Is Structural Subtyping Useful? An Empirical Study

Donna Malayeri and Jonathan Aldrich

Carnegie Mellon University, Pittsburgh, PA 15213, USA,
{donna, aldrich}@cs.cmu.edu

Abstract. Structural subtyping is popular in research languages, but all mainstream object-oriented languages use nominal subtyping. Since languages with structural subtyping are not in widespread use, the empirical questions of whether and how structural subtyping is useful have thus far remained unanswered. This study aims to provide answers to these questions. We identified several criteria that are indicators that nominally typed programs could benefit from structural subtyping, and performed automated and manual analyses of open-source Java programs based on these criteria. Our results suggest that these programs could indeed be improved with the addition of structural subtyping. We hope this study will provide guidance for language designers who are considering use of this subtyping discipline.

1 Introduction

Structural subtyping is popular in the research community and is used in languages such as O'Caml [15], PolyToil [6], Moby [11], Strongtalk [5], and a number of type systems and calculi (e.g., [7, 1]). In the research community, many believe that structural subtyping is beneficial and is superior to nominal subtyping. But, structural subtyping is not used in any mainstream object-oriented programming language—perhaps due to lack of evidence of its utility. Accordingly, we ask: what empirical evidence could show that structural subtyping can be beneficial?

Let us consider the characteristics that a nominally-typed program might exhibit that would indicate that it could benefit from structural subtyping. First, the program might systematically make use of a subset of methods of a type, with no nominal type corresponding to this method set. A particular such implicit type might be used repeatedly throughout the program. Structural subtyping would allow these types to be easily expressed, without requiring that the type hierarchy of the program change.

Second, there might be methods in two different classes that share the same name and perform the same operation, but that are not contained in a common nominal supertype. This could happen due to oversight, or perhaps the need did not yet exist to call that method in a generic manner for both classes. Alternatively, perhaps such a need did exist, but programmers resorted to code duplication rather than refactoring the type hierarchy. With structural subtyping, the two classes would automatically share a common supertype.

Or, programs might use the Java reflection method `Class.getMethod` to call a method with a particular signature in a generic manner. Structural subtyping provides

exactly this capability, with no need for reflection. Finally, what might a class do if it can only support a subset of its declared interface, but no such super-interface can be defined (due to library use)? One implementation strategy is to have some of its methods always throw an `UnsupportedOperationException`. In contrast, with structural subtyping, the intended structural super-interface could simply be used.

With these characteristics in mind, we examined up to 29 open-source Java programs, using both manual and automated analyses (in the case of manual analyses, a subset of the subject programs were considered). Each aimed to answer one question: are nominally-typed programs using implicit structural types? We found that indeed they were; representing these types explicitly could therefore be advantageous.

In our empirical evaluation, we sought to answer the following questions:

1. Does the body of a method use only a subset of the methods of its parameters? If so, structural types could ease the task of making the method more general. (Sect. 3)
2. If structural types are inferred for method parameters, do there exist types that are used repeatedly, suggesting that they represent a meaningful abstraction? (Sect. 3.3)
3. How many methods always throw "unsupported operation" exceptions? In such cases, classes support a structural supertype of the class type. (Sect. 4)
4. Do there exist *common methods*—methods with the same name and signature, but that are not contained in a common supertype of the enclosing classes? (Sect 5.1)
5. How many common methods represent an accidental name clash? (Sect 5.2)
6. Can structural subtyping reduce code duplication? (Sect. 5.3)
7. Is there synergy between structural subtyping and other proposed language features, such as multimethods? (Sect. 6)
8. Do programs use reflection where structural types would be preferable? (Sect. 7)

Thus, we considered a variety of facets of existing programs. While none of these aspects is conclusive on its own, taken together, the answers to the above questions provide evidence that even programs written with a nominal subtyping discipline could benefit from structural subtyping. This study provides initial answers to the above questions; further study is needed to fully examine all aspects of some questions, particularly questions 6 and 7. Additionally, this study considers only the potential benefits of structural subtyping, while there are situations where nominal types are more appropriate [23, 17].

To our knowledge, this is the first systematic corpus analysis to determine the benefits of structural subtyping. This paper makes the following contributions: (1) identification of a number of characteristics in a program that suggest the use of implicit structural types; and (2) results from automated and manual analyses that measure the identified characteristics.

2 Corpus and Methodology

For this study, we analyzed the source code of up to 29 open-source Java applications (details, including version numbers, are provided in [18]). The full set of subject programs were used for the automated analyses; due to practical considerations, for manual analysis we chose a subset thereof (ranging from 2 to 8 in size). The applications were chosen from the following sources: popular applications on SourceForge, Apache Foundation applications, and the DaCapo benchmark suite. The full set of programs range

from 12 kLOC to 161 kLOC, and for both kinds of analysis we selected programs based on size, type (library vs. standalone program) and domain (selecting for variety). For some of the manual analyses, we favored applications with which we were familiar, as this aided analysis. All of the manual analyses, including the subjective analyses, were performed by the first author. The methodology for each analysis is described in the corresponding section; further details are available in the companion technical report [18].

