolves to the type `ScrollBar`.

Additional tool support will enable these expressions to be connected to the appropriate constructs within the new environment. There are many ways that this can be done; Section 4 covers this topic in more detail.

3.1.2 Method Resolution

Further differences between incomplete reference resolution and the Eclipse JDT global reference resolution can be seen when resolving method declarations referenced from method invocations. In the Eclipse JDT, the method invocation from Line 11 does not resolve to a Java element, as shown in Figure 6. This is because, in `WidgetLibraryB`, there is no `Visual` class, so the global reference resolver cannot determine how to match the method invocation to the method declaration.

The incomplete reference resolver handles Line 11 of Figure 5 differently. Since there is no knowledge about the existence, or lack thereof, of the `Visual` class, expressions merely resolve to a name representing the type. It is apparent by the cast that the argument to the `draw` method is supposed to be of type `Visual`. As a result, the incomplete reference resolver will be able to correctly resolve the method invocation of Line 11 to the method declaration of Line 15.

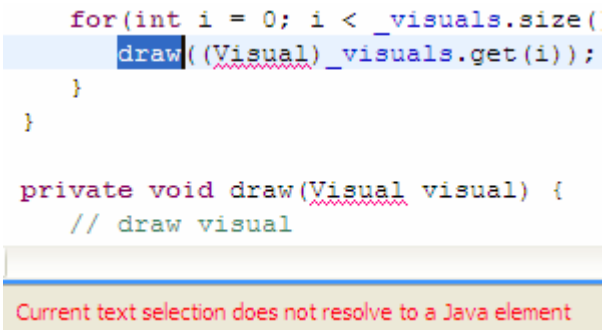

Figure 6: Reference cannot resolve to Java element.

3.2 Implementation

We have realized incomplete reference resolution as a proof-of-concept component of an Eclipse plug-in for applying non-meaning-preserving transformations to Java source code that is incomplete.

In order to perform incomplete reference resolution, two main inputs are required: a resolution boundary, such as a class or a method; and the code within that boundary. The boundary can be specified by providing the boundary name (i.e., the fully-qualified path to the element representing the boundary) and the boundary type (indicates whether the boundary surrounds a type or a method). For example, if the boundary is to surround the method `resolve(Name name)` in the class `ReferenceResolver` in the package `incomplete`, the boundary name would be `incomplete.ReferenceResolver.resolve(Name)`, and the boundary type would be specified as a method boundary. This provides the incomplete reference resolver with enough information to unambiguously store the boundary information, as well as obtain the code within the boundary.

The Eclipse JDT allows one to request bindings for many Java elements. Unfortunately, it does not allow one to set the boundary that will be used when bindings are resolved. They are always resolved taking into account all of the information available from the Eclipse Java project. As a result, there is no point in requesting bindings when Eclipse parses the Java code into an abstract syntax tree (AST), as these bindings will not be consistent with the resolutions produced with only the information internal to the boundary.

A standard approach to the problem of resolving references to types and method declarations within a boundary is to step through the code and construct a symbol table using only the information that is within the boundary. Such an approach can become quite complex; instead, it would be desirable to leverage existing infrastructure provided by the Eclipse JDT to minimize the amount of work required to achieve the same result.

While the bindings provided by the Eclipse JDT are not themselves useful, the AST and its support for the Visitor design pattern [5] provide an easy way to traverse and analyze code that has already been broken up into a set of objects representing the Java source code. This set of objects is part of the Eclipse JDT Core Document Object Model (DOM). The following sections will show that this infrastructure provided by Eclipse will be very useful in providing support for incomplete reference resolution.

3.2.1 Retrieving Boundaries

The ability to retrieve a boundary, which can be an Eclipse JDT `TypeDeclaration` or `MethodDeclaration`, is an important part of incomplete reference resolution. The Eclipse JDT API provides access to objects representing a Java project, its packages, and its compilation units (.java files). From a boundary name and a boundary type, it is possible to obtain the compilation unit that corresponds to the boundary. This compilation unit is the root of an AST that can then be traversed to find the appropriate AST node that represents the boundary.

