JavaGI in the Battlefield: Practical Experience with Generalized Interfaces

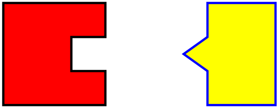
Stefan Wehr   Peter Thiemann

University of Freiburg

October 4, 2009
GPCE 2009, Denver, Colorado

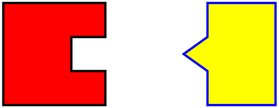
Motivation

Integration of software components is complicated. . .

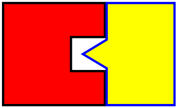


Motivation

Integration of software components is complicated. . .

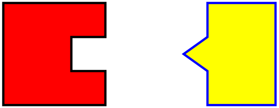


. . . sometimes the interfaces just don’t match!

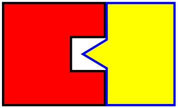


Motivation

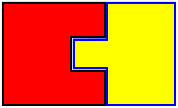
Integration of software components is complicated…



…sometimes the interfaces just don't match!



Adaptability is needed:



JavaGI

- Conservative extension of Java 1.5
- Generalization of Java-style interfaces
- Inspiration: type classes in Haskell
- Initial design proposed at ECOOP 2007 [Wehr et al., 2007]

Example: Database Library

Phillip implements a database library:

```
interface Connection {
  public QueryResult exec(String command);
}

class UseConnection {
  static QueryResult newCustomer(Connection conn,
                                 Customer cust) {
    String command = ...;
    return conn.exec(command);
  }
}
```

Example: Database Library

Phillip implements a database library:

```
interface Connection {
  public QueryResult exec(String command);
}

class UseConnection {
  static QueryResult newCustomer(Connection conn,
                                 Customer cust) {
    String command = ...;
    return conn.exec(command);
  }
}
```

Annette's class for accessing a MySQL database:

```
class MySQLConnection {
  QueryResult execCommand(String command) { ... }
}
```

Feature: Retroactive Interface Implementation

How to combine **Phillip's** library with **Annette's** class?

- Cumbersome in Java ($\Rightarrow$ Adapter Pattern)

Feature: Retroactive Interface Implementation

How to combine **Phillip's** library with **Annette's** class?

- Cumbersome in Java (⇒ Adapter Pattern)
- Simple in JavaGI:

```
implementation Connection [MySQLConnection] {
  QueryResult exec(String command) {
    return this.execCommand(command);
  }
}
```

⟹ `MySQLConnection` is now a subtype of `Connection`!

```
MySQLConnection mySqlConn = ...;
QueryResult result =
  UseConnection.newCustomer(mySqlConn, cust);
```

Example: Expression Evaluation

- Existing datastructure `Expr`

```
interface Expr {}
class IntLit implements Expr {
  int value;
}
class PlusExpr implements Expr {
  Expr left, right;
}
```

- Source code not modifiable
- No Visitor infrastructure anticipated

Feature: Dynamic Dispatch

How to evaluate expressions?

- Difficult in Java

Feature: Dynamic Dispatch

How to evaluate expressions?

- Difficult in Java
- Simple in JavaGI:

```
interface Evaluator { int eval(); } }
implementation Evaluator [Expr] {
  abstract int eval();
}
implementation Evaluator [IntLit] {
  int eval() { return this.value }
}
implementation Evaluator [PlusExpr] {
  int eval() {
    return this.left.eval() + this.right.eval();
  }
}
```

⇒ Bye, bye Visitor pattern!

⇒ Solution to a (restricted version of) the Expression problem

More Features

- Explicit implementing types
  - ⇒ Binary methods
- Type conditionals
  - ⇒ Type-conditional interface implementation
  - ⇒ Type-conditional methods
- Static interface methods
  - ⇒ Abstract over class constructors
  - ⇒ Supersede the Factory pattern
- Multi-headed interfaces
  - ⇒ Family polymorphism
  - ⇒ Multimethods
- Abstract implementation definitions and inheritance
  - ⇒ Partial default implementations
- Dynamic loading of retroactive implementations

Case Study: XPath Evaluation

Task

- Implement a **generic** evaluator for XPath expressions
- Generic means: works for different XML representations

The Java Approach

Jaxen's `Navigator` interface:

```
public interface Navigator {
  Object getParentNode(Object ctxNode)
  String getElementName(Object elem);
  // 39 methods omitted
}
```

The Java Approach

Jaxen's `Navigator` interface:

```
public interface Navigator {
  Object getParentNode(Object ctxNode)
  String getElementName(Object elem);
  // 39 methods omitted
}
```

