Cost and Benefit of Rigorous Decoupling with Context-Specific Interfaces

Florian Forster
University of Hagen
Universitätsstraße
D-58097 Hagen
+49 (0) 2331 987-4290
florian.forster@fernuni-hagen.de

ABSTRACT

In Java programs, classes are coupled to each other through the use of typed references. In order to minimize coupling without changing the executed code, interfaces can be introduced for every declaration element such that each interface contains only those members that are actually needed from the objects referenced by that element. While these interfaces can be automatically computed using type inference, concerns have been raised that rigorous application of this principle would increase the number of types in a program to levels beyond manageability. It should be clear that decoupling is required only in selected places and no one would seriously introduce a minimal interface for every declaration element in a program. Nevertheless we have investigated the actual cost of so doing (counted as the number of new types required) by applying rigorous decoupling to a number of open source Java projects, and contrasted it with the benefit, measured in terms of reduced overall coupling. Our results suggest that (a) fewer new interfaces are needed than one might believe and (b) that a small number of new interfaces accounts for a large number of declaration elements. Particularly the latter means that automated derivation of decoupling interfaces may at times be useful, if the number of new interfaces is limited a priori to the popular ones.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: *Modules and interfaces*

General Terms

Design

Keywords

The Java Language::Java-specific metrics
The Java Language::Optimization
Software Engineering with Java::Tools for Java Programming

1. INTRODUCTION

Interface-based programming as described in [1] is accepted as a useful object-oriented programming technique. According to it, references in a program should be declared with interfaces and not classes as their types. Two main benefits can be expected from this: First, flexibility is increased as classes implementing an interface can be exchanged without notification of the client using the classes' services via the interface. Second, the access to the class is restricted to the methods declared in the interface typing a variable which holds a reference to an instance of the class. While these interfaces can be automatically computed using type inference, concerns have been raised that rigorous application of this principle would increase the number of types in a program to levels beyond manageability

Nevertheless we have investigated the actual cost of so doing (counted as the number of new types required) by applying rigorous decoupling to a number of open source Java projects, and contrasted it with the benefit, measured in terms of reduced overall coupling.

The rest of the paper is organized as follows. In section 2 we will briefly introduce the refactoring and the metrics used in this paper, which already have been described in [10]. In section 3.1 we introduce our test suite, which consists of six open source projects. Afterward, we outline the initial situation before the refactoring using various metrics in section 3.2. The same metrics are applied on the projects after the refactoring in section 3.3. In section 3.4 we discuss the costs of the refactoring in terms of new types required in contrast to the reduced overall coupling. In section 3.5 we present additional insights gained during the investigation of our test suite. Section 4 recapitulates our results; section 5 concludes and provides pointers to future work.

2. THE REFACTORING

2.1 Measuring Coupling

Typing rules in Java enforce the type of a declaration element to offer at least the set of methods invoked on that declaration element. Coupling between classes increases to levels beyond what is necessary, when a declaration element is declared with a type offering more methods than actually needed by this declaration element. In the following we use the goal question metric approach to derive a suitable metric measuring unnecessary coupling.

The methods publicly available for a type $\boldsymbol{T}$ are a subset of all the methods declared in $\boldsymbol{T}$ or one of its supertypes[1]. Let $\mu(\boldsymbol{T})$ be the set of methods declared in a type T. Then we declare

[1] In Java we do not count the methods inherited from *Object*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPPJ 2006, August 30 – September 1, 2006, Mannheim, Germany.
Copyright 2006 ACM 1-58113-000-0/00/0004…$5.00.

Table 1: Use of interfaces and decoupling quality in six open source projects before the refactoring

