City University of New York (CUNY)
CUNY Academic Works

Spring 5-20-2017

Automated Refactoring of Legacy Java Software to Default Methods

Raffi T. Khatchadourian
CUNY Hunter College

Hidehiko Masuhara
Tokyo Institute of Technology

How does access to this work benefit you? Let us know!

Follow this and additional works at: http://academicworks.cuny.edu/hc_pubs

Part of the Programming Languages and Compilers Commons, and the Software Engineering Commons

Recommended Citation

Raffi Khatchadourian and Hidehiko Masuhara. Automated refactoring of legacy Java software to default methods. In International Conference on Software Engineering, ICSE '17. ACM/IEEE, May 2017.

This Poster is brought to you for free and open access by the Hunter College at CUNY Academic Works. It has been accepted for inclusion in Publications and Research by an authorized administrator of CUNY Academic Works. For more information, please contact AcademicWorks@cuny.edu.

Automated Refactoring of Legacy Java Software to Default Methods

Raffi Khatchadourian[1] Hidehiko Masuhara[2]
[1]Hunter College & the Graduate Center, City University of New York
[2]Tokyo Institute of Technology

Introduction

Java 8 introduces *enhanced* interfaces, allowing for *default* (instance) methods that implementers will inherit if none are provided [3]. Default methods can be used [2] as a replacement of the *skeletal implementation* pattern [1], which creates abstract skeletal implementation classes that implementers extend.

Migrating legacy code using the skeletal implementation pattern to instead use default methods can require significant manual effort due to subtle language and semantic restrictions. It requires preserving type-correctness by analyzing complex type hierarchies, resolving issues arising from multiple inheritance, reconciling differences between class and interface methods, and ensuring tie-breakers with overriding class methods do not alter semantics.

We propose an efficient, fully-automated, semantics-preserving refactoring approach, based on type constraints [4,5] and implemented as an open source Eclipse plug-in, that assists developers in taking advantage of enhanced interfaces. It identifies instances of the pattern and safely migrates class method implementations to interfaces as default methods.

Motivating Example

Consider the following interface:

```
interface Collection<E> {
    int size();
    void setSize(int i) throws Exception;
    void add(E elem); // optional.
    void removeLast();
    boolean isEmpty();
    int capacity();}
```

And the following skeletal implementation classes:

```
abstract class AbstractCollection<E> implements Collection<E>,
    Object[] elems; int size; // instance fields.
    @Override public int size() {return this.size;}
    @Override public void setSize(int i) {this.size = i;}
    @Override public void add(E elem)
        {throw new UnsupportedOperationException();}}
    @Override public void removeLast()
        {if (!isEmpty()) this.setSize(this.size()-1);}
    @Override public boolean isEmpty() {return this.size()==0;}
    @Override public int capacity() {return this.elems.length;}}
```

Implementers now extend `AbstractCollection` to inherit the skeletal implementations of `Collection`:

```
// inherits skeletal implementations:
List<String> list = new AbstractList<>() {};
```

And override its methods as needed:

```
List<string> immutableList = new AbstractList<> {
    @Override public void add(E elem) {
        throw new UnsupportedOperationException();}}
```

This pattern has several drawbacks, including:

Inheritance. Implementers extending `AbstractCollection` cannot further extend.

Modularity. No syntactic path between `Collection` and `AbstractCollection`.

Bloated libraries. Skeletal implementation classes must be separate from interfaces.

Refactoring Approach

Can we refactor abstract skeletal implementation classes to instead utilize default methods?

Type Constraints

$[E]$	the type of expression or declaration element E.
$[M]$	the declared return type of method M.
$Decl(M)$	the type that contains method M.
$Param(M, i)$	the i-th formal parameter of method M.
$T' \leq T$	T' is equal to T, or T' is a subtype of T.

Figure 1: Type constraint notation (inspired by [5]).