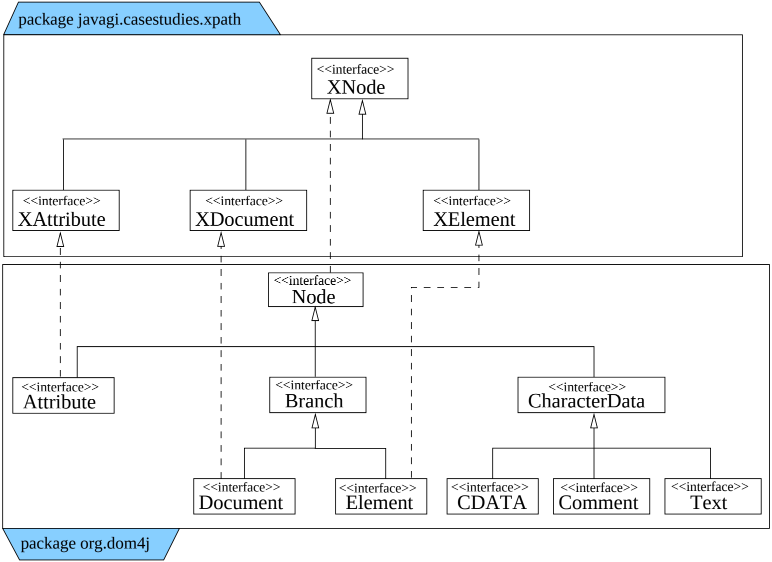
A navigator for dom4j

```
public class Dom4jNavigator implement Navigator {
  public Object getParentNode(Object ctxNode) {
    return ((org.dom4j.Node)ctxNode).getParent();
  }
  public String getElementName(Object elem) {
    return ((org.dom4j.Element)elem).getName();
  }
  // ...
}
                                   // some details omitted
```

The JavaGI Approach

An Interface Hierarchy for XML Nodes

```
public interface XNode {
  XNode getParentNode();
  // 25 methods omitted
}
public interface XElement extends XNode {
  String getName();
  // 2 methods omitted
}
// interfaces XAttribute, XDocument,
// XNamespace, and XProcessingInstruction omitted
```


package javagi.casestudies.xpath
<<interface>>
XNode
<<interface>>
XAttribute
<<interface>>
XDocument
<<interface>>
XElement
<<interface>>
Node
<<interface>>
Attribute
<<interface>>
Branch
<<interface>>
CharacterData
<<interface>>
Document
<<interface>>
Element
<<interface>>
CDATA
<<interface>>
Comment
<<interface>>
Text
package org.dom4j

The JavaGI Approach

Implementing the Node Hierarchy for dom4j

```
implementation XNode [org.dom4j.Node] {
  XNode getParentNode()
    return this.getParent();
  }
  // ...
}
implementation XElement [org.dom4j.Element] {
  String getName() { return this.getName(); }
  // ...
}
                                    // some details omitted
```

XPath Evaluation: Summary

- Adaptations for other XML (and non-XML) libraries possible
- JavaGI version requires no casts:

	Java	JavaGI
dom4j	28	0
JDOM	47	0

- JavaGI version provides better separation of concerns

More Case Studies

JavaGI for the Web

- Web application framework with the same static guarantees as WASH (Web Authoring System Haskell)
- Based on Java Servlet technology
- Essential: dynamic loading of retroactive implementations

More Case Studies

JavaGI for the Web

- Web application framework with the same static guarantees as WASH (Web Authoring System Haskell)
- Based on Java Servlet technology
- Essential: dynamic loading of retroactive implementations

Refactoring of the Java Collection Framework

- Modifying an unmodifiable collection results in a **compile-time** error (instead of a run-time error in Java)
- Heavily influenced by a case study done with cJ [Huang et al., 2007]

Implementation

- Compiler
  - Based on Eclipse Compiler for Java
  - All Java 1.5 Features (Generics, Varargs, . . .)
  - Produces Java bytecode
  - Eclipse plugin
  - Mostly modular typechecking, fully modular code generation
  - Large parts written in Scala
- Run-time system
  - Support for dynamic dispatch on retroactively implemented methods
  - Special cast, `instanceof` und == operations
  - Dynamic loading of retroactive implementations
  - Global consistency checks
- `http://www.informatik.uni-freiburg.de/~wehr/javagi/`

Translating Interfaces and Implementations (1/2)