For space considerations, in this paper we have omitted data from the 10 smallest applications that were not already the subject of a manual analysis (in order that the manual analysis subject programs be a subset of the included automated analysis programs). (There is one exception: Azureus was used in one manual analysis (Sect. 5.3), but was too large for our whole-program automated analyses.) We refer the reader to the companion technical report for full results [18].

3 Inferring Structural Types for Method Parameters

It is considered good programming practice to make parameters as general as the program allows. Bloch, for example, recommends favoring interfaces over classes in general—particularly so in the case of parameter types [3]. An analogous situation arises in the generic programming community, where it is recommended that generic algorithms and types place as few requirements as possible on their type parameters (e.g., what methods they should support) [22].

Bloch acknowledges that sometimes an appropriate interface does not exist (e.g., class `java.util.Random` does not implement any non-marker interfaces). In such a case the programmer is forced to use classes for parameter types—even though it is possible that multiple implementations of the same functionality could exist [3]. This is a situation where structural subtyping could be beneficial, as it allows programmers to create supertypes after-the-fact.

As it is impossible to retroactively implement interfaces in Java, we hypothesized that method parameter types are often overly specific, and sought to determine both (1) the degree and (2) the character of over-specificity. To answer question (1), we performed an automated whole-program analysis to infer structural types for method parameters. Methodology and quantitative results are described in Sect. 3.1. To properly interpret this data, however, we must consider question (2). Accordingly, we manually examined the inferred structural types from the previous analysis and considered the qualitative question of whether changing a method to have the most general structural type could potentially improve the method's interface (Sect. 3.2). Across all applications, we also counted occurrences of inferred structural types that were supertypes of classes and interfaces of the Java Collections Library. Of these, in Sect. 3.3 we present those structural types that a client might plausibly wish to implement while *not* simultaneously implementing a more specific nominal type (e.g., `Collection`, `Map`, etc.).

3.1 Quantitative Results

Our analysis infers structural types for method parameters, based on the methods that were actually called on the parameters. (For example, a method may take a `List` as an

argument, but may only use the `add` and `iterator` methods.) The analysis, a simple inter-procedural dataflow analysis, re-computes structural types for each parameter of a method until a fixpoint is reached. Details of the algorithm are described in the companion technical report [18]. Structural types were not inferred in the following cases: calls to library methods, assignments to fields and local variables, uses of primitive types, uses of types such as `String` and `Object`, and cases where the inferred structural type would have a non-public member.

The analysis is conservative; in the case where a parameter is not used (or only methods of class `Object` are used), no structural type is inferred for it. (A parameter may be unused because (a) it is expected that overriding methods will use the parameter, or (b) because the method may make use of the parameter when the program evolves, or (c) because it is no longer needed, due to changes in the program.) In the case of method overriding, the analysis ensures that the same structural types are inferred for corresponding parameters in the method family.

Our results suggest that a refactoring that infers structural types is of limited utility unless structural types are used in libraries. On average, only 15% of parameters could have a structural type inferred. The remaining parameters fell into the following three categories: an average of 14% were either a primitive type or were unused; an average of 25% were uses of `Object`, `String` or `StringBuffer`; an average of 49% were parameters on which a library method was transitively called, or were stored to fields/local variables, or which called non-public instance methods. Thus, our results do not paint a complete picture, though the fact that several of the subjects were libraries does increase confidence in our findings. In our future work, we plan to analyze some of the libraries used by the subject applications, in order to increase the percentage of inferrable parameters.

Analysis results for 19 programs are displayed in Table 1. For example, in Ant, 16.5% of parameters could have a structural type inferred. Of these, 98.6% of the parameters were declared with an *overly specific* nominal type (i.e., the nominal type contained more methods than were actually needed). For only 2.4% of the inferred parameters did a corresponding nominal type exist that would make the parameter type as *general* as possible (i.e., a nominal type that contained only those methods transitively called on the object). There were an average of 2.0 methods in the inferred structural types, while there were 33.3 methods in the corresponding nominal types. Finally, there was a median of 1 structural type inferred for each nominal type in the program, and a maximum of 27 structural types.

Several conclusions can be drawn from the data. First, most parameter types for which a structural type can be inferred (15% on average) are overly specific (94% on average). Moreover, for most inferred parameters (91% on average), no nominal type existed in the program that was as general as possible.

Second, inferred structural types do not have many methods (3.5 on average), while the corresponding nominal types have quite a few methods (37.8 on average). This shows that there is quite a large *degree* of over specificity—more than a full order of magnitude—in addition to the large percentage of overly specific parameters. This is likely due to the overhead of naming and defining nominal types, as well as the lack of retroactive interface implementation. We also found that when nominal types were as general as possible, they had very few members—one or two on average. This is in

Table 1. Results of running structural type inference. *Percent inferrable* is the percentage of parameters that could have a structural type inferred for them (i.e., where neither library methods were transitively called, nor was the parameter unused, etc.), *percent overly specific* is the percentage of the inferrable parameters that have an overly specific nominal type, *percent structural needed* is the percentage of the inferrable parameters for which a most general nominal type does *not* exist, *average methods per structural type* is the average number of methods in the inferred structural types, *average methods per nominal type* is the average number of methods in nominal types that appear as parameter types (including inherited methods), and *median/maximum structural types per nominal* are the median and maximum, respectively, of the number of inferred structural types corresponding to each nominal type.