Project:	JChessBoard	JUnit	Mars	GoGrinder	DrawSWF	JHotDraw
Types	49 (100%)	73 (100%)	103 (100%)	118 (100%)	346 (100%)	395 (100%)
Classes	32 (65%)	48 (66%)	77 (75%)	95 (80%)	282 (82%)	301 (76%)
Interfaces	17 (35%)	25 (34%)	26 (25%)	23 (20%)	64 (18%)	94 (24%)
Ratio	1:0,53	1:0,52	1:0,34	1:0,24	1:0,23	1:0,31
Declaration Elements	190	166	278	468	1361	1801
Class typed	190	71	244	450	1109	395
Interface typed	0	95	34	18	252	1406
Ratio	-	1:1,338	1:0,139	1:0,040	1:0,227	1:3,560
ACD						
Average	0,365	0,216	0,447	0,567	0,416	0,142
Highest	0,813	0,997	1	0,998	1	1
Lowest	0	0	0	0	0	0
BCD						
Average	0,341	0,211	0,336	0,484	0,351	0,100
Highest	0,813	0,992	0,961	0,998	0,996	0,998
Lowest	0	0	0	0	0	0
ACD-BCD						
Average	0,024	0,005	0,110	0,084	0,065	0,041
Highest	0,222	0,326	1	0,795	1	1
Lowest	0	0	0	0	0	0

$$\pi(T) := \{ m \in \mu(T) \mid m \text{ is a public nonstatic method} \}$$

as the set of methods offered by a type ***T*** to all declaration element typed with ***T***.

In strongly typed languages like Java, each declaration element has a declared type. Besides its declared type, each declaration element also has an inferred type. Here, we define the inferred type of a declaration element ***d*** as the type ***I*** whose set of publicly available methods $\pi(I)$ is the smallest set containing all methods directly accessed on ***d***, united with the sets of methods accessed on all declaration elements ***d*** possibly gets assigned to. We define $\pi(I)$ as a function of ***d***, $\iota(d)$, which can be computed by analyzing the source code using *Static Class Hierarchy Analysis* as described in [9]. We call this inferred type ***I*** a *context-specific interface* or *minimal interface* for ***d*** and $\iota(d)$ the *access set* of ***d***.

In order to reduce unnecessary coupling between classes we have to redeclare every declaration element ***d*** with the type ***I*** so that $\pi(I)=\iota(d)$, i.e. the new declared type of ***d*** offers only those methods which are actually invoked on ***d*** or one of the declaration elements ***d*** possibly gets assigned to. According to the goal question metric approach described in [14] we can define a metric for coupling. Our question is how much unnecessary coupling is introduced to a type ***T*** by means of a declaration element ***d***, and our goal is to reduce this unnecessary coupling. We therefore define the coupling induced by a declaration element ***d*** as the quotient of the number of methods needed from ***d***, and the number of methods provided by its declared type ***T***:

$$\frac{|\iota(d)|}{|\pi(T)|}$$

A value of 1 indicates that ***d*** is declared with the least specific (most general or *minimal* in the sense that it has as few members as possible) type ***T***, i.e. the type offering only those methods actually needed, whereas one of 0 implies that none of **T**'s methods are used.

Obviously 1 – the quotient is a measure of the possible reduction of the unnecessary coupling established by the declaration element ***d,*** if ***d*** is redeclared with a type ***I*** so that $\pi(I)=\iota(d)$. We have called this metric the *Actual Context Distance* (ACD) [10] for a declaration element ***d*** with type ***T*** and write

$$ACD(d,T) = \frac{|\pi(T)| - |\iota(d)|}{|\pi(T)|}$$

The *Best Context Distance* (BCD) for a declaration element ***d*** is then the lowest value of ACD achievable by redeclaring ***d*** with a type ***T*** which is already available in the project. It is a measure for the maximum decoupling that can be achieved by using only existing types.

We define ACD and BCD for a type ***T*** as the average ACD and BCD of all declaration elements typed with ***T***. The ACD and BCD for a project is the weighted average, the weight of a type ***T*** being the number of declaration elements typed with ***T***, of the ACD and BCD for each type of the project.

2.2 Refactoring for interface-based programming

In order to improve decoupling to the maximum, we can use the refactoring described in [10]. It reduces *ACD(**d**,**T**)* to zero for every declaration element ***d*** in a project. This is done by either

Table 2: Use of interfaces and decoupling quality in six open source projects after the refactoring