Recall the `Connection` example

```
interface Connection {
  QueryResult exec(String command);
}

class MySQLConnection {
  QueryResult execCommand(String command) { ... }
}

implementation Connection [MySQLConnection] {
  QueryResult exec(String command) {
    return this.execCommand(command);
  }
}
```

Translating Interfaces and Implementations (2/2)

- Interface definition $\rightarrow$ dictionary interface + wrapper class

```
interface Connection_Dict {
  QueryResult exec(Object this$, String command);
}
class Connection_Wrapper implements Connection {
  Object wrapped;
  Connection_wrapper(Object x) { this.wrapped = x; }
  QueryResult exec(String command) { ... }
}
```

- Implementation definition $\rightarrow$ dictionary class

```
class Connection_MySQLConnection_Dict
    implements Connection_Dict {
  QueryResult exec(Object this$, String command) {
    return ((MySQLConnection)this$).execCommand(command);
  }
}
```

- Actual implementation class untouched

Translating Method Invocations

JavaGI code

```
MySQLConnection mc = ...;
QueryResult r = mc.exec("INSERT ... INTO customer");
```

Translating Method Invocations

JavaGI code

```
MySQLConnection mc = ...;
QueryResult r = mc.exec("INSERT ... INTO customer");
```

Translated code retrieves the dictionary dynamically

```
MySQLConnection mc = ...;
Connection_Dict d = (Connection_Dict)
  javagi.runtime.RT.getDict(Connection_Dict.class, mc);
QueryResult r = d.exec(mc, "INSERT ... INTO customer");
```

Wrapper Classes

Original code

```
QueryResult newCustomer(Connection c, Customer cust){...}
MySQLConnection mc = ...;
QueryResult r = newCustomer(mc, new Customer())
```

Wrapper Classes

Original code

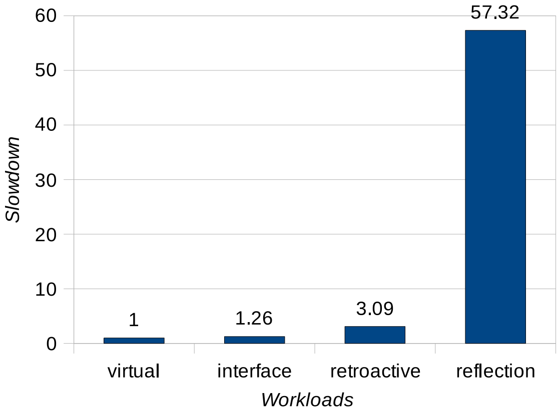
```
QueryResult newCustomer(Connection c, Customer cust){...}
MySQLConnection mc = ...;
QueryResult r = newCustomer(mc, new Customer())
```

Translated code

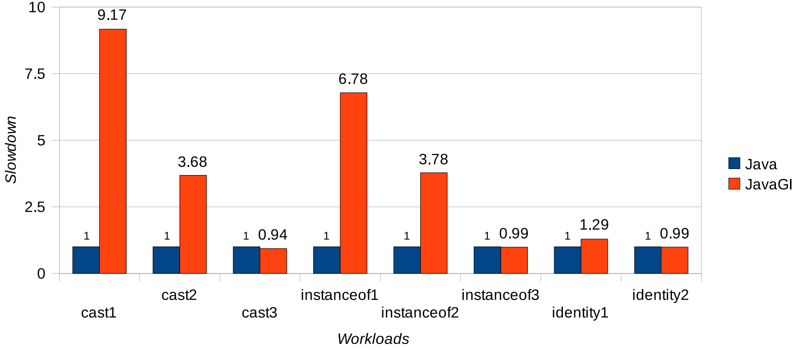
```
QueryResult newCustomer(Connection c, Customer cust){...}
MySQLConnection mc = ...;
QueryResult r = newCustomer(new Connection_Wrapper(mc),
                            new Customer());
```

- Subsumption to an interface type $\overset{\text{(possibly)}}{\Longrightarrow}$ wrapper creation
- Wrap an object at most once
- Casts, **`instanceof`**, and $==$ must be aware of wrappers

