Code Structures in Java

Itay Maman
Supervisor: Prof. Yossi Gil

Computer Science Department, Technion – Israel Institute of Technology
March 17, 2010

Publications

- Micro Patterns in Java Code
(Gil and Maman, OOPSLA'05)

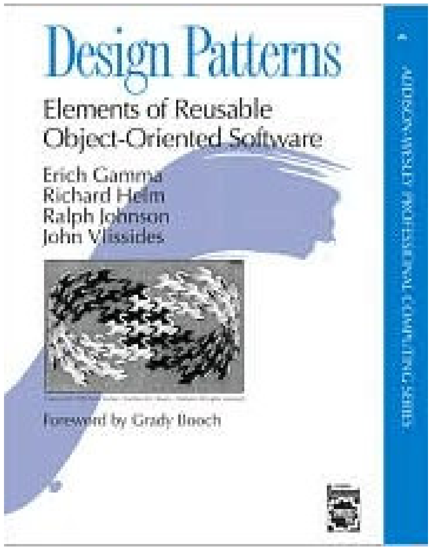
- JTL – the Java Tools Language
(Cohen, Gil and Maman, OOPSLA'06)

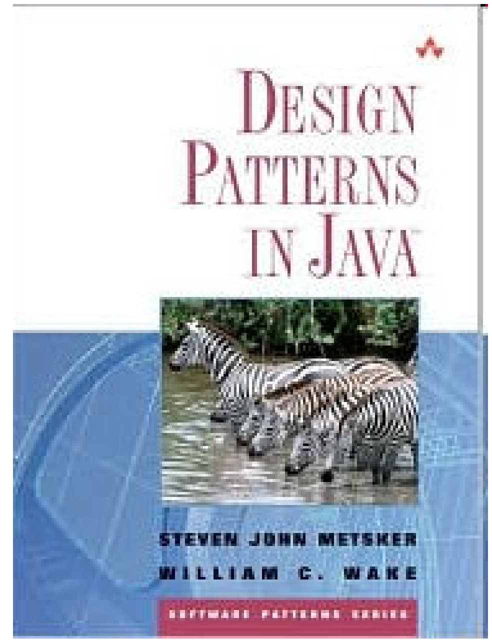
- Whiteoak: Introducing Structural Typing into Java
(Gil and Maman, OOPSLA'08)

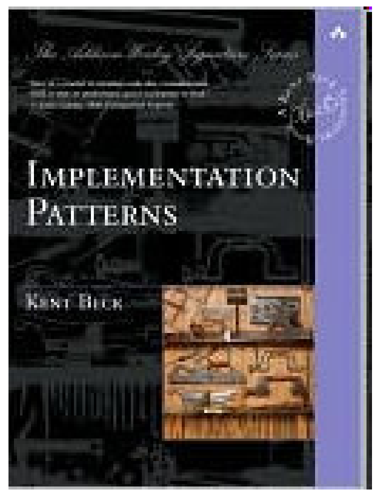
- Guarded Program Transformations using JTL
(Cohen, Gil and Maman, TOOLS-EUROPE'08)

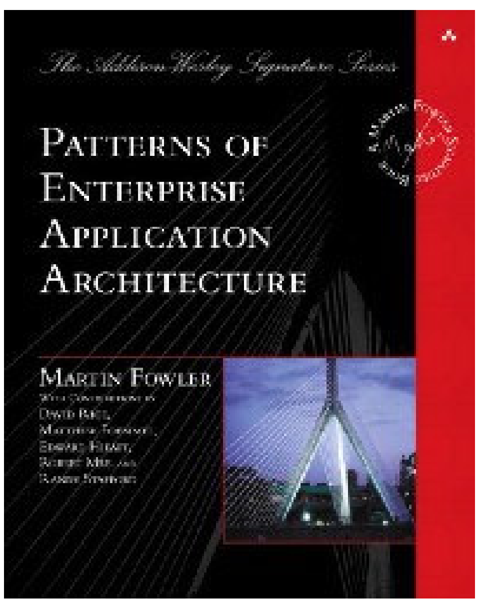
Many thanks to Tal Cohen!

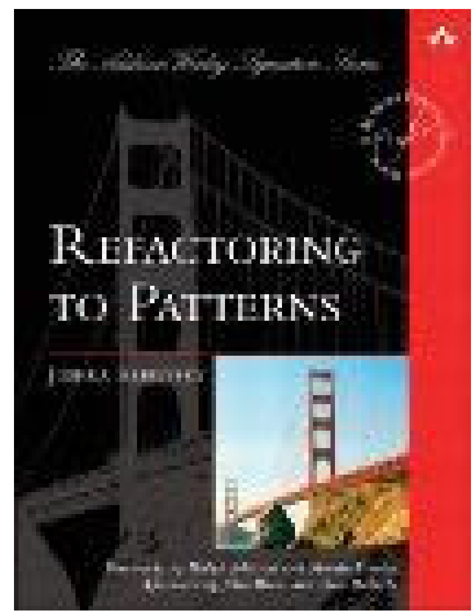
Chapter 1
The Era of Patterns


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


DESIGN PATTERNS IN JAVA
STEVEN JOHN METSKER
WILLIAM C. WAKE
SOFTWARE PATTERNS SERIES


IMPLEMENTATION PATTERNS
KENT BECK


The Addison-Wesley Signature Series
PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE
MARTIN FOWLER