	LOC	% inferrable	% overly specific	% structural needed	Avg methods/ structural type	Avg methods/ nominal type	Struct types/nominal	
							median	max
Ant	62k	16.5%	98.6%	97.6%	2.0	33.3	1	27
Apache collect.	26k	10.9%	90.1%	83.6%	1.9	9.9	1	11
Areca	35k	15.0%	99.1%	97.0%	2.6	35.1	1	35
Cayenne	95k	21.1%	96.8%	93.0%	2.4	27.6	2	27
Columba	70k	12.0%	99.6%	98.7%	2.0	55.3	1	19
Crystal	12k	15.9%	97.7%	92.5%	3.5	13.7	1	19
hsqldb	62k	7.8%	99.4%	99.4%	1.9	50.8	2	34
jEdit	71k	6.7%	95.1%	95.1%	2.2	105.2	1	20
JFreeChart	93k	17.2%	97.4%	94.4%	3.2	53.4	1	35
JHotDraw	52k	17.5%	100.0%	99.6%	2.7	55.2	2	19
JRuby	86k	24.6%	97.4%	96.7%	4.4	66.1	1	85
LimeWire	97k	16.7%	98.4%	94.8%	2.1	35.2	1	21
Log4j	13k	17.1%	96.7%	95.0%	1.9	54.9	1	6
Lucene	24k	9.2%	80.5%	77.4%	1.6	9.9	1.5	8
OpenFire	90k	20.6%	99.3%	99.2%	2.5	34.3	1	45
PLT collections	19k	10.3%	49.7%	51.0%	1.6	15.2	1	25
Smack	40k	13.7%	100.0%	90.8%	4.6	25.2	1	13
Tomcat	126k	13.4%	96.7%	96.3%	4.5	34.4	2	32
Xalan	161k	12.4%	95.5%	95.1%	3.1	55.7	1	16
Average		**15.0%**	**93.5%**	**91.3%**	**3.5**	**37.8**	**1.2**	**23.4**

accordance with previous work which found that interfaces are generally smaller than classes [25].

Next, for a given nominal type, there were not many corresponding structural types (2.5 on average, a median of 1.2). The data followed a power law distribution, with an average maximum of 24; that is, small values were heavily represented, but there were also a few large values. The low median suggests that the overhead of naming structural types is not necessarily high; it is plausible that programmers would be able to name and use structural types for around half of the nominal parameter types.

Finally, if we were to define new interfaces everywhere possible, the average increase in the number of interfaces is 313%, the median is 287%, and the maximum is 1000%. This illustrates the infeasibility of defining new nominal types for the inferred structural types. Note that we considered only those interfaces for which the `implements` clause of a class could be modified (i.e., those classes in the program's source); in general, the situation is even worse, as programmers may wish to define new supertypes for types contained in libraries.

3.2 Qualitative Results

Though our results show that many parameters are overly specific, we do not necessarily recommend that every parameter be made as general as possible. This is because a method might be currently only using a particular set of methods, but later code modifications may make it necessary to use a larger set; a more general type could hinder program evolution. On the other hand, more general types make methods more reusable, which aids program evolution. For this reason, a refactoring to structural types (or even structural type inference) cannot be a fully automated process—programmers must consider each type carefully, keeping in view the kinds of program modifications that are likely to occur. Additionally, for some structural types, there may ever be only one corresponding nominal type, in which case using a structural type is of limited utility.

Accordingly, we considered the empirical question of whether changing a given method to have the most general structural types for its parameters would make the method more general in a way that could improve the program. To determine this, we inspected each method and asked two questions. First, does the inferred parameter type S generalize the abstract operation performed by the method, as determined by the method name? Second, does it seem likely that there would be multiple subtypes of S?

We studied two applications: Apache Collections (a collections library) and Crystal (a static analysis framework). Of methods for which a structural type was inferred on one or more parameters, we found that 58% and 66%, respectively, would be generalized in a potentially useful manner if the inferred types were used.

For example, in Apache Collections, in the class `OnePredicate` (a predicate class that returns true only if one of its enclosing predicates returns true), the factory method `getInstance(Collection)` had the structural type `{iterator(); size();}` inferred for its parameter. This would make the method applicable to any collection that supported only iteration and retrieving the collection size, even if it didn't support collection addition and removal methods. There were 25 other methods in the library that used this structural type. Another example is the method `ListUtils.intersection` which takes two `List` objects. However, the first `List` need only have a `contains` method, and the second `List` need only have an `iterator` method (for this latter parameter, the interface `Iterable` could be used). There were also 8 methods that took an `Iterator` as a parameter, but never called the `remove` method. With a structural type for the method, the type would clearly specify that a read-only iterator can be passed as an argument.

In Crystal, two methods took a `Map` parameter that used only the `get` and `put` methods. Converting the method to use this structural type would make it applicable to a map that did not support iteration (such a type exists in Apache Collections, for example). Also, there were 11 methods that use only the methods `getModifiers()` and `getName()` on an `IBinding` object (an interface in the Eclipse JDT). Replacing the nominal type with a structural type would allow the program to substitute a different "bindings" class that supported only those two methods.