Project:	JChessBoard	JUnit	Mars	GoGrinder	DrawSWF	JHotDraw
Types	92 (100%)	110 (100%)	190 (100%)	260 (100%)	599 (100%)	709 (100%)
Classes	32 (35%)	48 (43%)	77 (41%)	95 (37%)	282 (47%)	301 (42%)
Interfaces	60 (65%)	63 (57%)	113 (59%)	165 (63%)	317 (53%)	418 (58%)
Ratio	1:1,88	1:1,31	1:1,47	1:1,74	1:1,13	1:1,39
Declaration Elements	177	124	267	447	1241	1655
Class typed	103	34	81	121	404	81
Interface typed	74	90	186	326	837	1574
Ratio	1:0,71	1:2,64	1:2,30	1:2,69	1:2,07	1:19,43
Declaration Elements per type	1,92	1,12	1,41	1,72	2,07	2,33

a) using an existing, better suited interface ***I*** for ***d*** with $\pi(\boldsymbol{I})=\iota(\boldsymbol{d})$, or by

b) introducing a new interface ***I*** for ***d*** tailored to fulfill $\pi(\boldsymbol{I})=\iota(\boldsymbol{d})$

for the redeclaration of ***d*** so that $ACD(\boldsymbol{d},\boldsymbol{I})=0$.

For the purpose of redeclaring ***d*** with its least specific type ***I***, no matter if this type already exists or is newly introduced, the type inference algorithm we use has to consider three cases.

First if the declared type ***T*** of a declaration element ***d*** is already minimal, i.e. $\pi(\boldsymbol{T})=\iota(\boldsymbol{d})$, no changes to the program are necessary.

If however $|\pi(\boldsymbol{T})|>|\iota(\boldsymbol{d})|$, i.e. ***d***'s declared type is not minimal, and ***d*** terminates an assignment chain, i.e. ***d*** is not assigned to any other declaration element, all we have to do is declare ***d***'s minimal type ***I***, i.e. $\pi(\boldsymbol{I})=\iota(\boldsymbol{d})$, as a supertype of ***T*** and redeclare ***d*** with ***I***.

Finally ***d***'s declared type might neither be minimal nor terminate an assignment chain. To illustrate this case we use a primitive scenario with two declaration elements ***A a*** and ***B b*** and the assignment ***b=a***.

Assuming that the program at hand is type correct and using the rules for typing in Java we know that the type ***B*** is the same type as **A** or a supertype of ***A***. Computing $\iota(\boldsymbol{a})$ gives us the new type ***I*** for the declaration element ***a***. Unfortunately redeclaring ***a*** with ***I*** results in a program which is not type correct, as the assignment of ***a*** to ***b***, i.e. ***b=a*** for ***I a*** and ***B b***, is undefined in Java, if ***I*** is not a subtype of ***B*** .

To solve this issue we can simply declare ***I*** as a subtype of ***B***. This makes the program type correct as ***A*** is a subtype of or equal to ***B*** and ***I*** is a subtype of or equal to ***A***. However introducing this relationship might renders ***I*** not to be a minimal type for ***a*** as it might add unwanted methods to ***I*** coming from ***B***, i.e. $\pi(\boldsymbol{B}) - \pi(\boldsymbol{I})$ is not empty.

As the introduction of this relationship might result in unwanted methods added to ***I***, we redeclare the declaration element ***b*** with some type ***J***, so that ***J*** is a supertype of ***I***, to make the program type correct again.

In case ***J*** is a real supertype of ***I***, i.e. $\pi(\boldsymbol{J}) \subset \pi(\boldsymbol{I})$, we have to make sure that ***J*** is declared as a supertype of ***I***. Furthermore in order to keep other assignments to ***b*** correct we have to make sure that ***J*** is declared as a supertype of ***B***.

2.3 Limitations of the Refactoring for Java

The implementation of the refactoring described in the last section has a few limitations due to Java's type system. If fields of a class are directly accessed using a declaration element, i.e. without using accessor methods, this declaration element can not be redeclared with an interface. Even though one could define an abstract class for the purpose of redeclaration we do not do so, as multiple inheritance is not possible in Java. Therefore this workaround would work only in a limit number of cases. Furthermore redeclaration of a declaration element with an interface is also not possible if nonpublic methods are accessed on the declaration element. Thus we excluded these declaration elements from our investigation as they can not be redeclared, i.e. unnecessary coupling does not exist.