The Addison-Wesley Signature Series
REFACTORING TO PATTERNS

Getters? (I)

```
package java.awt;

public class Component {

  int width;
  int height;
  int x;
  int y;


  public int getWidth() {
    return width;
  }

  public int getHeight() {
    return height;
  }

  ...
}
```

Getters? (II)

```
package java.lnag;

public class Throwable {

  private String detailMessage;

  public String getMessage() {
    return detailMessage;
  }

  public String getLocalizedMessage() {
    return getMessage();
  }

  public StackTraceElement[] getStackTrace() {
    return (StackTraceElement[]) getOurStackTrace().clone();
  }

  private synchronized StackTraceElement[] getOurStackTrace() {
    // Initialize stack trace if this is the first call to this method
    if (stackTrace == null) {
      int depth = getStackTraceDepth();
      stackTrace = new StackTraceElement[depth];
      for (int i=0; i < depth; i++)
        stackTrace[i] = getStackTraceElement(i);
    }
    return stackTrace;
  }

  ...

}
```

Sampler

- A class that provides pre-made instances of itself
- (A relaxed singleton)

Sampler (I)

"A class that provides pre-made instances of itself"

```
public class Color implements Paint, java.io.Serializable {

  public final static Color white = new Color(255, 255, 255);
  public final static Color gray = new Color(128, 128, 128);
  public final static Color red = new Color(255, 0, 0);

    …
}
```

Sampler (II)

"A class that provides pre-made instances of itself"

```
public interface CookiePolicy {
  public static final CookiePolicy ACCEPT_ALL = new CookiePolicy(){
    public boolean shouldAccept(URI uri, HttpCookie cookie) {
      return true;
    }
  };

  public static final CookiePolicy ACCEPT_NONE = new CookiePolicy(){
    public boolean shouldAccept(URI uri, HttpCookie cookie) {
      return false;
    }
  };

  public boolean shouldAccept(URI uri, HttpCookie cookie);
}
```

Sampler (III)

"A class that provides pre-made instances of itself"

```
static class MouseEventTargetFilter implements EventTargetFilter {

  static final EventTargetFilter FILTER = new MouseEventTargetFilter();

  private MouseEventTargetFilter() {}

  public boolean accept(final Component comp) {
    return (comp.eventMask & AWTEvent.MOUSE_MOTION_EVENT_MASK) != 0
        || (comp.eventMask & AWTEvent.MOUSE_EVENT_MASK) != 0
        || (comp.eventMask & AWTEvent.MOUSE_WHEEL_EVENT_MASK) != 0
        || comp.mouseListener != null
        || comp.mouseMotionListener != null
        || comp.mouseWheelListener != null;
  }
}
```

Sampler (IV)

"A class that provides pre-made instances of itself"

```
public class Proxy implements java.io.Serializable {

  private static Object pendingGenerationMarker;
  private static Object nextUniqueNumberLock;

  // additional fields & methods ...
}
```

Sampler (V)

"A class that provides pre-made instances of itself"

- "class" – Are interfaces or enums allowed?
- "provides" – Inherited fields? Inherited private fields?
- "itself" – Superclass? Superinterface? Indirect super type? Object?

- Difficulties
  - Imprecise terms

Code Structure

A formal condition on the type, name, attributes, and sub-elements, of a software element

1. Purposeful: fulfill a useful programming need
2. Limiting: restrict the design space variety
3. Mechanically recognizable: by an automatic checker
4. Simple: human understandable

Wishful Thinking

```
class implements Iterable {
   public void sort();
   private int size();
   private int n;
}
```

```
abstract class {
  no private method;
}
```

```
class extends* Date
```

Chapter 2

The Java Tools Language

Mission Statement

- A language that supports such expressions
- Not based on templates
- Full power of (recursive) relational algebra

JTL: Unary Predicates

- A simple predicate:

```
const := static final field;
```

  - Three sub-queries
  - Space (and comma) denotes conjunction

- The subject variable

```
const := #static #final #field;
```

  - The hash symbol, #, denotes the subject variable
  - A variable in a prefix position => becomes the subject of the callee
    - If not specified, defaults to # (the subject)

★ `static`, `final`, etc. are library predicates, not keywords.

JTL: Binary Predicates

```
has_static_member X := class, X static, declares X;
extends+ Y := extends Y | extends Y', Y' extends+ Y;
```

- Accepts
  - A subject
  - An additional parameter (in a postfix position)
- Syntax
  - Variables/Parameters: begin with a capital letter
  - Predicates: begin with non-captial
- Semantics
  - Evaluation of a (n-ary) predicate yields a (n-ary) relation
  - Each tuple position corresponds to a parameter (incl. the subject)

JTL: Quantification { }

- Quantifying the members of a class
  - Subject variable iterates over a set
  - Each condition must hold
  - Set conditions:
    - Unary: `exists, all, no, one, many`
    - Binary: `->` (implies)

```
some_class := class {
   exists static int field;
   many method;
}
```

- Quantifying with a generator

Generator

```
no_abstract_super := class extends+: {
   no abstract;
}
```

- Default generator for classes: `members:`
- Default quantifier: `exists`

Other Features

```
sortable_list := class extends /java.util.ArrayList {
   static final long field 'SerialVersionUID;
   public void sort();
}
```

Other Features

```
sortable_list := class extends /java.util.ArrayList {
   static final long field 'SerialVersionUID;
   public void sort();
}
```

Type literal

Name matching
(Regular expression)

Method signature
pattern

Kinds (Type System)

- JTL’s Kinds: PACKAGE, TYPE, MEMBER
  - And a few others

- Observation: Kind errors => non-satisfiable predicate
  - (Permanently yields an empty results)

```
p X := extends X, X method;
```

"find all pairs <#,X> such that # extends X and X is a method"

- Can we detect these without evaluating the predicate?