Benchmarks: Method Calls



Benchmarks: Casts, `instanceof`, and `==`

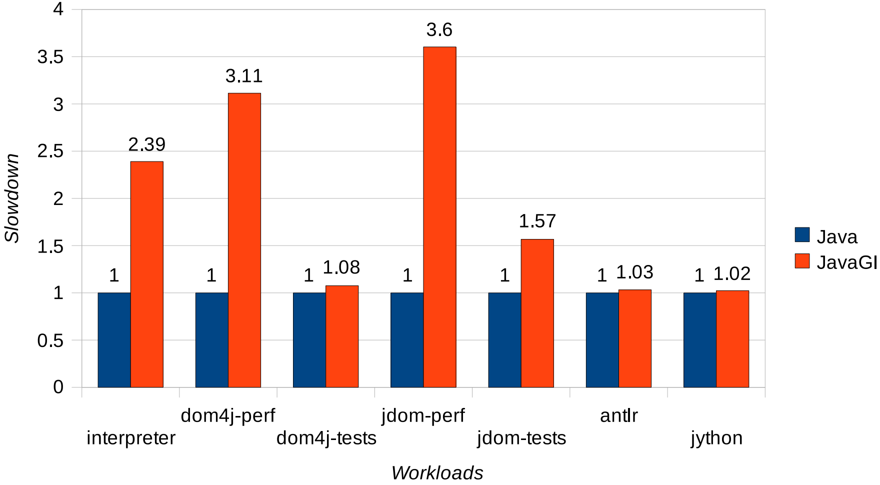


cast1: `Object obj; I i = (I) obj;`

cast2: `Object obj; C c = (C) obj;`

cast3: `D d; C c = (C) d;`

Benchmarks: Real World



Related Work

- Type classes in Haskell [Wadler and Blott, 1989, Hall et al., 1996]
- Software Extension and Integration with Type Classes [Lämmel and Ostermann, 2006]
- MultiJava [Clifton et al., 2000, 2006] and Relaxed MultiJava [Millstein et al., 2003]
- Expanders [Warth et al., 2006]
- Scala [Odersky, 2008]
- `C++` concepts [Gregor et al., 2006]
- ...

Summary

- JavaGI is a conservative extension of Java 1.5
- JavaGI generalizes Java-style interfaces
- JavaGI solves common problems with extension and integration of existing software
- JavaGI has an expressive type system
- JavaGI is implemented
- JavaGI comes with a formalization
- JavaGI Homepage:

`http://www.informatik.uni-freiburg.de/~wehr/javagi/`

References

Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proc. 15th ACM Conf. OOPSLA*, pages 130–145, Minneapolis, MN, USA, 2000. ACM Press, New York. ISBN 1-58113-200-X. doi: http://doi.acm.org/10.1145/353171.353181.

Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM TOPLAS*, 28(3):517–575, 2006. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1133651.1133655.

Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Proc. 21th ACM Conf. OOPSLA*, pages 291–310, Portland, OR, USA, 2006. ACM Press, New York. ISBN 1-59593-348-4. doi: http://doi.acm.org/10.1145/1167473.1167499.

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM TOPLAS*, 18(2):109–138, 1996. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/227699.227700.

Shan Shan Huang, David Zook, and Yannis Smaragdakis. cJ: Enhancing Java with safe type conditions. In *Proc. 6th AOSD*, pages 185–198, Vancouver, BC, Canada, March 2007. ACM Press, New York. ISBN 1-59593-615-7.

Ralf Lämmel and Klaus Ostermann. Software extension and integration with type classes. In *GPCE '06*, pages 161–170, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-237-2. doi: http://doi.acm.org/10.1145/1173706.1173732.

Todd Millstein, Mark Reay, and Craig Chambers. Relaxed MultiJava: Balancing extensibility and modular typechecking. In *Proc. 18th ACM Conf. OOPSLA*, pages 224–240, Anaheim, CA, USA, 2003. ACM Press, New York. ISBN 1-58113-712-5. doi: http://doi.acm.org/10.1145/949305.949325.

Martin Odersky. The Scala language specification version 2.7, April 2008. Draft, `http://www.scala-lang.org/docu/files/ScalaReference.pdf`.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Proc. 16th ACM Symp. POPL*, pages 60–76, Austin, Texas, USA, January 1989. ACM Press.

Alessandro Warth, Milan Stanojevic, and Todd Millstein. Statically scoped object adaptation with expanders. In *Proc. 21th ACM Conf. OOPSLA*, pages 37–56, Portland, OR, USA, 2006. ACM Press, New York. ISBN 1-59593-348-4. doi: http://doi.acm.org/10.1145/1167473.1167477.

Stefan Wehr, Ralf Lämmel, and Peter Thiemann. JavaGI: Generalized interfaces for Java. In Erik Ernst, editor, *21st ECOOP*, volume 4609 of *LNCS*, pages 347–372, Berlin, Germany, July 2007. Springer. ISBN 978-3-540-73588-5.