We use type constraints [4,5] to express refactoring preconditions. For each program element, type constraints denote the subtyping relationships that must hold between corresponding expressions for that portion of the system to be considered well-typed. Thus, a complete program is type-correct if all constraints implied by all program elements hold. Fig. 2 depicts several constraints used.

program construct	**implied type constraint(s)**
method call $E.m(E_1, \ldots, E_n)$ to a virtual method M (throwing exceptions $Ex_{t1}, \ldots, Ex_{tj}$)	$[E.m(E_1, \ldots, E_n)] \triangleq [M]$ (1) $[E_i] \leq [Param(M, i)]$ (2) $[E] \leq Decl(M_1) \vee \cdots \vee$ $[E] \leq Decl(M_k)$ (3) where $RootDefs(M) =$ $\{M_1, \ldots, M_k\}$ $\forall Ex_t \in \{Ex_{t1}, \ldots, Ex_{tj}\}$ (4) $\exists Ex_h \in Handle(E.m(E_1, \ldots, E_n))$ $[[Ex_t] \leq [Ex_h]]$
access $E.f$ to field F	$[E.f] \triangleq [F]$ (5) $[E] \leq Decl(F)$ (6)

Figure 2: Example type constraints.

Inferring Type Constraints

Type constraints are used to determine if a possible migration will either result in a type-incorrect or semantically different program. For example:

- Migrating `size()` and `capacity()` from `AbstractCollection` to `Collection` implies that $[\texttt{this}] = \texttt{Collection}$, violating constraint (6) that $[\texttt{this}] \leq [\texttt{AbstractCollection}]$.
- Migrating `AbstractCollection.setSize()` to `Collection` implies that the method now throws an `Exception`, violating constraint (4) as $Handle(\texttt{this.setSize(this.size()-1)}) = \emptyset$.
- `add()`, `removeLast()`, and `isEmpty()` from `AbstractCollection` can be safely migrated to `Collection` as a **`default`** method (Lst. 1) since there are no type constraint violations.

```
interface Collection<E> {
    int size();
    void setSize(int i) throws Exception;
    default void add(E elem) // optional.
        {throw new UnsupportedOperationException();}
    default void removeLast()
        {if (!isEmpty()) this.setSize(this.size()-1);}
    default boolean isEmpty() {return this.size()==0;}
    int capacity();
abstract class AbstractCollection<E> implements Collection<E> {
    Object[] elems; int size; // instance fields.
    @Override public int size() {return this.size;}
    @Override public void setSize(int i) {this.size = i;}
    ~~@Override public void add(E elem)~~
        ~~{throw new UnsupportedOperationException();}}~~
    ~~@Override public void removeLast()~~
        ~~{if (!isEmpty()) this.setSize(this.size()-1);}~~
    ~~@Override public boolean isEmpty(){return this.size()==0;}~~
    @Override public int capacity() {return this.elems.length;}
```

1: Refactored version.

Evaluation

subject	KL	KM	cnds	dflts	fps	δ	-δ	tm (s)
ArtOfIllusion	118	6.94	16	1	34	1	0	3.65
Azureus	599	3.98	747	116	1366	31	2	61.83
Colt	36	3.77	69	4	140	3	0	6.76
elasticsearch	585	47.87	339	69	644	21	4	83.30
Java8	291	30.99	299	93	775	25	10	64.66
JavaPush	6	0.77	1	0	4	0	0	1.02
JGraph	13	1.47	16	2	21	1	0	3.12
JHotDraw	32	3.60	181	46	282	8	0	7.75
JUnit	26	3.58	9	0	25	0	0	0.79
MWDumper	5	0.40	11	0	24	0	0	0.29
osgi	18	1.81	13	3	11	2	0	0.76
rdp4j	2	0.26	10	8	2	1	0	1.10
spring	506	53.51	776	150	1459	50	13	91.68
Tomcat	176	16.15	233	31	399	13	0	13.81
verbose	4	0.55	1	0	1	0	0	0.55
VietPad	11	0.58	15	0	26	0	0	0.36
Violet	27	2.06	104	40	102	5	1	3.54
Wezzle2D	35	2.18	87	13	181	5	0	4.26
ZKoss	185	15.95	394	76	684	0	0	33.95
Totals:	2677	232.2	3321	652	6180	166	30	383.17

Table 1: Experimental results.

Able to automatically migrate 19.63% (column **dflts**) of candidate methods despite of its conservatism.