Kind Ineference

- The idea:
  - A primitive predicate has a predefined “signature”
  - Signature: Several tuples, specifying kinds of parameters
  - Same arity as the predicate
  - Rules for deducing signature of compound predicates
- Unary
  - class :: { <TYPE> }
  - method :: { <MEMBER> }
- Binary
  - extends :: { <TYPE,TYPE> }
- Unary, overloaded
  - abstract :: { <TYPE>, <MEMBER> }

A Kind-Correct Predicate

```
p X := class extends X, X abstract;
```

- Expand each signature to accommodate all variables
  - #class :: { <#=TYPE, X=?> }
  - #extends X :: { <#=TYPE, X=TYPE> }
  - X abstract :: { <#=?, X=TYPE>, <#=?, X=MEMBER> }

- Join these signatures (treat them like relations)
  - { <#=TYPE, X=TYPE> }

A Kind-Incorrect Predicate

```
p X := class extends X, X method;
```

- Expand each signature to accommodate all variables
  - #class :: { <TYPE,?> }
  - #extends X :: { <TYPE,TYPE> }
  - X method :: { <?,MEMBER> }

- Join these signatures
  - {}

Kind Inference: Disjunction

```
p X := extends X | declares X;
```

- Expand each signature to accommodate all variables
  - #extends X :: { <TYPE,TYPE> }
  - #declares X :: {<TYPE,MEMBER> }

- Union these signatures
  - { <TYPE,TYPE>, <TYPE,MEMBER> }

Kind Inference: Summary

- Primitive predicate:
  - A relation (of kinds) is predefined
  - Same arity as the predicate
- Compound predicate
  - Conjunction -> Join
  - Disjunction -> Union

Kind Inference: Summary

- Primitive predicate:
  - A relation (of kinds) is predefined
  - Same arity as the predicate
- Compound predicate
  - Conjunction -> Join
  - Disjunction -> Union
- Same rules as those of evaluation* !!
  - Kind inference == evaluating with a different set of primitive relations
  - Calculating the “not-necessarily empty” property
  - (*Almost)

Delving into Methods

- The idea: Symbolically name intermediate results

- The SCRATCH type
  - Constants
  - Values passed to parameters
  - Values pushed onto the stack by JVM instructions

- Represent code sites

- Library predicates for checking whether a scratch is:
  - Copied
  - Used in a calculation
  - Written/Read from a field
  - Stored/loaded from the local variable array
  - ...

Scratches in Actions

```
chain_method := !void instance method {
   returned -> this;
};

setter F := void instance method (_) {
   this put_field[F,V], F field, V parameter;
};
```

★ Default generator expression for a method is: `scratches:`

- Generates all scratches of the method

Chapter 3

Discovering Code Structures

Micro-Patterns

- Class-level Code structures
- Similar to design patterns, but:
  - Mechanically recognizable
  - Stand at a lower level of abstraction
- A catalog of 28 micro patterns
  - Empirically discovered
- Expressed as JTL expressions
  - Precision
  - Ability to explore behavior of methods

Compound Box

A class whose list of members (including inherited ones) includes exactly one non-primitive instance field, and one or more primitive instance fields.

Definition

```
compound_box := offers: {
  one !primitive instance field;
  primitive instance field;
};
```

Purpose

This is a variant of a Box pattern where that most of the state is provided by the non-primitive field and auxiliary, bookkeeping data, is maintained by the primitive fields.

Example

```
public class ArrayList<E> extends AbstractList<E> implements List<E>,
    RandomAccess, Cloneable, Serializable {
  private transient Object[] elementData;
  private int size;
  private static final long serialVersionUID = ...;

  public ArrayList(int initialCapacity) { ... }
  public ArrayList() { this(10); }
  public E get(int index) { ... }
  ...
}
```

Prevalence

4.4%

Common State

A class whose list of members (excluding those defined by Object) includes no instance methods, no instance fields, and at least one non-final static field.

Definition

```
common_state := class non_global_members: {
    no instance field;
    exists !final static field;
    no instance method;
  };
```

Purpose

Unlike Stateless classes, Common State classes maintain state, but this state is shared by all of their instances. A Common State with no instance methods is in fact an incarnation of the packages mechanism of Ada.

Example

```
public final class System {
  private System() {}
  public final static InputStream in = nullInputStream();
  public final static PrintStream out = nullPrintStream();
  public final static PrintStream err = nullPrintStream();

  private static volatile Console cons = null;
  private static volatile SecurityManager security = null;
  public static void setIn(InputStream in) { ... }
  public static Console console() { ... }
  ...
}
```

Prevalence

3.8%

Additional Micro-Patterns

- Immutable: a class whose state never changes
- Sampler: offers customers a collection of pre-made instances
- Sink: does not propagate calls
- Stateless: carries no state information
- Designator: an empty interface
- Implementor: gives body to abstract methods, without introducing any new methods.