The second limitation are empty interfaces. Declaration elements with empty access sets might be redeclared with empty interfaces as they are an ideal type. However empty interfaces, so called *tagging interfaces* or *marker interfaces*, are used in *instanceof b*oolean expressions in Java. Typing a declaration element with an empty interface might therefore lead to circumstances, in which the boolean expression evaluates to true after the redeclaration. To avoid these cases we rather redeclare declaration elements with empty access sets with the root of the type hierarchy, i.e. *Object* in Java. As only the number of declaration elements typed with types defined within the project is considered for the metrics, the declaration elements retyped with *Object* disappear. This is justified by the fact that every declaration element which is typed with *Object* has no influence on the coupling as a "coupling" with *Object* always exists due to the nature of Java, i.e. every type is subtype of *Object*.

Furthermore interfaces already existing in a project might become superfluous after the refactoring, i.e. no declaration elements are typed with these interfaces. However, even though we could, we do not delete these interfaces!

In the remainder of this paper we will use InferType on a number of open source projects, so that we can evaluate both the costs, in terms of newly introduced types, and the benefits, in terms of improved decoupling, of rigorous decoupling.

Table 3: Comparison of the situation before and after the refactoring

Project:	JChessBoard	JUnit	Mars	GoGrinder	DrawSWF	JHotDraw
ΔTypes	+43 / +88%	+37 / +50%	+87 / +85%	+142 / +105%	+253 / +73%	+314 / +80%
ΔInterfaces	+43 / --	+37 / +37%	+87 / +335%	+142 / +617%	+253 / +395%	+314 / +343%
ΔDeclaration Elements	-13 / -7%	-42 / -25%	-11 / -4%	-21 / -5%	-120 / -8%	-146 / -8%
ΔClass typed	-87 / -46 %	-37 / -52%	-163 / -66%	-329 / -73%	-705 / -64%	-314 / -80%
ΔInterface typed	+74 / +∞ %	-5 / -5%	+ 152 / +447%	+308 / +1711%	+585 / +232%	+168 / +12%
ΔACD average	-0,365	-0,216	-0,447	-0,567	-0,416	-0,142
ACD average per new type	-0,0084	-0,0057	-0,0051	-0,0040	-0,0016	-0,0004

2.4 Implementation of the Refactoring

The refactoring described in [10] was implemented and called *InferType*[2]. The algorithm used for applying the refactoring to a complete project is outlined below:

```
changes=false
do
  foreach type in project
    DEs:=getDeclarationElements(type)
    foreach DE in DEs
      refactor(DE)
      if(hasChanged(project))
        changes=true
      endif
    endfor
  endfor
while(changes)
```

For each type in the project we iterate over all the declaration elements declared with this type. We then apply the refactoring described in section 2.2 to each of these declaration elements. If there was a change, i.e. a new type was introduced to the project during the refactoring of a declaration element; we repeat the process until no more changes happen. After using this algorithm on a project every declaration element in this project is typed with a minimal type, i.e. *ACD(**d**,**T**)* is always zero.

3. ANALYSING RIGOROUS DECOUPLING WITH CONTEXT-SPECIFIC INTERFACES

3.1 Introducing the Test Suite

To evaluate the costs and benefits of rigorous decoupling using minimal interfaces we investigated six picked open source projects. We created a balanced test suite using popular Java projects[3] which span a number of domains.

JChessBoard [2] is a chess game capable using a regular TCP/IP connection to play against human opponents. Furthermore it is capable of editing and viewing the PGN[4] format.

JUnit [3] is a popular framework for unit testing in the Java programming language.

Mars [4] is a simple, extensible, services-oriented network status monitor written in Java.

GoGrinder [5] is a Java program for practicing Go problems using the SGF[5] format to load these problems.

DrawSWF [6] is a simple drawing application written in Java. The drawings created can be exported as an animated SWF (Macromedia Flash) file.

JHotDraw [7] is a well-known framework for developing two-dimensional structured drawing editors.

These projects have been completely refactored using InferType. Table 1 presents metrics regarding the size of the projects and decoupling before the refactoring occurs. We will discuss these results in detail in the next subsections.

3.2 Before the Refactoring

3.2.1 General Observations

In every project we found that there exist more classes than interfaces. Values range from about two classes per interface to about five classes per interface. We expect that after the refactoring the numbers are in favor of the interfaces, i.e. there are more interfaces than classes in each project.