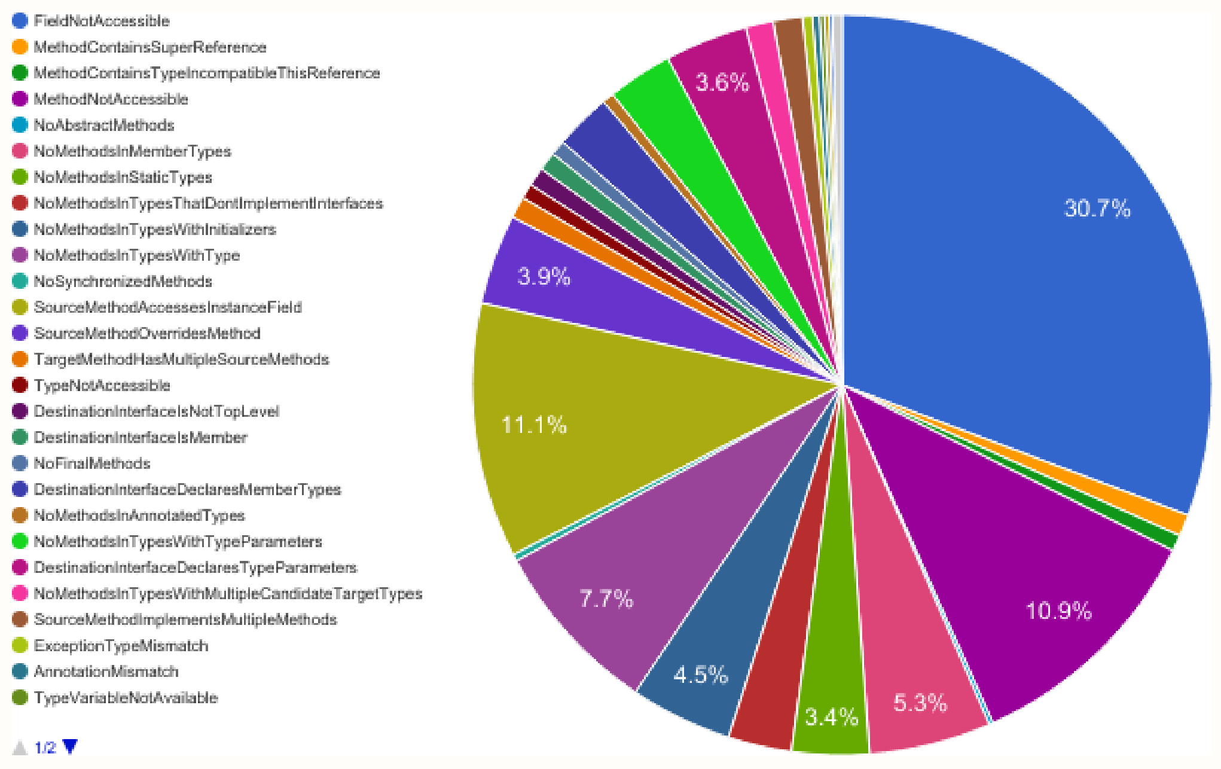

Figure 3: Precondition failures

Preliminary Pull Request Study

Submitted 19 pull requests to Java projects on GitHub, of which 4 have been successfully merged, 5 are open, and 10 were closed without merging. Merged projects totaled 163 watches, 1071 stars, and 180 forks. Projects rejecting requests citing reasons such as that they needed to support older Java clients.

Conclusion

We have presented an efficient, fully-automated, type constraint-based, semantics-preserving approach, featuring an exhaustive rule set, that migrates the skeletal implementation pattern in legacy Java code to instead use default methods. It is implemented as an Eclipse IDE plug-in and was evaluated on 19 open source projects. The results show that our tool scales and was able to refactor, despite its conservativeness and language constraints, 19.63% of all methods possibly participating in the pattern with minimal intervention. Our study highlights pattern usage and gives insight to language designers on applicability to existing software.

References

[1] Joshua Bloch. *Effective Java*. Prentice Hall, 2008.
[2] Brian Goetz. Interface evolution via virtual extensions methods. Technical report, Oracle Corporation, June 2011.
[3] Oracle Corporation. Java Programming Language Enhancements.
[4] Jens Palsberg and Michael I. Schwartzbach. *Object-oriented type systems*. John Wiley and Sons Ltd., 1994.
[5] Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. *ACM TOPLAS*, 33(3):9:1–9:47, May 2011.

International Conference on Software Engineering, May 20–28, 2017, Buenos Aires, Argentina