★ **Not mutually exclusive**

★ **Discovered, not invented**

Five Most Popular Patterns

	Jedit	Scala	Shared	Jboss	Poseidon	Tomcat	Sun-14	AVG
Canopy	26.5%	3.9%	5.2%	6.2%	10.3%	4.6%	9.8%	9.5%
PureType	2.5%	20.5%	11.2%	11.3%	11.9%	5.6%	7.7%	10.1%
Overrider	23.1%	4.1%	10.4%	7.0%	16.8%	20.2%	12.4%	13.4%
Sink	9.0%	14.0%	15.0%	12.9%	11.3%	12.1%	20.6%	13.5%
Implementor	37.1%	10.5%	16.8%	23.0%	22.1%	12.7%	26.1%	21.2%
Coverage	70.6%	48.6%	51.8%	54.0%	61.9%	50.6%	61.4%	55.2%
Total Coverage	83.7%	79.4%	65.7%	76.2%	76.9%	67.3%	79.5%	75.5%

Findings (Highlights)

- 90%-10% principle
  - Most classes use a mere fraction of Java's design space
- 45% of all classes are trivial
  - One in ten classes is a wrapper of a single instance field
  - One in seven classes has no instance state
  - One in seven classes has immutable instance state
  - One in seven classes is a sink
- A handy mechanism for measuring software

Applications

– Automatic Identification of Bug-Introducing Changes (Kim et-al, ASE'06)

– Decrypting The Java Gene Pool Predicting objects' lifetimes with micro-patterns (Marion, Jones, and Ryder, ISMM'07)

– Exploiting the Correspondence between Micro Patterns and Class Names (Singer and Kirkham, SCAM'08)

– Sourcerer: a search engine for open source code supporting structure-based search (Bajracharya et-al, Data Mining and Knowledge Discovery Journal, Oct.08)

Exploiting Code Structures

- Two directions
  - Capture useful patterns as keywords
  - JTL-defined types

Chapter 4
Whiteoak

Higher Degree of Resue

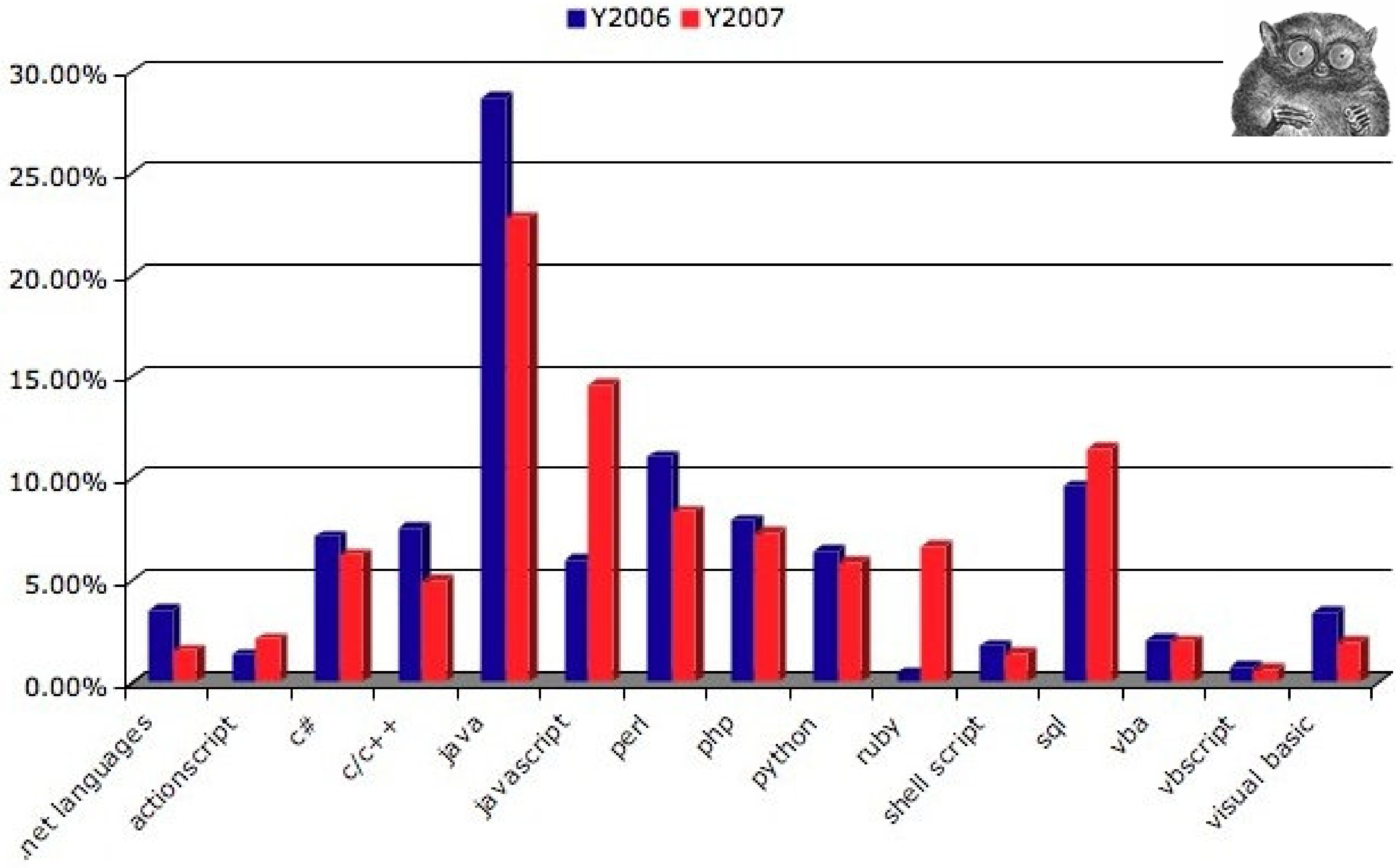
```
public class SomeProgram {

   public static void printRed({ int getRed(); } x) {
     System.out.println(x.getRed());
  }

  public static void main(String[] args) {
     printRed(Color.YELLOW);

     printRed(new Object() {
       public int getRed() { return 100; }
     });
  }
}
```