Table 1 also reveals that there are developers, or project teams, which use interfaces for typing declaration elements, and those who don't. In particular in *JUnit* and *JHotDraw* more declaration elements are typed with interfaces than with classes. Contrary to these two projects a much smaller number of declaration elements are typed with interfaces in *Mars*, *GoGrinder* and *DrawSWF*. Even worse in *JChessBoard* there is not a single declaration element typed with an interface.

[2] Available at http://www.fernuni-hagen.de/ps/docs/InferType/.

[3] We used the popularity rating provided at http://www.freshmeat.net.

[4] PGN stands for "Portable Game Notation", a standard designed for the representation of chess game data using ASCII text files.

[5] SGF is the abbreviation of 'Smart Game Format'. The file format is designed to store game records of board games for two players.

As could be expected given the large number of available interfaces ACD values for both *JUnit* and *JHotDraw* are low. For example in *JHotDraw* a declaration element on average does not use 10% of the available methods, whereas a declaration element in *GoGrinder* on average does not use 57% of the available methods.

However, BCD values indicate that decoupling in all projects could be improved using only existing types. Nevertheless these improvements are small and therefore we conclude that developers already make good use of existing types for typing declaration elements.

3.2.2 The Projects in Detail

JChessBoard was the smallest project in our test suite. Even though half of the used types in these projects are interfaces, not a single declaration element is typed with an interface. This is due to the fact that JChessBoard extends classes from the JDK. These classes therefore contain methods from the JDK classes for which the formal parameters are typed with interfaces. Due to Java's typing rules the classes in JChessBoard have to implement these interfaces to make use of the inherited methods. Additionally 149 out of 190 declaration elements are typed with one out of five types from the total number of 49 available types. We expect that the benefit of refactoring in relation to the number of newly introduced types is biggest for this project.

JUnit is one of two projects in our test suite in which more declaration elements are typed with an interface than with a class. In *JUnit* the difference of the average ACD and the average BCD is significantly low, i.e. *JUnit*'s declaration elements are mostly typed with the best fitting type existing in the project. Furthermore the interface *junit.framework.Test* is used to type 68 out of 166 declaration elements. We expect that most of these declaration elements will be retyped with new interfaces, i.e. we expect that *junit.framework.Test* offers more methods than needed for most declaration elements.

Mars is the counterpart to *JUnit* regarding the usage of existing types. Redeclaration of every declaration element with existing types would already reduce the ACD value by 0,11. Notably, similar to *JChessBoard,* five out of 103 types account for 143 out of 278 declaration elements.

GoGrinder is similar to *JChessBoard* in terms of typing declaration elements with interfaces. Only 4% of all declaration elements in this project are typed with interfaces. Furthermore it has the highest average ACD value of all projects. We expect that some types, most likely the ones with a high ACD value, will trigger the creation of many new interfaces.

DrawSWF has the lowest class-to-interface ratio of all projects. There are approximately five times as many classes as interfaces used in this project. Furthermore half of the declaration elements were typed with 7% of the existing types.

JHotDraw is outstanding in two ways. First it is the project which makes most use of interfaces for typing declaration elements. Second both the average ACD value and the average BCD value are the lowest in our test suite, i.e. there is little coupling existing in this project and most times the best fitting and existing type is used to type a declaration element.

3.3 After the Refactoring

3.3.1 General Observations

Table 2 shows the same metrics as Table 1, but this time after using InferType on the projects. Note that we omitted all ACD and BCD values as the very purpose of the refactoring is making these values zero, in which it succeeded.

However, it is surprising that less new interfaces were introduced to the projects than one might fear. The worst case, i.e. one new interface for every existing declaration element, never occurred. Actually all projects were not even close to the worst case as the last row in Table 2 shows. This is an indication that at least some declaration elements are using the same access set and could therefore be declared with the same type. Nevertheless there are many newly introduced interfaces which are unpopular, i.e. there are only few declaration elements typed with these interfaces. Figures 1 to 8 in the appendix show the popularity, in terms of declaration elements typed with a particular interface, of each interface for every project.

Yet, not every declaration element is declared with an interface as its type. Except the two extremes *JChessBoard* and *JHotDraw* around two or three times as many declaration elements are typed with interfaces as with classes.

A comparison of the situation before and after the refactoring is show in Table 3. In the next section we will present more detailed information about the changes which occurred during the refactoring.