Feature: Explicit Implementing Types

Binary Methods

```
interface EQ { boolean eq(This that); }
```

- `This` stands for the **class implementing the interface**
- `eq` is a **binary method**:
  type of receiver = type of argument

```
implementation EQ [Integer] {
  boolean eq(Integer that) {
    return this.intValue() == that.intValue();
  }
}
```

```
static <X implements EQ> X find(X z, List<X> lst) {
  for (X y : lst) if (z.eq(y)) return y;
  return null;
}
```

Feature: Explicit Implementing Types

Symmetric Multiple Dispatch

```
implementation EQ [Number] {
  boolean eq(Number that) {
    /* Convert to double and compare. */
  }
}
```

```
    Number d = new Double(3.2);
    Number i = new Integer(1);
    Number j = new Integer(2);
    i.eq(j); // invokes the code for Integer
    i.eq(d); // invokes the code for Number
```

Feature: Explicit Implementing Types

Solution in Java with F-Bounds and Wildcards

```
import java.util.*;
interface EQ<X> { boolean eq(X that); }
class C implements EQ<C> {
    int f;
    C(int i) { this.f = i; }
    public boolean eq(C that) {
        return this.f == that.f;
    }
}
class D extends C {
    D(int i) { super(i); }
}
class Main {
    static <X extends EQ<? super X>> X find(X z, List<X> lst) {
        for (X y : lst) if (z.eq(y)) return y;
        return null;
    }
    public static void main(String[] args) {
        List<D> lst = new ArrayList<D>();
        D d = find(new D(42), lst);
        System.out.println(d);
    }
}
```

Feature: Type Conditionals

Type-Conditional Interface Implementations

```
implementation<X> EQ [List<X>] where X implements EQ {
  boolean eq(List<X> that) {
    Iterator<X> thisIt = this.iterator();
    Iterator<X> thatIt = that.iterator();
    while (thisIt.hasNext() && thatIt.hasNext()) {
      X thisX = thisIt.next();
      X thatX = thatIt.next();
      if (!thisX.eq(thatX)) return false;
    }
    return !(thisIt.hasNext() || thatIt.hasNext());
  }
}
```

Feature: Type Conditionals

Type-Conditional Methods

```
class Box<X> {
  X x;
  ...
  boolean containedBy(List<X> lst)
      where X implements EQ {
    return Lists.find(this.x, lst) != null;
  }
}
```

Feature: Static Interface Methods

```
interface Parseable { static This parse(String s); }

class Parser {
  static <X> List<X> parse(InputStream in)
      throws IOException
      where X implements Parseable {
    BufferedReader br =
      new BufferedReader(new InputStreamReader(in));
    String line;
    List<X> res = new ArrayList<X>();
    while ((line = br.readLine()) != null) {
      X x = Parseable[X].parse(line);
      res.add(x);
    }
    return res;
  }
}
```

Feature: Multi-Headed Interfaces

Family Polymorphism

```
interface ObserverPattern [Subject, Observer] {
  receiver Subject {
    void register(Observer o);
    void notifyObservers();
  }
  receiver Observer {
    void update(Subject s);
  }
}
```

```
  <S,O> void genericUpdate(S subject, O observer)
      where S*O implements ObserverPattern {
    observer.update(subject);
  }
```

Feature: Multi-Headed Interfaces

Symmetric Multiple Dispatch

```
abstract class Shape { ... }
class Rectangle extends Shape { ... }
class Circle extends Shape { ... }
interface Intersect [Shape1, Shape2] {
  receiver Shape1 { boolean intersect(Shape2 that); }
}
implementation Intersect [Shape, Shape] {
  receiver Shape { boolean intersect(Shape that) { ... } }
}
implementation Intersect [Rectangle, Rectangle] {
  receiver Rectangle {
    boolean intersect(Rectangle that) { ... }
  }
}
```

```
  Shape circle = new Circle();
  Shape r1 = new Rectangle(); Shape r2 = new Rectangle();
  r1.intersect(circle);
  r1.intersect(r2);
```

Feature: Dynamic Loading of Retroactive Implementations

```
Class<?> clazz = Class.forName("SQLiteConnection");
SQLiteConnection sqlite =
  (SQLiteConnection) clazz.newInstance()
javagi.runtime.RT.addImplementation(Connection.class,
                                   clazz);
UseConnection.newCustomer(sqlite, new Customer(...));
```