No need to retroactively adapt Color's definition

Is this Needed?

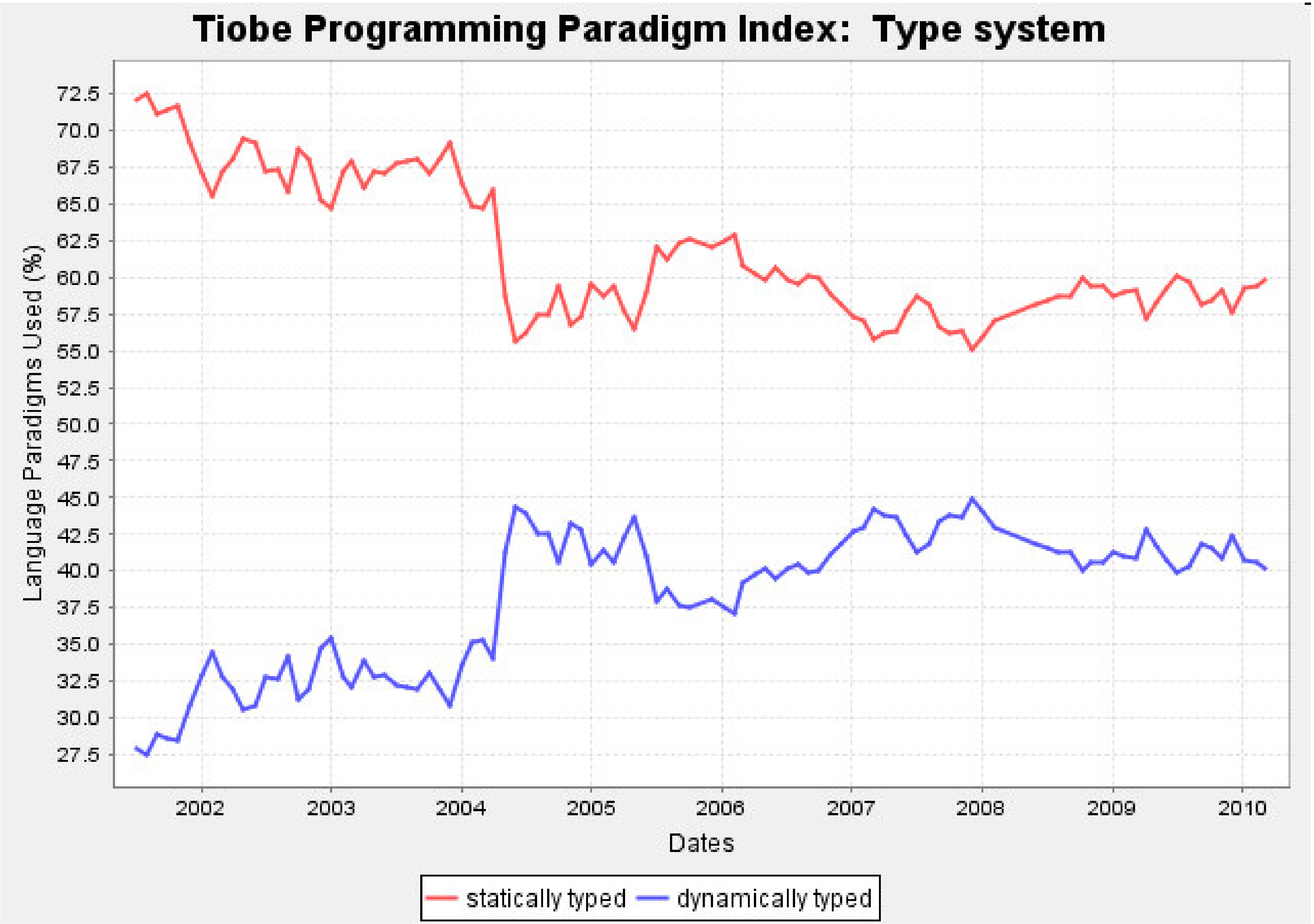
- Go language
- Scala (Odersky et-al)
  - The next Java?
  - Recently introduced structural subtyping

Popularity: O'reilly Book Sells



Source: http://radar.oreilly.com/archives/2007/03/programming_lan.html

Popularity: O'reilly Book Sells



TIOBE Programming Community Index for March 2010

?