3.3.2 The Projects in Detail

JChessBoard profited the most from the refactoring which is not astonishing, because it was using no interface at all for typing declaration elements. About half of the declaration elements are retyped with interfaces during the refactoring. From 149 declaration elements declared with one out of five types only 73 declaration elements where still typed with these types after the refactoring. In particular all declaration elements formally typed with the inner class *STR* from *jchessboard.PGN* are now typed with an interface.

JUnit offered a little surprise as after the refactoring less declaration elements were typed with interfaces than before the refactoring. This is due to the fact that declaration elements formerly typed with an interface are now typed with *Object*, as the access set of these declaration elements was empty. To be precise, 42 out of 68 declaration elements of the interface *junit.framework.Test* are now typed with *Object.*

Mars had a similar starting position as *JChessBoard*. In both projects a small number of types have been used to type an overwhelming part of the existing declaration elements. Hence both projects behaved similar during the refactoring. Like in *JChessBoard*, from 143 declaration elements typed with one out of five types from all available types only 46 of them were still typed with these types after the refactoring. In particular from the 34 declaration elements typed with the clas *org.altara.mars.Status* only one was still typed with this class after the refactoring.

GoGrinder was the second worst project -*JChessBoard* being the worst- in terms of using interfaces for typing declaration elements. Furthermore it had the highest average ACD value of all projects, i.e. a declaration element in *GoGrinder* on average did not use 57% of the available methods, and we expected that some types

will trigger the creation of many new interfaces. For example *GoGrinder.ProbCollection*, the most popular type before the refactoring, triggered the creation of 19 new interfaces for redeclaring the declaration elements formally typed with *GoGrinder.ProbCollection*.

DrawSWF had the lowest class-to-interface ratio before the refactoring and after the refactoring not much changed. It is interesting to note that one of the newly introduced interfaces is more popular than any interface or class before the refactoring[6]. This leads to the conclusion that this new interface is used to redeclare declaration elements from various types, a strong indication that an unwanted structural match occurred. This leads to circumstances in which two declaration elements are considered in terms of types, and therefore the methods which can be accessed, even though one of the objects is not. Thus even the program is type correct; semantics of the program might have changed. Furthermore 158 of the 253 newly introduced interfaces were so specific that each of them was used to retype only one declaration.

JHotDraw was the project which made the heaviest use of interfaces for typing declaration elements. Furthermore the ACD values were small throughout, i.e. the amount of unused methods was relatively small. Thus newly introduced interfaces are very specific. As a matter of fact, 164 out of 324 newly introduced interfaces were used to redeclare just one declaration element.

3.4 Costs of Rigorous Decoupling

About twice as many types exist after the refactoring than before refactoring in every project. Even though the introduction of additional types is necessary for removing unnecessary decoupling every additional type makes the type hierarchy harder to understand and maintain. For example the class *GoGrinder.ProbCollection* in the project *GoGrinder* implements as many as seventeen interfaces after the refactoring. Therefore to evaluate the refactoring we use the number of newly introduced types as the cost for the refactoring.

Table 3 shows the average reduction of the ACD value in relation to the number of new types introduced, i.e. the higher the value the better. We will use this number as our cost/benefit ratio as the number of new types is our cost of the refactoring, and the average reduction of the ACD value is the benefit of the refactoring.

The low number for *JHotDraw* is eye-catching but not surprising, as this project already used more interfaces for typing declaration elements than classes before the refactoring. As mentioned in the last section about half of the new interfaces were so specific that they were used to type just one declaration element each.

The cost/benefit ratio for *DrawSWF* is similar to the one of *JHotDraw*. This is due to the fact that during the refactoring many very specific interfaces have been introduced to this.

In the remaining four projects the newly introduced interfaces were not as specific as in the above mentioned projects. This circumstance is reflected in the higher values of the cost/benefit ratio. Still a big part of the newly introduced interfaces was so specific that only few declaration elements could be redeclared with these interfaces. Figures 1 to 8 in the appendix show the popularity of the newly introduced types for a project. It is eye-catching that all projects have a few popular and many unpopular types.

3.5 Popular Types

Therefore the most interesting insight we gained after refactoring for each project is that *popular access sets*, which lead to *popular interfaces* during the refactoring, exist in every project. Figures 1 to 8 in the appendix show the popularity of each inferred interface. The popularity of an interface is defined as the number of declaration elements (the y-axis in the figures) declared with this interface.

