A visual interface for type-related refactorings

Philip Mayer
Institut für Informatik
Ludwig-Maximilians-Universität
D-80538 München
plmayer@acm.org

Andreas Meißner
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
meissner@acm.org

Friedrich Steimann
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org

ABSTRACT
In this paper, we present our approach to a visual refactoring tool, the Type Access Analyzer (TAA), which uses program analysis to detect code smells and for suggesting and performing refactorings related to typing. In particular, the TAA is intended to help the developers with consistently programming to interfaces.

1. INTRODUCTION
When looking at currently available type-related refactoring tools, a noticeable gap shows between simple refactorings like *Extract Interface* and more complex, “heavyweight” ones like *Use Supertype Where Possible* and *Infer Type* [2]: While the former do not provide any analysis-based help to the user, the latter perform complex program analyses, but due to their autonomous workings – without interacting further with the user except for preview functionality – it is not always clear when to apply them, what result to expect, and just how far the changes of the refactorings will reach. For example,

- *Extract Interface* keeps programmers in the dark about which methods to choose,
- *Use Supertype Where Possible* replaces all declaration elements found without a proper way of restricting it,
- *Infer Type* creates new types guaranteeing a type-correct program, but often lacking a conceptual justification.

As a remedy, we propose a new approach to refactoring. The contributions of this approach consist of:

- moving precondition checking and parameterization from refactorings to a dedicated program analysis component,
- presenting the analysis results visually in such a way that they suggest refactorings, and
- breaking down existing refactorings into simpler tools which perform predictable changes immediately visible and controllable by the visual refactoring view.

This approach is prototypically realized in our Type Access Analyser (TAA) tool for type-related refactorings.

2. THE TYPE ACCESS ANALYZER
A loosely coupled and extensible software design can be reached by consistently programming to interfaces [1], specifically to what we have called *context-specific interfaces* [3]. An interface is considered to be context-specific if it contains exactly – or, in a more relaxed interpretation, not much more than – the set of members of a type required in a certain area of code (which is comprised of variables and methods declared with the interface as their types and their transitive assignment closure).

Refactoring to the use of such interfaces requires an analysis of what is really needed in contexts by analyzing the code to find used or unused members. With this information, the code can be refactored in an informed way by:

- creating, adapting, or removing interfaces, and
- retrofitting existing variable types to the newly introduced, or adapted, interfaces.

The TAA follows the approach discussed in the introduction by analyzing the code using the type inference algorithm we have introduced in [2], presenting the results in a visual form, and providing access to and feedback from simplified versions of refactorings such as *Extract Interface* or *Infer Type*.

To give the programmer a comprehensive and concise view of the program that is tailored to the specific problems of interface-based programming, we have developed the supertype lattice view described in [4]. In this view, the supertype hierarchy of the type under consideration is enhanced by displaying a bounded lattice of the set of members of the type, each node being enriched with various kinds of information. Figure 1 shows a screenshot of the TAA in action on a four-method class (due to space limitations, only a part is shown).

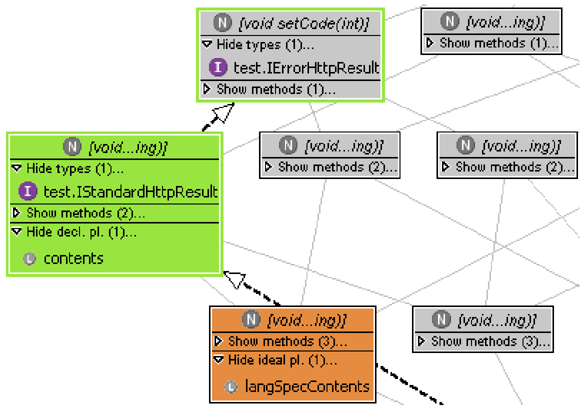

Figure 1: TAA Visual View

Four types of information are immediately visible from the graph:

- **Possible types** – each node is a possible supertype of the class (which is situated at the bottom; not shown).
- **Available types** are shown in the *types* section on a node. Subtyping relations between types are indicated by UML-style subtyping arrows.