Java vs. Whiteoak

```
public interface Node {
  public int degree();
  public Node getNeighbor(int i);
}

public class Graph {
  private class NodeImpl implements Node { ... }
  public static Node parse(String s) { ... }
}
```

- Our mission:

  Introduce a getFirstChild(int) method

X is a subtype of S

```
public struct S {
  public int value;
  public constructor(int);
  public int add(int);
}

public class X {
  public int value;
  public X(int n) { value = n; }
  public int add(int n) { return value + n; }
}
```

Y is not a subtype of S

```
public struct S {
  public int value;
  public constructor(int);
  public int add(int);
}

public class Y {
  private int value;
  public Y(int n, double d) { value = n; }
  public int addTo(int n) { return value + n; }
}
```

Runtime Semantics

```
X x = new X(5);

S s = x;                // s – static type: S,
                        //     dynamic type: X

s.add(3);               // Effectively: x.add()

s.value                 // Effectively: x.value

s.constructor(10);      // Effectively: new X(10)
```

Extending Behavior

```
public struct MyNode {
  public int degree();
  public Node getNeighbor(int i);

  public MyNode getFirstChild() {
    return degree() < 1 ? null : getNeighbor(0);
  }
}
```

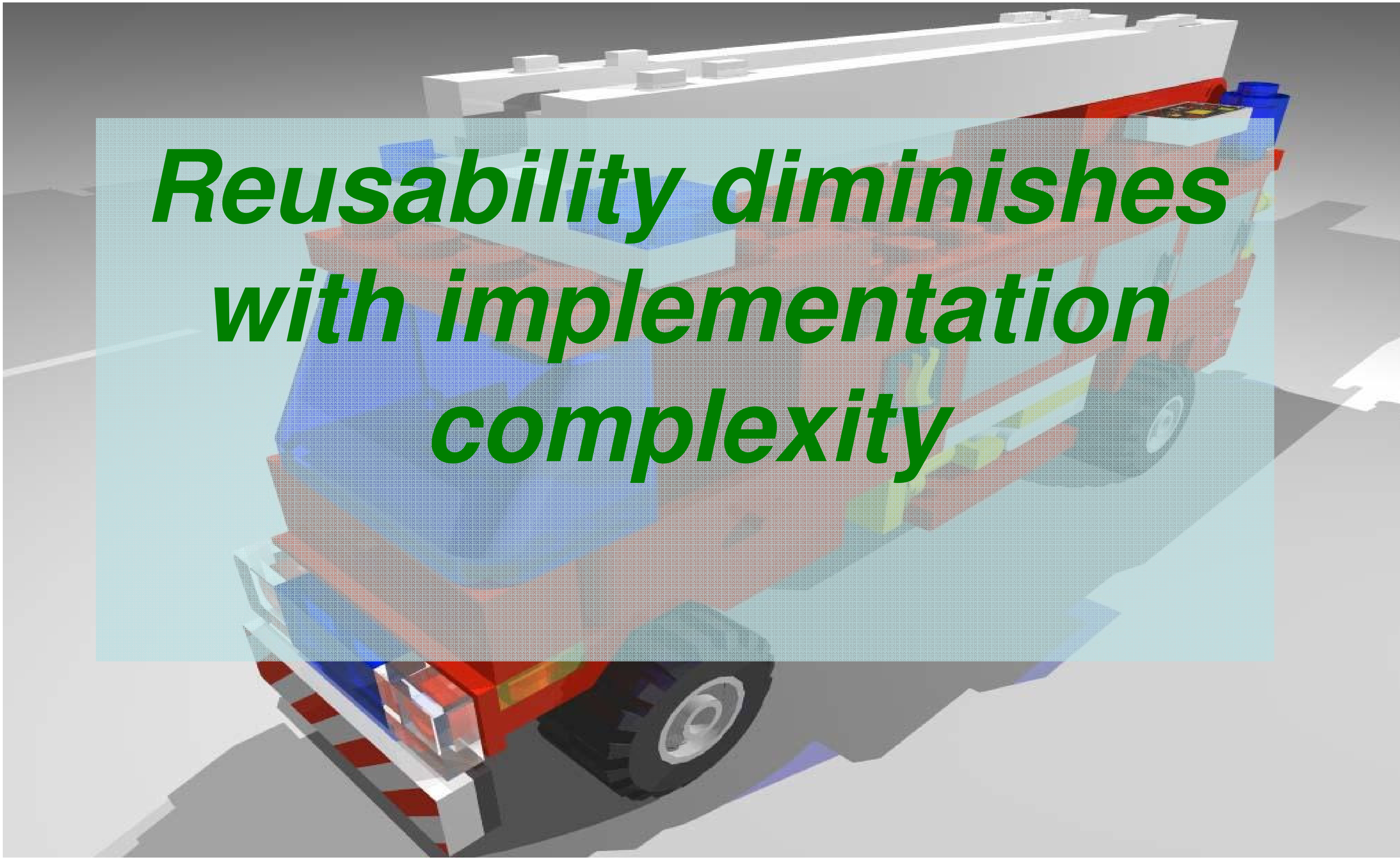
Whiteoak Solution

```
public struct MyNode {
  public int degree();
  public Node getNeighbor(int i);

  public MyNode getFirstChild() {
    return degree() < 1 ? null : getNeighbor(0);
  }
}
```