Feature: Abstract Implementations and Inheritance

- Retroactive implementations can be abstract
- Retroactive implementations support inheritance

$\Rightarrow$ useful for providing default implementations

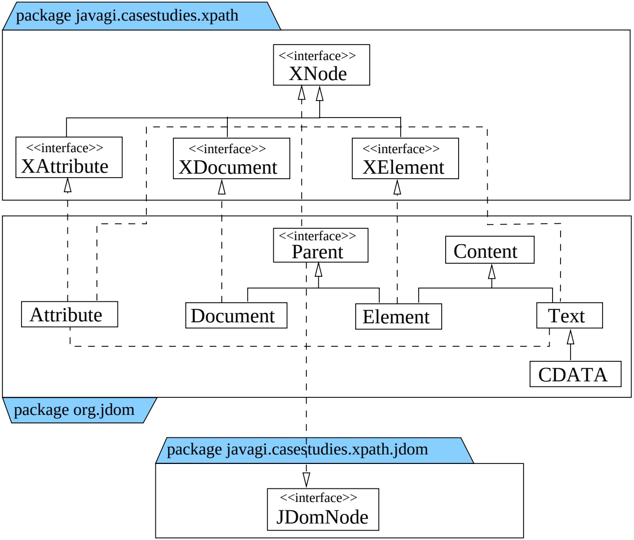
Feature: Abstract Implementations and Inheritance

- Retroactive implementations can be abstract
- Retroactive implementations support inheritance

⇒ useful for providing default implementations

```
interface RList<X> {
  int size();
  boolean isEmpty();
  X elementAt(int i);
}
// A default implementation of RList
abstract implementation<X> RList<X> [RList<X>] {
  boolean isEmpty() { return this.size() == 0; }
}
// A concrete implementation of RList
implementation RList<Character> [String]
    extends RList<Character> [RList<Character>] {
  int size() { return this.length(); }
  Character elementAt(int i) { return this.charAt(i); }
}
```

Case Study: XPath Evaluation for JDOM



Case Study: Refactoring of the JCF

```
public interface List<E,M> extends Collection<E,M> {
  E set(int index, E element) where M extends Modifiable;

  void add(int index, E element) where M extends Resizable;
  boolean add(E o) where M extends Resizable;
  boolean addAll(Collection<? extends E,?> c)
    where M extends Resizable;

  E remove(int index) where M extends Shrinkable;
  boolean remove(Object o) where M extends Shrinkable;
  boolean removeAll(Collection<?,?> c) where M extends Shrinkable;
  boolean retainAll(Collection<?,?> c) where M extends Shrinkable;
  void clear() where M extends Shrinkable;

  // omitted 16 read-only operations such as size(), isEmpty()
}
// Mode types:
public class Modifiable {}
public class Shrinkable extends Modifiable {}
public class Resizable extends Shrinkable {}
```

Translating Method Invocations — A Failed Attempt

JavaGI code

```
MySQLConnection mc = ...;
QueryResult r = mc.exec("INSERT ... INTO customer");
```

- At this point, the compiler could statically construct the dictionary... Could it, really?

Translating Method Invocations — A Failed Attempt

JavaGI code

```
MySQLConnection mc = ...;
QueryResult r = mc.exec("INSERT ... INTO customer");
```

- At this point, the compiler could statically construct the dictionary... Could it, really?
- `mc` might belong to a subclass of `MySQLConnection`
- The subclass might have its own implementation of `Connection`

Translating Method Invocations — A Failed Attempt

JavaGI code

```
MySQLConnection mc = ...;
QueryResult r = mc.exec("INSERT ... INTO customer");
```

- At this point, the compiler could statically construct the dictionary... Could it, really?
- `mc` might belong to a subclass of `MySQLConnection`
- The subclass might have its own implementation of `Connection`
- To serve the correct dictionary, the compiler has to know all subclasses and their implementations. (Or give up dynamic dispatch.)

Translating Method Invocations — A Failed Attempt

JavaGI code

```
MySQLConnection mc = ...;
QueryResult r = mc.exec("INSERT ... INTO customer");
```

- At this point, the compiler could statically construct the dictionary... Could it, really?
- `mc` might belong to a subclass of `MySQLConnection`
- The subclass might have its own implementation of `Connection`
- To serve the correct dictionary, the compiler has to know all subclasses and their implementations. (Or give up dynamic dispatch.)

⇒ **Static dictionary construction is inappropriate**

- Not modular
- Too static
- Makes dynamic loading of retroactive implementations impossible