Declared Type Generalization Checker: An Eclipse Plug-In for Systematic Programming with More General Types

Markus Bach, Florian Forster, and Friedrich Steimann

Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
bach.markus@gmx.net, florian.forster@fernuni-hagen.de, steimann@acm.org

Abstract. The Declared Type Generalization Checker is a plug-in for Eclipse's Java Development Tools (JDT) that supports developers in systematically finding and using better fitting types in their programs. A type *A* is considered to fit better than a type *B* for a declaration element (variable) *d* if *A* is more general than *B*, that is, if *A* provides fewer members unneeded for the use of *d*. Our support comes in the form of warnings generated in the Problem View of Eclipse, and associated Quick Fixes allowing elements to be re-declared automatically. Due to the use of Eclipse extension points, the algorithm used to compute more general types is easily exchangeable. Currently our tool can use two publicly available algorithms, one considering only supertypes already present in a project, and one computing new, perfectly fitting types.

1 The Problem: Too Strong Coupling Due to Overly Specific Types

A class *C* is coupled to a type *B* if one or more declaration elements of *C* (i.e., fields, formal parameters, local variables, or methods with non-void return types) are declared with *B* as their type. Even though coupling between types cannot be eliminated completely (because without any coupling a type would be isolated from the rest of the system and therefore useless [1]), there is often a certain amount of unnecessary coupling which can be reduced in many cases by using a more general type than *B*. In fact, unnecessary coupling between *C* and *B* arises when a declaration element *d* in *C* is declared with *B* as its type and *B* offers more members than actually needed by *d*. In [2] we have shown that developers rarely use the best fitting type available in a program for typing declaration elements, and that by introducing new, better fitting types unnecessary coupling can be reduced to a minimum. However, we believe that developers cannot be blamed for not using more general types in a project as long as proactive tool support for indicating where which types can be used is lacking: programmers tend to think of their objects more in terms of the classes from which they are instantiated, and less in terms of the generalizations they posses (which are often unknown, or at least not known to be useable in a given context).

2 The Solution: The Declared Type Generalization Checker

To support developers in becoming aware of — and in using — more general types, we implemented a tool, called the Declared Type Generalization Checker, as a plug-in for the Eclipse Java Development Tools (JDT) [6, 7][1]. This plug-in provides a new type of warning for the Problem View, which informs developers of unnecessary coupling arising from overly specific declaration elements (i.e., elements declared with types providing more members than actually needed). At the same time, the plug-in extends Eclipse's Quick Fixes by one that lets programmers re-declare elements with better fitting types. To compute these types and to perform the re-declaration, currently one out of two available algorithms for type generalization (and their associated refactorings) can be selected in the project properties tab of the plug-in.

2.1 Generation of Warnings

The Declared Type Generalization Checker is implemented as a builder that, if activated in a project's properties, is automatically started after each compilation of the project. Since compilation in Eclipse is itself implemented as a builder, the Declared Type Generalization Checker can take advantage of Eclipse's incremental build process — in particular, after a change only the compilation units affected by that change are rebuilt. This helps shorten checking times considerably (cf. Section 3).

The builder visits each declaration element of a compilation unit and invokes the algorithm selected for checking for possible generalizations (see Section 4). The results of each check are communicated to the IDE using its standard interface for builders.

2.2 Provision of Quick Fixes

With each warning a Quick Fix can be associated that triggers a refactoring introducing a more general type (thus resolving the warning). Whether such a Quick Fix exists depends on the algorithm chosen to generate the warning, which is selected in the project property settings. Currently, two such algorithms are available.

2.3 Algorithms Computing More General Types

For every project, the programmer can choose the algorithm the checker uses to generate the warnings. Currently, the available algorithms are the standard algorithms delivered with their corresponding refactorings, which also provide the Quick Fixes.

Generalize Declared Type. Generalize Declared Type is a standard refactoring of Eclipse distributed with JDT. After invocation of the refactoring on a declaration element *d* the developer is presented the type hierarchy for the declared type of *d*. In

[1] http://www.eclipse.org

this hierarchy, every supertype that can be used in the declaration of *d* (because it includes all members required from *d*) is highlighted and can be selected as the new type of *d*. Note that this refactoring does not necessarily reduce coupling to a theoretical minimum, as the new type may still contain excessive members, and the perfect generalization may not (yet) have been introduced (and therefore is not among the presented supertypes). Nevertheless, as [2] has shown, even if generalizations are available in a project, they are often not used.