Java Solution

```
public abstract class DecoratedNode<T extends Node> implements Node {
  protected abstract T create(Map<Node, WeakReference<T>> m, Node n);
  private Map<Node, WeakReference<T>> map;
  private Node inner;
  protected DecoratedNode(Map<Node, WeakReference<T>> map, Node n) {
    inner = n;
    this.map = map;
  }
  public int degree() { return inner.degree(); }
  public T getNeighbor(int i) {
    Node n = inner.getNeighbor(i);
    synchronized(map) {
      WeakReference<T> r = map.get(n);
      T t = r == null ? null : r.get();
      if(t == null) {
        t = create(map, n);
        map.put(n, new WeakReference<T>(t));
      }
      return t;
    }
  }
}

public class MyNode extends DecoratedNode<MyNode> {
  public MyNode(Map<Node, WeakReference<MyNode>> map, Node n) {
    super(map, n);
  }
  protected MyNode create(Map<Node, WeakReference<MyNode>> m, Node n) {
    return new MyNode(m, n);
  }
  public MyNode getFirstChild() {
    return degree() < 1 ? null : getNeighbor(0);
  }
}
```

The Lego Conjecture



Reusability diminishes with implementation complexity

Related Work

- Patterns as signs
- JQuery
- Using code structures
  - Sung Kim
  - Names verbs

- Scala’s structural types

Evaluation scheme

- JTL is translated into Datalog
- The Datalog program is evaluated top-down

- “need to know basis”
  - The JTL evaluation algorithm does not see more information than it needs
  - Only necessary facts are extracted from the native predicates

=> No database setup is needed
(a bottleneck in similar applications)

Q: What's special about p2?

```
p1 := # extends A, A abstract;

p2 := A extends #, A abstract;

p3 := A extends #, A abstract, # p4 A;

p4 := ...
```

(Suppose we invoke p1, p2, p3 with some type passed in as a subject)

- A: Result of p2 is infinite
  - Result of p3 may be infinite (depending on the definition of p4)

- Consequences
  1. Use exhaustive iteration
     - Assume a closed world, usually: the classpath
     - Performance penalty
  2. Beware: Fragile results!!

Fragility

- Can we determine whether a query is fragile?
  - In the general case: No
  - In JTL's case: Yes
    - Requires the native predicates to maintain a simple property
    - (See Cohen Gil and Zarivach 2007)

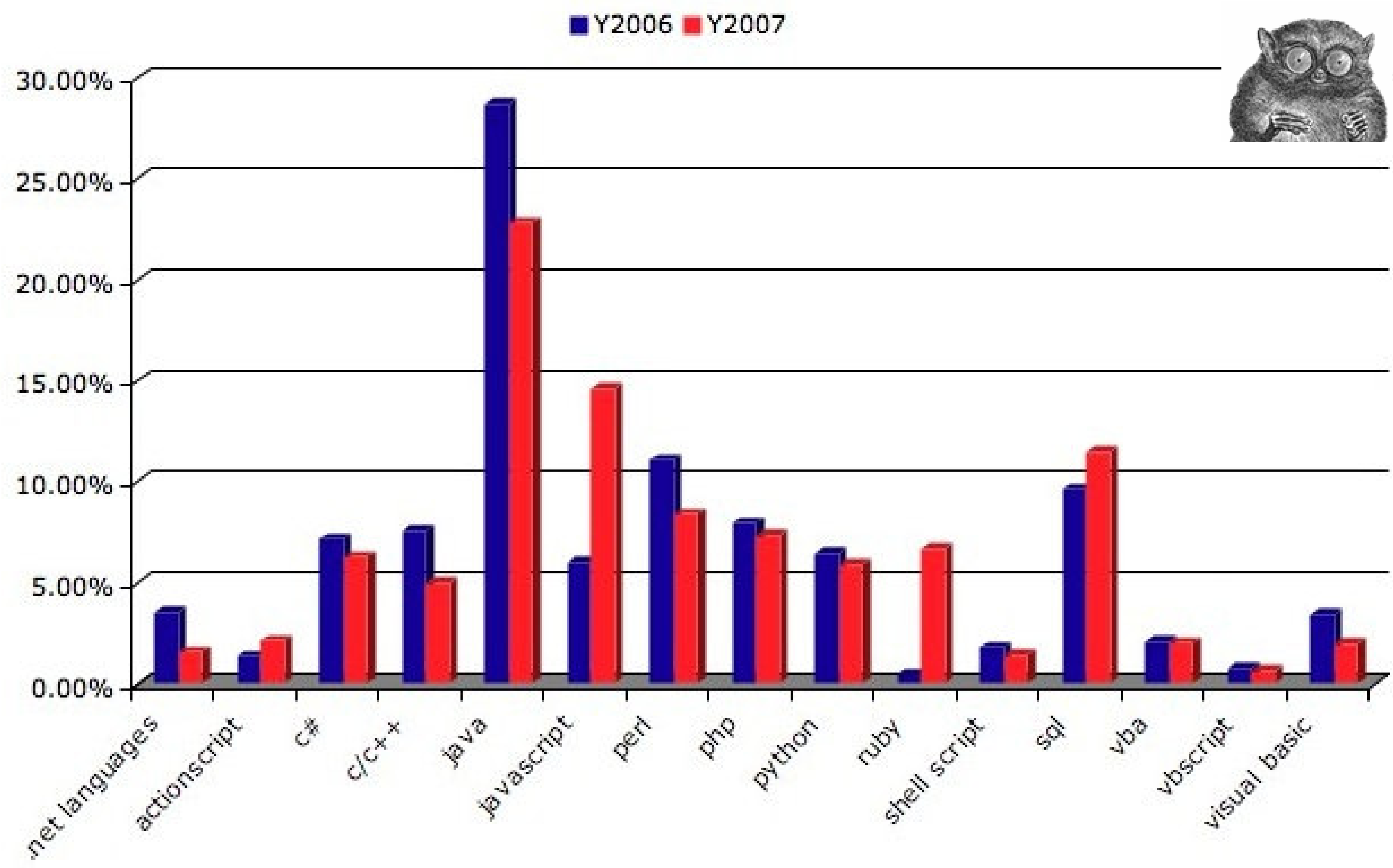
- General idea
  - Define "computability" for each native predicate
  - Rules for computing "computability" of compound predicates