- **Variables and methods**. Each variable or method typed with one of the type(s) under consideration is included in the graph. Assignments between these elements are shown with red arrows.
    - **Declared placement**. A variable or method is shown in the *declared placement* section of the node containing the declared type of the variable or method.
    - **Ideal placement**. If different from the declared placement, a variable or method is shown in the *ideal placement* section of the node which corresponds to the set of members (transitively) invoked on this variable or method.

The quality of the *variable* and *method declarations* (i.e. the matching between types and their usage contexts) is shown by the colours of the background of the node. A green colour represents the use of context specific types, while a red colour signals a mismatch between types and usage contexts.

Selecting variables or methods in the graph further enriches the display:

- A line is drawn connecting the ideal and declared placements of an element (if different).
- Additional lines are drawn connecting all elements which the current element is being assigned to (transitively).

This data may be used to detect smells in the code and take appropriate action. The following section will detail this.

3. VISUAL REFACTORINGS

By analyzing a type in the TAA view, the developer has complete overview of the usages of this type. The annotations on the supertype lattice suggest a number of ways of improving the typing situation; specifically, the arrangements of types and variables/methods visualize code smells which can be removed by applying refactorings.

The following table associates design problems in the code, the way these problems show up in the visual view (as smells), and the actions to be taken by the developer to deal with those problems. Later on, we will present refactorings for executing these actions.

Problem	**Smell**	**Action**
No interfaces available for a context	Nodes in the graph with ideal placement of variables/methods, but without interfaces	Extract interface and redeclare variables/ methods with new interface
Poorly designed interface	Ideal placement of variables/methods swarming around existing interfaces	Move existing interface up or down in hierarchy
Two interfaces for the same purpose	Two interfaces share the same/neighbouring nodes, each with ideally placed variables/methods	Merge interfaces
Superfluous interface	Interface present in a node without declared placements; no ideal placements in vicinity	Remove interface from hierarchy
Interface is not (yet) used	Interface is present in a node with ideally but no declared placements	Redeclare variables /methods with existing interface

Table 1: Code Smells

As can be seen from the table above, the TAA suggests a number of actions to be taken as a result of identified smells. These actions are implemented as refactorings. In line with our approach of putting the developer in charge, these refactorings may be selected as (semantically) appropriate by the user.

Contrary to existing refactorings, the ones invoked from the TAA in general do not require further dialog-based parameterization – all information required for a refactoring is already available in the graph and the way the user invokes the refactoring in a visual way. These visual start procedures are shown in Table 2:

Refactoring	**Visual start procedure**
Extract interface	Alt + Drag an interface to another, higher node in the graph
Move interface in hierarchy	Drag an interface to another node in the graph (up or down)
Remove interface from hierarchy	Select an interface, select delete
Merge interfaces	Select two interfaces, select merge
Redeclare declaration elements (transitively, i.e. following assignments)	Select a variable or method, select redeclare

Table 2: Refactorings

While the *Extract Interface* refactoring is already available as-is in many tools, the others have been either adapted or specifically written for the TAA.

4. SUMMARY

In this paper, we have described our approach to visual refactoring. The TAA tool aids the developer by providing valuable information about the typing situation in the code – in itself suitable for program understanding – and thereby suggests refactorings whose effect is more predictable and which can be executed directly from the visual view. The results of the refactorings are likewise directly shown in the graph.

In the future, we will investigate ways of further improving the user interface and add more refactorings to the TAA.

5. REFERENCES

[1] E Gamma, R Helm, R Johnson, J Vlissides *Design Patterns - Elements of Reusable Software* (Addison-Wesley, 1995).

[2] F Steimann "The Infer Type refactoring and its use for interface-based programming" *JOT* 6:2 (2007) 67–89.

[3] F Steimann, P Mayer "Patterns of interface-based programming" *JOT* 4:5 (2005) 75–94.

[4] F Steimann, P Mayer "Type Access Analysis: Towards informed interface design" in: *TOOLS* (2007) to appear.

This work has been partially sponsored by the project SENSORIA, IST-2005-016004.