Of course, for some of these structural types, there may not be a large number of classes that implement its methods but not all of the methods of a more specific nominal type, e.g., `Collection`. However, we believe that all of the aforementioned types represent meaningful abstractions. Furthermore, since it is conceivable that a programmer

may define a class implementing that abstraction, using these more general types would increase the applications' reusability.

Translation to Whiteoak Using the inference algorithm, we also developed an automated translation of programs from Java to Whiteoak [14], a research language that extends Java with support for structural subtyping. We performed this translation on two programs: Apache Collections and Lucene, validating the results of the analysis and demonstrating its practical use.

3.3 Uses of Java Collections Library

We examined the inferred structural types that were generalizations of types in the Java Collections Library. Over all applications, there were 67 distinct types in total, though not all appeared to express an important abstraction. We made a conservative subjective finding that at least 10 of these types *were* potentially useful; these are displayed in Table 2, along with a description of possible implementations. The relatively high number of occurrences of each of these structural types further suggests their utility, even though the types contain few methods. It further shows that programs routinely make use of types that the library designers either did not anticipate or chose not to support.

Table 2. Uses of Java Collections classes across 19 programs, as inferred using the parameter structural type inference. (Erasures are used in lieu of generic types.)

Methods in type	Uses	Description
`get(Object); containsKey(Object);`	168	Read-only non-iterable map; for instance, a read-only hashtable
`iterator(); isEmpty(); size();`	114	Read-only iterable collection that knows its size; for instance, a read-only list
`add(Object); addAll(Collection);`	101	Write-only collection; for instance, a log
`put(Object, Object);`	55	Write-only map
`hasNext(); next();`	28	Read-only iterator
`contains(Object);`	21	Read-only collection that does not support iteration; for instance, a read-only hashset
`get(Object); put(Object, Object);`	15	Non-iterable map; for instance, a hashtable
`contains(Object); iterator(); size();`	11	Read-only iterable collection that knows its size and can be polled for the existence of an element; for instance, an iterable hashset
`add(Object); contains(Object); iterator(); size();`	10	Same as above, but that also supports adding elements
`iterator(); size(); toArray(Object[]);`	8	Read-only collection that can be converted to an array; for instance, a read-only array

In summary, the data shows that programs make repeated use of many implicit structural types. A language that would allow defining these types explicitly could be beneficial, as it can help programmers make their methods more generally applicable.

3.4 Related work

Forster [12] and Steimann [24] have described experience using the *Infer Type* refactoring, which generates new interfaces for inferred types and replaces uses of overly

Table 3. A selection of the structural interfaces "implemented" by classes in the subject programs once methods unconditionally throwing an `UnsupportedOperationException` are removed. (Actual method sets are omitted to conserve space.)

	Number of classes
Read-only `Iterator`	50
Read-only `Collection`	19
Read-only `Map`	9
Read-only `Map.Entry`	6
Read-only `ListIterator`	6
`Collection` supporting everything but removal	5
`Map` supporting everything but removal	4
`Collection` supporting only read and removal methods	1
`Collection` supporting iteration, addition, and size only	1
`ListIterator` supporting read, add, and remove (but not `set()`)	1
`ListIterator` supporting only read and `set()` operation	1
`Map` supporting read, put, and size only	1
`Map` supporting read and put, but not size or removal	1
`Map` supporting everything but `entrySet()`, `values()` and `containsValue()`	1

specific types with these interfaces. This analysis is more general than ours, because it considers all type references, not just parameter types. However, the refactoring is limited by the fact that classes in libraries cannot retroactively implement new interfaces. Steimann found that when applying this refactoring, the number of total interfaces almost quadrupled—an increase of 369%.[1]

4 Throwing "Unsupported Operation" Exceptions

In the Java Collections Library, there are a number of "optional" methods whose documentation permits them to always throw an exception. This decision was due to the practical consideration of avoiding an "explosion" of interfaces; the library designers mentioned that at least 25 new interfaces would be otherwise required [19].

To determine if such super-interfaces would be useful in practice, we totalled the methods in the subject programs that unconditionally throw an `UnsupportedOperationException`. The program that had the most such methods was Apache Collections: there were 148 methods that unconditionally throw the exception (out of 3669 total methods, corresponding to 4%). Next, we considered those methods that were overriding a method in the Java Collections Library. To encode these optional methods directly would require 18 additional interfaces. There are only 27 interfaces defined in the library, so this represents a 67% increase. Note that this is a conservative estimate, as we did not consider interactions between classes (e.g., an `Iterable` returning a read-only `Iterator`). A selection of these structural super-interfaces are summarized in Table 3. For instance, there were 50 iterator classes that did not support the `remove()` operation, and 19 subclasses of `Collection` that supported a read-only interface.

[1] This differs slightly from our average of 313%, though this difference is likely due to the fact that Steimann considered only two applications.

Note that, with the exception of the read-only iterator, the sets of interfaces in Tables 3 and 2 are distinct from one another (though some are subtypes). This is likely due to the fact that different applications use different subsets of the methods of a class.