Implementation

```
// Original program

struct S { void f(); }
class C { void f() { } }

void someMethod() {
  S s = new C();
  ...
  s.f();
}
```

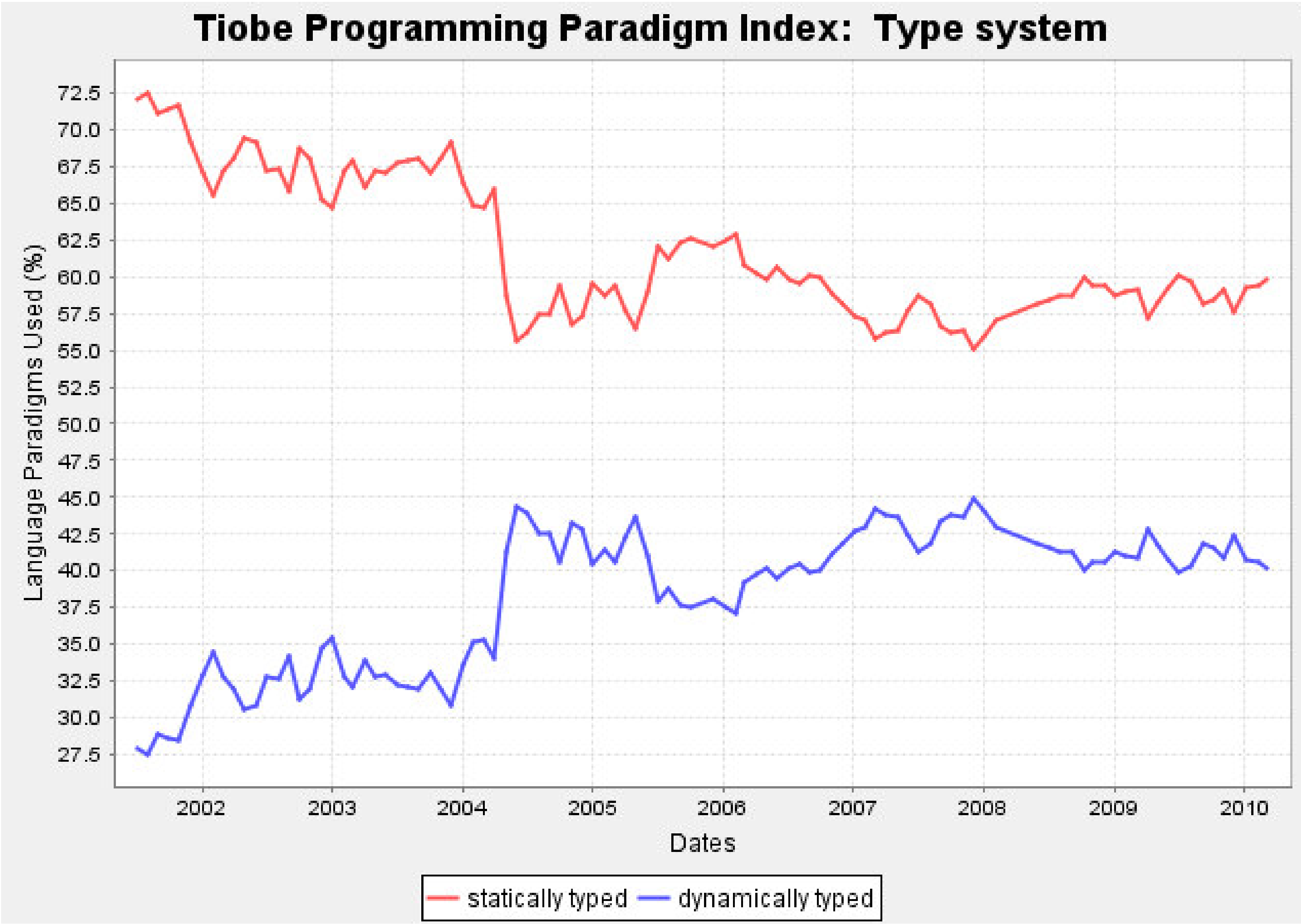
```
// Compiled program

interface S { void f(); }

void someMethod() {
  Object s = new C();
  ...
  // C <- dynamic type of s
  // create an S-to-C adapter
  // invoke f() on the adapter
}

Class S_C implements S {
  private C c;
  S_C(Object o) { c = (C) o; }
  public void f() { c.f(); }
}
```

Popularity: O'reilly Book Sells



Source: http://radar.oreilly.com/archives/2007/03/programming_lan.html

Popularity: O'reilly Book Sells



TIOBE Programming Community Index for March 2010

Patterns Multiplicity