All the diagrams in Figures 1 to 8 suggest a pareto distribution [13]. As a matter of fact the distribution of declaration elements among the types approximately follows the 80/20 rule, i.e. 80% of all declaration elements are typed with 20% of the available types, whereas the remaining 20% of all declaration elements are typed with 80% of the available types. Unfortunately most areas in which such a distribution occurs suffer from the so called *long tail*. In our case the long tail are all those types which are used only by a few declaration elements.

The results *JHotDraw* provides strong evidence that the cost/benefit ration also suffers from this distribution and that popular interfaces should be preferred. The average ACD value of this project was already low before the refactoring, i.e. the declared types provided a good decoupling. The refactoring introduced many unpopular types which were used to retype just one declaration element. This consequently led to the worst cost/benefit ratio. We therefore conclude that using only the most popular types, i.e. the 20% which are used by 80% of the retyped declaration elements, instead of using minimal interfaces everywhere results in a better cost/benefit ratio in terms of average ACD decrease per type.

In [11] we presented a metric, a tool and a guideline for finding popular access sets for a specific type. Using the refactoring however explicitly declares interfaces for all popular access sets in a project. These popular interfaces can be introduced to the original version of the project to reduce the ACD value, yet keeping the number of new types limited.

4. LESSONS LEARNED

In [12] the author noted that interfaces represent the roles a class plays and vice versa. However using an automatic refactoring to introduce minimal interfaces for every declaration element violates this principle. For example all declaration elements typed with *junit.framework.Test* in *JUnit* obviously play a specific role which is designated by the name of the interface. After redeclaring these declaration elements with *Object* no indication to a role is left. In section 3.5 we have shown that popular types exist in every project after the refactoring. [11] has shown that in many cases a role can be found for these popular types.

Rigorous decoupling comes with a high cost as shown in section 3.4. In section 3.5 we argued that introducing only popular interfaces might significantly reduce coupling, yet keeping the number of new types small.

Finally the results from section 3.2 indicate that finding the best fitting and existing type in a project for typing a declaration

---

[6] Unfortunately we have to omit the data which provided this insight due to its length.

element is not a problem. The difference of ACD and BCD value was low amongst all projects. This might be due to the fact that refactorings in prominent IDEs like Eclipse and IntelliJ exist which help the developer to find the best fitting type among all existing.

5. FUTURE WORK AND CONCLUSION

We have used an existing refactoring to evaluate both cost and benefit of the most rigorous decoupling as made possible by introducing context-specific types. Our results provide evidence that -as would be expected- rigorous decoupling is not a good idea. Too many unpopular interfaces are introduced during the refactoring. The data we have shown indicate that the best trade-off between decoupling and number of types is to introduce only the most popular interfaces for classes. We will have to adjust our refactoring and present data which either confirms or disproves our assumption.

6. ACKNOWLEDGMENTS

The author thanks Andreas Meißner and Phillip Mayer for implementing most of the described refactoring.

7. REFERENCES

[1] Löwy, J. *Programming .NET Components,* O'Reilly Media, 2005.

[2] JChessBoard is available from http://jchessboard.sourceforge.net

[3] JUnit is available from http://www.junit.org.

[4] Mars is available from http://leapfrog-mars.sourceforge.net/.

[5] GoGrinder is available from http://gogrinder.sourceforge.net/.

[6] DrawSWF is available from http://drawswf.sourceforge.net/.

[7] JHotDraw is available from http://www.jhotdraw.org/.

[8] Gamma, E. et al., *Design Patterns*, Addison-Wesley Professional, 1997.

[9] Dean, J. , Grove, D. and Chambers, C.*, Optimization of object-oriented programs using static class hierarchy analysis*, In: *Proc of ECOOOP*, 1995, 77-101.

[10] Steimann, F., Mayer, P. and Meißner, A., *Decoupling classes with inferred interfaces*, In: *Proceedings of the 2006 ACM Symposium on Applied Computing*, (SAC) (ACM 2006).

[11] Forster, F., *Mining Interfaces In Java Programs,* Technical Report, Fernuniversität Hagen, 2006.

[12] Steimann, F., *Role = Interface: a merger of concepts*, Journal of Object-Oriented Programming 14:4, 2001, 23–32.