Structural subtyping could be helpful for statically ensuring that "unsupported operation" exceptions cannot occur, as it would allow programmers to express these superinterfaces directly.

5 Common Methods

In our experience, there are situations where two types share an implicit common supertype, but this relationship is not encoded in the type hierarchy. For example, suppose two classes both have a `getName` method with the same signature, but there does not exist a supertype of both classes containing this method. We call `getName`, and methods like it, *common methods*. Common methods can occur when programmers do not anticipate the utility of a shared supertype or when two methods have the same name, but perform different operations; e.g., `Cowboy.draw()` and `Circle.draw()` [16].

Accordingly, this section aims to answer three questions: (1) how often do common methods occur, (2) how many common methods represent an accidental name clash, and (3) do common methods result in code clones.

5.1 Frequency

We performed a simple whole-program analysis to count the number of common methods in each application. Only public instance methods were considered (resulting in slightly different data than that previously presented [17]). Results are in Table 4. Overall, common methods comprise an average of 19% of all public instance methods. That is, for 19% of methods, there existed another method with the same name and signature and the method was not contained in a common supertype of the enclosing types.

We also computed the number of types that share at least two common methods with another type; there were an average of 9% of such types. These are the cases in which a structural supertype is most likely to be useful. This high percentage indicates that there are a number of implicit structural types in most applications.

For example, in Apache Collections, `UnmodifiableSortedMap` and `OrderedMap` share the methods `firstKey()` and `lastKey()`. And, `AbstractLinkedList` and `SequencedHashMap` share the methods `getFirst()` and `getLast()`. Finally, `BoundedMap` and `BoundedCollection` have the common methods `isFull()` and `maxSize()`.

In Lucene, a document indexing and search library, `RAMOutputStream` and `RAMInputStream` both support the `seek()`, `close()`, and `getFilePointer()` methods, which might be useful to move to a supertype. Also, the classes `PhraseQuery` and `MultiPhraseQuery` both support the methods `add(Term)`, `getPositions()`, `getSlop()`, and `setSlop(int)`.

5.2 Accidental Name Clashes

Of course, to interpret this data, we must consider cases where the common methods have the same *meaning*, and where callers are likely to call the methods with the same

Table 4. Common methods for each application. *Number of types* indicates the total number of types in the application, *types with greater than one common method* is the number of types that share more than one common method, *percentage* is the percentage of this compared to the total number of types, *percent common methods* is the percentage of public instance methods that is a common method, and *average number of classes per common signature* is the average number of classes for each common method signature.

	LOC	Number of types	Types with >1 common method	Percentage	% common methods	Avg # classes/ common signature
Ant	62k	945	65	6.9%	31.3%	3.7
Apache Collections	26k	550	19	3.5%	7.3%	2.7
Areca	35k	362	30	8.3%	15.4%	2.7
Cayenne	95k	1415	104	7.3%	18.1%	2.8
Columba	70k	1232	48	3.9%	17.3%	3.1
Crystal	12k	211	4	1.9%	5.1%	2.9
hsqldb	62k	355	31	8.7%	19.5%	2.6
jEdit	71k	880	40	4.5%	11.7%	2.5
JFreeChart	93k	789	301	38.1%	39.5%	3.9
JHotDraw	52k	616	59	9.6%	19.0%	2.8
JRuby	86k	997	83	8.3%	15.6%	3.1
LimeWire	97k	1689	88	5.2%	17.7%	3.1
log4j	13k	201	4	2.0%	13.6%	2.4
Lucene	24k	398	21	5.3%	13.4%	2.6
OpenFire	90k	1039	110	10.6%	19.0%	3
plt collections	19k	812	60	7.4%	7.5%	2.8
Smack	40k	847	115	13.6%	23.5%	3.3
Tomcat	126k	1727	234	13.5%	32.6%	3.6
xalan	161k	1223	94	7.7%	16.1%	2.9
Average				**9.3%**	**19.0%**	**2.9**

purpose in mind. If two methods have the same meaning, it might be useful to define a structural type consisting of that method. Two methods are defined as "having the same meaning" if they perform the same abstract operation, taking into account (a) the semantics of the method, and (b) the semantics of the enclosing types. This determination was made by examining the source code, using javadoc where available.

We studied two applications: Apache Collections and Lucene. In Collections, under condition (a), there were no methods that had the same signature but performed different abstract operations. However, there were 2 cases (1% of all common methods) where the methods had the same meaning, but the enclosing classes did not appear to be semantic subtypes of some common supertype containing that method; i.e., condition (b) was not satisfied. For example, the classes `ChainedClosure` and `SwitchClosure` both had a `getClosures()` method, but `ChainedClosure` calls each of these closures in turn, while `SwitchClosure` calls that closure whose predicate returns `true`.