For example, given a boundary name like `pkg.MyClass.method(String)` and a method boundary type, the compilation unit `pkg.MyClass` can be found. That can be traversed to find the method declaration represented by `method(String)`.

3.2.2 Expression Resolution

A necessary part of incomplete reference resolution is the ability to resolve an expression to a type.

Many expressions are easy to resolve to types. For example, a `CastExpression` will simply resolve to the type to which the variable is being cast. By contrast, both simple and qualified names are more complicated to resolve to types. The main complication is that names can represent variables which must be bound to types by taking into account the variable scoping rules present in Java, but using only the information that is available within the confines of the boundary.

Variables can be declared in many places, including within blocks, as method parameters, and as field declarations. There are multiple ways in which a variable can be declared, including as part of a `SingleVariableDeclaration`, or a `VariableDeclarationStatement`. Subclassing the `ASTVisitor` provided by the Eclipse JDT is an effective way of moving between the types, methods, and blocks necessary to resolving variables.

Another difficult to resolve expression is the method invocation, which is the subject of the next subsection.

3.2.3 Method Invocation Resolution

Incomplete reference resolution must include the ability to resolve a method invocation to a method declaration. Method invocations are made difficult to resolve by the potential for method overloading. This means that a method cannot be uniquely identified within a class solely by its name. Instead, both the number and types of the arguments must be considered in order to match a method invocation to its declaration.

The `ASTVisitor` provides a simple means of accessing all of the method invocations and declarations within a boundary `ASTNode`. This, and the ability to resolve expressions into types as described in the previous subsection, allows the incomplete reference resolver to match a method invocation to a method declaration within a boundary, if one exists.

4. DISCUSSION AND RELATED WORK

The idea of scoped binding resolution can be extended to include more than just types and methods as potential boundaries. Resolution boundaries can also be considered around fields, packages, or arbitrary combinations of any of these [15,16]. Such additional boundaries can aid in composing arbitrary sections of code into a new context. Arbitrary boundaries are also reminiscent of the notion of *crosscutting concerns* [9].

An alternative approach to composition of code exists in multi-dimensional separation of concerns [14] with its attendant Hyper/J tool and its more recent replacement, the Concern Manipulation Environment (CME). CME is intended as a basis for compositional tools, such as those supporting aspect-oriented programming. The approach has a strong notion of *declarative completeness*, i.e., that every reference can be fully resolved. In the context of partial reuse of a system, this requires that constraints imposed by the old environment must be reconciled with those of the new environment. Furthermore, declarative completeness can only be provided in contexts where full resolution can be performed, which is not the case with arbitrary code fragments. In contrast, our incomplete resolution approach limits the constraints to those imposed by the code being reused, and supports developer-oriented copy-and-modify operations.

We are implementing a tool, called IConJava [16], for compositional transformations that utilize our incomplete reference resolution approach. This tool takes as input the source code to be

composed and a description of the transformations to be performed, in a specialized language. This approach is reminiscent of declarative approaches to realizing refactorings [11], save that the transformations we permit need not be meaning-preserving. The alternative approach of interactive, step-wise transformation is also possible, and would be more similar in nature to the refactoring support provided by the Eclipse JDT. An interactive approach is useful for simple sequences of transformations, but more complex transformations that require debugging and that are to be applied repeatedly are less practicable in the interactive approach. Recent work by Henkel and Diwan [7] suggests that intermediate solutions are possible, by recording the interactions of a developer with Eclipse during refactoring tasks in a macro-like fashion; a similar approach could be taken for non-meaning-preserving transformations. Regardless of the point on the spectrum that one chooses for the transformation tool, incomplete reference resolution would provide beneficial support.