Our Declared Type Generalization Checker uses the type inference algorithm employed by Generalize Declared Type to check whether a more general type is available (the basis for a warning); also, it launches the refactoring itself as the corresponding Quick Fix.

Infer Type. So-called type inference can compute type annotations for program elements independently of whether or how they are actually typed [3–5]. We designed our own type inference algorithm for Java [4] specifically to compute the most general type that can be used for a declaration element, and this independently of the types that already exist. Our algorithm is the basis of a new refactoring, called Infer Type[2], which can be characterized as an automatic version of the Extract Interface refactoring distributed with Eclipse's JDT and other Java IDEs. Since the types computed by Infer Type are always maximally general (meaning that no member can be removed without causing a static type error), types using only inferred types for their declaration elements are always maximally decoupled.

The type inference algorithm underlying Infer Type is used by our Declared Type Generalization Checker as an alternative to that of Generalize Declared Type in exactly the same way as described above.

3 Performance Evaluation

Checking every declaration element of a program for the availability of a more general type is a time-consuming task. To get an impression of how the systematic search for type generalizations influences the development cycle, we performed the measurements summarized in the following table.

PROJECT	NUMBER OF DECLARATION ELEMENTS	ALGORITHM			
		Generalize Declared Type		*Infer Type*	
		time	*warnings*	*time*	*warnings*
JUnit 3.8.1	1501	≈ 3.5 mins	205	≈ 8 mins	315
JHotDraw 6.0b1	7788	≈ 42 mins	1230	–	–

These times (obtained on a ThinkPad run at 2 GHz) may appear unacceptable, especially for Infer Type, but since they refer to full builds, they rarely occur in practice. What we found instead is that for average change/build cycles, the overhead incurred by the Declared Type Generalization Checker is reasonable. As for Infer Type, we hope to be able to present a more efficient implementation soon (see Section 5).

[2] http://www.fernuni-hagen.de/ps/prjs/InferType/

4 Extending the Declared Type Generalization Checker

As mentioned above, the algorithm used to compute more general types for declaration elements is variable. In fact, our tool accommodates further extensions, by allowing one to add other algorithms and refactorings. The corresponding extension point requires implementations of three methods, namely `boolean checkType(...)`, `boolean hasResolution()`, and `IMarkerResolution2 getResolution()`. The first, `checkType`, answers for a given declaration element and its declared type whether a better matching type exists so that a corresponding warning can be generated. If it does, `hasResolution` tells the plug-in whether the extension can also offer a Quick Fix to resolve the issue (which is the case for both extensions currently in offered). If so, the method `getResolution` delivers an object that, through its `run` method, redeclares the declaration element in question (in the current extensions by starting a refactoring).

5 Availability

The Declared Type Generalization Checker can be installed from the update site http://www.fernuni-hagen.de/ps/prjs/DTGC/update/. It depends on the availability of the Generalize Declared Type refactoring (which is part of the standard distribution) and optionally also that of Infer Type (which is part of the Yoxos[3] distribution, but can also be installed separately from the link given in Footnote 2).

We are currently working on a new implementation of Infer Type that utilizes Eclipse's type constraint framework and that can handle Java generics. Once available, it will be offered as an alternative extension to our Declared Type Generalization Checker.

References

1. E Berard *Essays on Object-Oriented Software Engineering* (Prentice-Hall 1993).
2. F Forster "Cost and benefit of rigorous decoupling with context-specific interfaces" in: *Proceedings of the 4th International Conference on Principles and Practices of Programming in Java* (2006) 23–30.
3. J Palsberg, MI Schwartzbach "Object-oriented type inference" in: *Proceedings of OOPSLA* (1991) 146–161.
4. F Steimann, P Mayer, A Meißner "Decoupling classes with inferred interfaces" in: *Proceedings of the 2006 ACM Symposium on Applied Computing* (2006) 1404–1408.
5. T Wang,SF Smith "Precise constraint-based type inference for JAVA" in: *Proceedings of ECOOP* (2001) 99–117.
6. D Bäumer, E Gamma, A Kiezun "Integrating refactoring support into a Java development tool" in: *OOPSLA'01 Companion* (2001).
7. E Gamma, K Beck *Contributing to Eclipse* (Addison-Wesley Professional 2003).

[3] www.yoxos.com