In Lucene, there were 42 instances of methods that had the same signature, but did not have the same meaning (19% of all common methods). In 32 of these cases, the methods were actually performing a different abstract operation. For example, `HitIterator.length()` returned the number of hits for a particular query, while `Payload.length()` returned the length of the payload data. An additional 10 cases did not satisfy condition (b) above. For example, in a high-level class `IndexModifier`, there were several cases where a method *m* performed some operation, then called

```
// repeated exactly in 19 classes
if (property == EXPRESSION_PROPERTY) {
  if (get) {
    return getExpression();
  } else {
     setExpression((Expression) child);
     return null;
  }
}
```

(a)

```
private InlineMethodRefactoring(ICompilationUnit unit,
   MethodInvocation node, int offset, int length)
{
  this(unit, (ASTNode)node, offset, length);
  fTargetProvider= TargetProvider.create(unit, node);
  fInitialMode= fCurrentMode= Mode.INLINE_SINGLE;
  fDeleteSource= false;
}

private InlineMethodRefactoring(ICompilationUnit unit,
   SuperMethodInvocation node, int offset, int length)
{
  ... // same method body as above
}
```

(b)

Fig. 1. Examples of code duplication in the Eclipse JDT. Structural subtyping could eliminate this duplication.

`IndexWriter.`m, the latter performing a lower-level operation. So, the semantics of the methods were similar, but the semantics of each class was different.

Overall, the data is very promising, as it indicates that most common methods have the same meaning and would benefit from being contained in a structural supertype—90% on average, across both applications. Structural subtyping would allow these methods to be called in a generic manner, without the need to create additional interfaces.

5.3 Code Clones

We hypothesized that common methods can lead to code clones, as there is a common structure that is not expressed in the type system. To determine this, we examined two applications: Eclipse JDT and Azureus.

In the Eclipse Java Development Tools (JDT), many AST classes have methods `getExpression` and `setExpression`, but these methods are not contained in a supertype. As a result, there is repeated code in each of these classes, e.g., related to reading and storing these attributes in a generic internal AST map. The code for this is shown in Fig. 1a. This could be re-written using structural subtyping by writing a helper method taking a parameter of structural type `{ getExpression; setExpression; }`. The repeated code would then be replaced with something similar to `getSetExpr(this, get, child)`. A similar situation occurs with the methods `typeArguments()` and `getBody()`.

Similarly, the classes `FieldAccess` and `SuperFieldAccess` have no superclass other than `Expression`. The same problem occurs with `MethodInvocation` and `SuperMethodInvocation`, and `ConstructorInvocation` and `SuperConstructorInvocation`. We found 44 code clones involving these types (though some were only a few lines long). An example of a code clone involving `MethodInvocation` and `SuperMethodInvocation` appears in Fig. 1b.

In the Eclipse SWT (Simple Windowing Toolkit), there are 13 classes (such as `Button`, `Label`, and `Link`) with the methods `getText` and `setText` that get and set the main text for the control. But, there is no common `IText` interface. Azureus, a BitTorrent client, is an application that requires the ability to call these methods in a

```
if (widget instanceof Label)
  ((Label) widget).setText(message);
else if (widget instanceof CLabel)
  ((CLabel) widget).setText(message);
else if (widget instanceof Group)
   ((Group) widget).setText(message);
... // 5 more items
```

(a)

```
if (widget instanceof CoolBar) {
  CoolItem[] items = ((CoolBar)widget).getItems();
  for(int i = 0; i < items.length; i++) {
    Control control = items[i].getControl();
    updateLanguageForControl(control);
  }
} else if (widget instanceof TabFolder) {
  ... // same code
} else if (widget instanceof CTabFolder) {
  ... // same code
... // 5 more items
```

(b)

Fig. 2. Code excerpts from Azureus, illustrating an awkward coding style and duplication.

generic fashion. Azureus is localized for a number of languages, which can be changed at runtime. Accordingly, there are several instances of code similar to that of Fig. 2.

Note that some of this code duplication might be avoided if the class hierarchy were refactored. Obviously, this is not always possible—e.g., Azureus cannot modify SWT.

In summary, common methods can lead to undesirable code duplication. Structural subtyping can help eliminate this problem, without refactoring the class hierarchy.

6 Cascading "instanceof" Tests

We considered the question of whether structural subtyping could provide benefits if used in conjunction with other language features—external methods in particular. *External methods* (also known as open classes) are similar to ordinary methods and provide the the usual dispatch semantics, but can be implemented outside of a class's definition, providing more flexibility. Multimethods are a generalized form of external method, defined outside all classes and allowing dispatch on any subset of a method's arguments [9, 4, 10, 17].

Since Java does not support any form of external dispatch, programmers often compensate by using cascading `instanceof` tests. This programming pattern is problematic because it is tedious, error-prone, and lacks extensibility [10]. Many instances of this pattern could be re-written to use external methods, but a problem arises if an `instanceof` test is performed on an expression of type `Object`.

To illustrate this, let us consider how `instanceof` tests would be translated to external methods. Suppose we have a cascaded `instanceof`, with each case of the form "`[else] if expr instanceof` C_i `{` $block_i$ `}`." This would be translated to an external method f defined on `expr`'s class, and overridden for each C_i by defining $C_i.f$ `{` $block_i$ `}`. The top part of Fig. 3b shows the external methods translated from the `instanceof` tests in Fig. 3a (but without an external method defined on `Object`, the type of `query`, which we will come to in a moment).