Balaban et al. [2] have presented a means for porting an application from an old version of a class library to a new version. This goal is easier to achieve than the integration of arbitrary code in the absence of the complete original system, because both versions of the class library are available for analysis and there is an assumption of greater consistency. However, their approach does utilize type inference to improve the translation process, and a similar approach would be useful in the support of incomplete resolution of references. Each reference within a resolution boundary provides a set of constraints on the entity that can satisfy that reference. By building up a set of facts about each reference one can build a picture of the constraints that must either be satisfied or transformed; this is effectively the notion of local type inference [12]. We are working towards integrating a local type inferencing approach with our toolset [8] to detect simple errors by the developer, such as transforming a field access into a method invocation on the left-hand side of an assignment.

Recent work by Ancona et al. [1] indicates the value of flexibility beyond the compilation stage. They maintain the ambiguity of unresolved references into a modified Java bytecode language, for delayed resolution and composition. Their approach points to another use for a tool supporting incomplete resolution of references.

5. CONCLUSION

In this position paper, we have presented the notion of incomplete resolution of references and motivated its usefulness in support of a variety of tools, particularly ones involved in improved copy-and-modify tasks. We have outlined the issues involved in providing such support and briefly described our implementation of an Eclipse plug-in for this purpose. This plug-in is used as the back-end for a transformation tool called IConJava that we continue to research; the plug-in could also be used as a back-end for a variety of other tools involving composition. Although the idea of incomplete reference resolution is motivated in this paper through copy-and-modify code reuse, IConJava in particular is intended to support stronger forms of evolution and reuse.

6. ACKNOWLEDGMENTS

We thank Reid Holmes, Shafquat Mahmud, Mark McIntyre, Kevin Viggers, and our "anonymous" reviewers for their comments on this paper. This work was supported in part by an NSERC Discovery Grant.

7. REFERENCES

[1] Ancona, D., et al. Polymorphic bytecode: Compositional compilation for Java-like languages. *Proc. ACM SIGPLAN Conf. Principles of Programming Languages*, 26–37, 2005.

[2] Balaban, I., Tip, F., and Fuhrer, R. Refactoring support for class library migration. *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications*, 2005. To appear.

[3] Batory, D. Feature-oriented programming and the AHEAD tool suite. *Proc. Int. Conf.Software Engineering*, 702–703, 2004.

[4] Filman, R., Elrad, T., Clarke, S., and Akşit, M., editors. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[5] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*. Addison-Wesley, 1994.

[6] Garlan, D., Allen, R., and Ockerbloom, J. Architectural mismatch. *Proc. Int. Conf. Software Engineering*, 179–185, 1995.

[7] Henkel, J., and Diwan, A. CatchUp!: Capturing and replaying refactorings to support API evolution. *Proc. Int. Conf. Software Engineering*, 274–283, 2005.

[8] Jaeger, S. ECO Design Description. Internal research report, Software Modification Laboratory, Dept. of Computer Science, Univ. of Calgary, October 2004.

[9] Kiczales, G., et al. Aspect-oriented programming. *Proc. European Conf. Object-Oriented Programming*, 220–242, 1997.

[10] Krueger, C.W. Software reuse, *ACM Computing Surveys*, 24(2):131–183, June 1992.

[11] Mens, T., and Tourwé, T. A declarative evolution framework for object-oriented design patterns. *Proc. Int. Conf. Software Maintenance*, 570–579, 2001.

[12] Pierce, B.C., and Turner, D. N. Local type inference. *ACM Trans. Progr. Lang. and Syst.*, 22(1):1–44, January 2000.

[13] Szyperski, C., Gruntz, D., and Murer, S. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002.

[14] Tarr, P. Ossher, H., Harrison, W., and Sutton, S. *N* degrees of separation: Multi-dimensional separation of concerns. *Proc. Int. Conf. Software Engineering*, 107–119, 1999.

[15] Walker, R.J., and Murphy, G.C. Implicit context : Easing software evolution and reuse. *Proc. ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, 69–78, 2000.

[16] Walker, R.J. *Essential Software Structure through Implicit Context*. Ph.D. thesis, Dept. of Computer Science, Univ. of British Columbia, March 2003.