[13] Pareto Distribution, *http://en.wikipedia.org/wiki/Pareto_distribution*, last visit 27.05.2006

[14] Basili, V.R., Caldiera, G. and Rombach, D., *The goal question metric approach*, In: *Encyclopedia of Software Engineering*, (John Wiley & Sons, 1994).

Appendix A

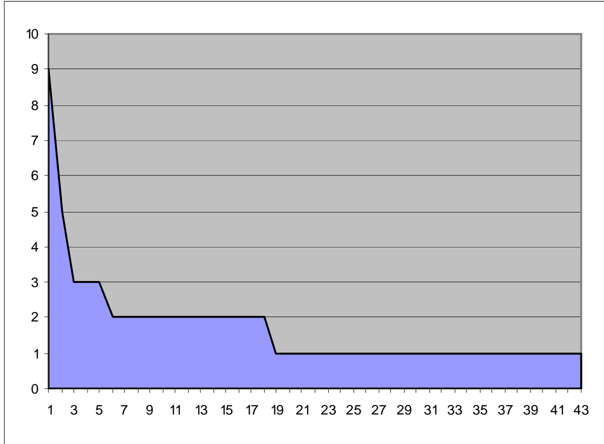

Figure 1: Popularity of new interfaces in JChessBoard

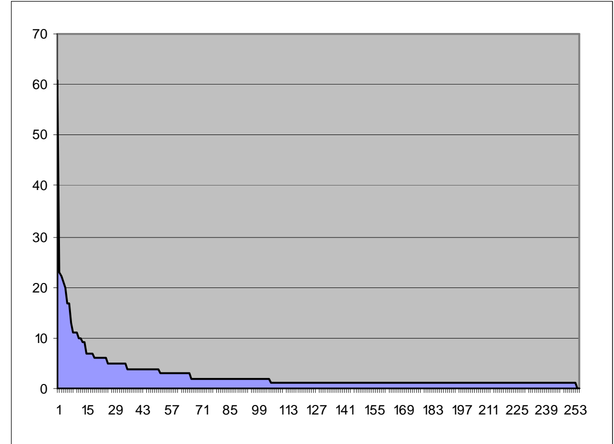

Figure 2: Popularity of new interfaces in DrawSWF

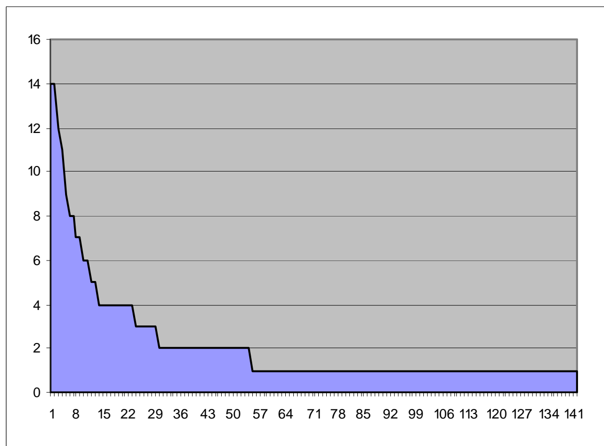

Figure 3: Popularity of new interfaces in GoGrinder

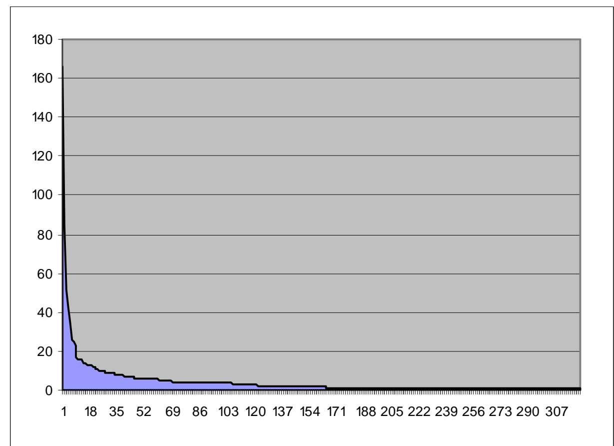

Figure 4: Interface Popularity of JHotDraw

Figure 5: Interface Popularity of JUnit

Figure 6: Interface Popularity of Mars