A problem arises when the target expression in the `instanceof` test is of type `Object`, as an external method must be defined on `Object`, then overridden for each type tested via an `instanceof`. The problem with this solution is that it pollutes the interface of `Object`. In many cases, the implementation of this method performs a generic fallback operation that does not make sense for an object of arbitrary type—but

```
List qlist = ...
Object query = qlist.get(i);
Query q = null;
if (query instanceof String)
  q = parser.parse((String) query);
else if (query instanceof Query)
  q = (Query) query;
else
  System.err.println(
    "Unsupported query type");
```

(a)

```
// external methods
Query String.toQuery(QueryParser parser) {
  return parser.parse(this);
}
Query Query.toQuery(QueryParser parser) {
  return this;
}
   ...
// structural type
struct QueryConvert { Query toQuery(QueryParser) };
List<QueryConvert> qlist = ...
Query q = qlist.get(i).toQuery(parser);
```

(b)

Fig. 3. Rewriting `instanceof` using structural subtyping and external dispatch. Listing (a) is the original code; listing (b) is the translated code, which defines the structural type `QueryConvert` and external methods on `Query` and `String`. Note that the translated code eliminates the need for the error condition.

Table 5. Total `instanceof` tests, the number present in cascading `if` statements that perform the test on an expression of type `Object`, and that number expressed as a percentage. Code written using this pattern can be translated to a language with structural subtyping and external dispatch.

	`instanceof`	Expression of type `Object`	Percentage
Apache collections	225	75	33%
Areca	77	10	13%
JHotDraw	229	50	22%
log4j	54	8	15%
Lucene	56	10	18%
PLT collections	119	64	54%
Smack	56	20	36%
Tomcat	959	158	16%
Average			**26%**

this method becomes part of every class's interface and implementation. (While it is also possible to pollute the interface of an arbitrary class C, this is generally less severe, and detecting such a situation requires application-specific knowledge.)

To determine the prevalence of this pattern, we manually searched for `instanceof` tests in 8 applications, and found that 13% to 54% (with an average of 26%) were performing a cascading `instanceof` test on an expression of type `Object` (see Table 5).

Structural subtyping provides one solution to this problem. We have previously defined a language with both structural subtyping and external dispatch [17]. The type of the expression on which the `instanceof` is performed would be changed from `Object` to the structural type consisting of the newly defined external method f. That is, instead of making the target operation applicable to an arbitrary object, it would be applicable to only those objects that contain method f. Figure 3b defines an external method `toQuery` on `String` and `Query`, then uses the structural type `{ toQuery(...) }` as the type for the `List` elements. The advantage of using structural subtyping is that the main code can call this method uniformly. [2]

[2] Note that it would not be possible to make use of a nominal interface containing the method f

Thus, for many applications, there is a potential benefit to using structural subtyping in a language that supports external dispatch; an average of 26% of `instanceof` tests could be eliminated.

Note that since we refined the element type of the `List` object, this obviates the need for the error condition—an additional advantage. However, it is not always possible to refine types to a structural type; an expression may simply have type `Object`, due to the loss of type information. In such a case, it would be possible to re-write the code using a structural downcast. Though the use of casts would not be eliminated, there are still several advantages to this implementation style. First, the external methods could be changed without having to also modify the method that uses them. Also, if subclasses are added, a new internal or external method could be defined for them. Finally, since the proposed cast would use a structural type, it would be more general, applying to any type for which the method were defined.

7 Java Reflection Analysis

We aimed to answer the following question: do Java programs use reflection where structural types would be more appropriate? We hypothesized that uses of reflection fall into two categories: cases where dynamic class instantiation and classloading are used, and cases where the type system is not sufficiently powerful to express the programming pattern used. It is difficult to eliminate reflection in the first category, as these uses represent an inherently dynamic operation. However, some of the uses in the second category could potentially be rewritten using structural downcasts. Reducing the uses of reflection is beneficial as it decreases the number of runtime errors and can improve performance.

We examined 28 applications, and found that an average of 32% of uses of the reflection method `Class.getMethod` could be re-written using a structural downcast (see Table 6). A structural downcast is preferable to reflection because type information is retained when later calling methods, as opposed to `Method.invoke`, which is passed an `Object` array and must typecheck the arguments at runtime. Additionally, it is easier to combine sets of methods in a downcast; when using reflection, each method must be selected individually. There is also the potential to make method calls more efficient, which is difficult with reflection, due to the low-level nature of the available operations. (For example, the language Whiteoak [14] supports efficient structural downcasts.)

In summary, the high percentage of reflection uses that can be translated to structural downcasts suggests that programmers may sometimes use reflection as a workaround for lack of structural types.

8 Related Work

A number of research languages support structural subtyping, such as O'Caml [15], PolyToil [6], Moby [11], and Strongtalk [5]. We have also previously defined a language

to call the method in a generic manner. For external methods to be modular, once a method is defined as an internal method, it cannot be implemented with an external method; see [20, 10].

Table 6. Uses of the reflection method `Class.getMethod`, and the number and percentage that could be re-written using a structural downcast. Programs that did not call this method are omitted. The percentage entry in the last row is calculated by dividing the total "*could be rewritten*" by the total "*uses of getMethod*."

	Uses of `getMethod`	Could be rewritten	Percentage
Ant	36	9	25%
Apache Collections	4	3	75%
Areca	1	0	0%
Azureus	27	6	22%
Cayenne	28	4	14%
Columba	10	8	80%
hsqldb	2	0	0%
jEdit	10	7	70%
JFreeChart	1	1	100%
JHotDraw	26	1	4%
JRuby	17	6	35%
log4j	4	1	25%
OpenFire	2	0	0%
Tomcat	37	10	27%
Xalan	28	11	39%
Totals	**233**	**67**	**29%**

supporting both external dispatch and structural subtyping [17]. An evaluation of the benefits of each of nominal and structural subtyping is available in [23, 17].

As mentioned in Sect. 3, researchers have studied the problem of refactoring programs to use most general nominal types where possible [12, 24]. Structural subtyping would make such refactorings more feasible (since new types would not have to be defined) and applicable to more type references in the program (since structural supertypes for library types could be created, while new interfaces cannot).

Muschevici et al. measured the number of cascading `instanceof` tests in a number of Java programs, to determine how often multiple dispatch might be applicable [21]. They found that cascading `instanceof` tests were quite common, and that many cases could be rewritten to use multimethods; this is consistent with our results.

Corpus analysis is commonly used in empirical software engineering research. For example, it has been used to examine non-nullness [8], aspects [2], micro patterns [13], and inheritance [25].

9 Summary and Conclusions

In summary, we found that a number of different aspects of Java programs suggest the potential utility of structural subtyping. While some of the results are not as strong as others, taken together the data suggests that programs could benefit from the addition of structural subtyping, even if they were written in a nominally-typed language.

We hope that the results of this study will be used to inform designers of future programming languages, as well as serve as a starting point for further empirical studies in this area. Ultimately, one must study the way structural subtyping is eventually used by mainstream programmers; this work serves as a step in that direction.

Acknowledgements We would like to thank Ewan Tempero for helpful discussions and feedback, and Nels Beckman and the reviewers for comments on an earlier version of this paper. This research was supported in part by the U.S. Department of Defense, Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems," and NSF CAREER award CCF-0546550.

References

[1] R. Amadio and L. Cardelli. Subtyping recursive types. *ACM TOPLAS*, 15(4), 1993.
[2] P. Baldi, C. Lopes, E. Linstead, and S. Bajracharya. A theory of aspects as latent topics. In *OOPSLA*, 2008.
[3] J. Bloch. *Effective Java, Second Edition*. Addison-Wesley, 2008.
[4] J. Boyland and G. Castagna. Parasitic methods: an implementation of multi-methods for Java. In *OOPSLA '97*, pages 66–76, 1997.
[5] G. Bracha and D. Griswold. Strongtalk: typechecking Smalltalk in a production environment. In *OOPSLA '93*, pages 215–230, 1993.
[6] K. Bruce, A. Schuett, R. van Gent, and A. Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Trans. Program. Lang. Syst.*, 25(2):225–290, 2003.
[7] L. Cardelli. Structural subtyping and the notion of power type. In *POPL '88*, 1988.
[8] P. Chalin and P. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *ECOOP*, 2007.
[9] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92*, 1992.
[10] C. Clifton, T. Millstein, G. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM TOPLAS.*, 28(3):517–575, 2006.
[11] K. Fisher and J. Reppy. The design of a class mechanism for Moby. In *PLDI*, 1999.
[12] Florian Forster. Cost and benefit of rigorous decoupling with context-specific interfaces. In *PPPJ '06*, pages 23–30, 2006.
[13] J. Gil and I. Maman. Micro patterns in Java code. In *OOPSLA '05*, pages 97–116, 2005.
[14] J. Gil and I. Maman. Whiteoak: Introducing structural typing into Java. In *OOPSLA*, 2008.
[15] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, release 3.10. Available at `http://caml.inria.fr/pub/docs/manual-ocaml`, 2007.
[16] B. Magnusson. Code reuse considered harmful. *Journal of Object-Oriented Programming*, 4(3), November 1991.
[17] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In *ECOOP '08*, July 2008.
[18] Donna Malayeri and Jonathan Aldrich. Is structural subtyping useful? An empirical study. Technical Report CMU-CS-09-100, School of Computer Science, Carnegie Mellon University, January 2009.
[19] Sun Microsystems. Java collections API design FAQ. Available at `http://java.sun.com/j2se/1.4.2/docs/guide/collections/designfaq.html`, 2003.
[20] T. Millstein and C. Chambers. Modular statically typed multimethods. *Inf. Comput.*, 175(1):76–118, 2002.
[21] R. Muschevici, A. Potanin, E. Tempero, and J. Noble. Multiple dispatch in practice. In *OOPSLA 08*, October 2008.
[22] D. Musser and A. Stepanov. Generic programming. In P. Gianni, editor, *ISAAC '88*, volume 38 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 1989.
[23] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.
[24] F. Steimann. The infer type refactoring and its use for interface-based programming. *Journal of Object Technology*, 6(2), 2007.
[25] E. D. Tempero, J. Noble, and H. Melton. How do Java programs use inheritance? An empirical study of inheritance in Java software. In *ECOOP '08*, pages 667–691, 2008.