Language design smells

or: deconstructing language design (?)

Tijs van der Storm

CWI
Centrum Wiskunde & Informatica

```
PLEASE KNOCK BEFORE ENTERING

(1000) PLEASE IGNORE .4
PLEASE ABSTAIN FROM (1005)
(1009) DO STASH .1 + .2 + .5 + .6
DO .4 <- #1
DO (1004) NEXT
(1004) PLEASE FORGET #1
DO .3 <- '¥.1¢.2'~'#0¢#65535'
DO .6 <- '&.1¢.2'~'#0¢#65535'
PLEASE DO .5 <- "¥!6~#32768'¢#1"~#3
DO (1002) NEXT
DO .4 <- #2
(1005) DO (1006) NEXT
(1999) DOUBLE OR SINGLE PRECISION OVERFLOW
(1002) DO (1001) NEXT
(1006) PLEASE FORGET #1
DO .5 <- '¥"!6~.6'~#1"¢#1'~#3
DO (1003) NEXT
DO .1 <- .3
```

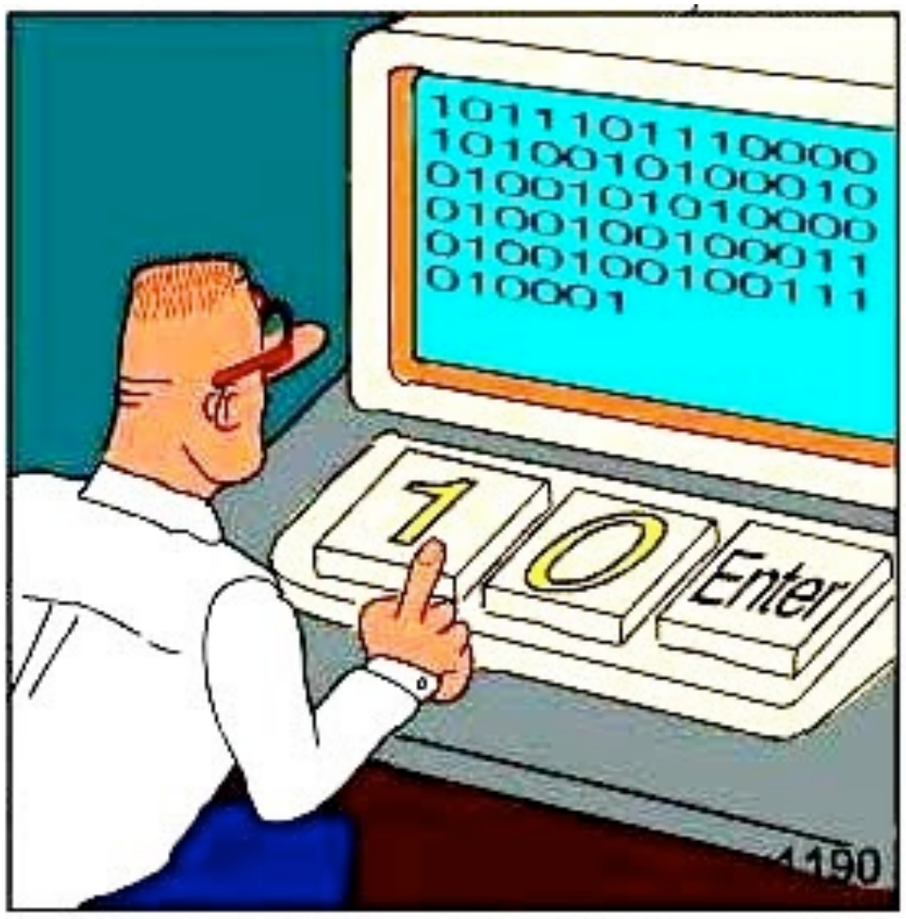
Which one is better designed?

Why language design smells?

- More language design now than ever (DSLs)
- Teaching language design
- Body of knowledge
- Common vocabulary
- Making implicit knowledge of experts explicit

Language as user interface

- NOT implementation



REAL Programmers code in BINARY.

do Considered od:
A Contribution to the Programming Calculus*

Eric C.R. Hehner

Computer Systems Research Group, University of Toronto, Toronto M5S 1A4, Canada

Summary. The utility of repetitive constructs is challenged. Recursive refinement is claimed to be semantically as simple, and superior for programming ease and clarity. Some programming examples are offered to support this claim. The relation between the semantics of predicate transformers and "least fixed point" semantics is presented.

Eric C. R. Hehner, *Acta Informatica*, 11(4), 287-304, 1979

do Considered od:
A Contribution to the Programming Calculus*

Eric C.R. Hehner

Computer Systems Research Group, University of Toronto, Toronto M5S 1A4, Canada

Summary. The utility of repetitive constructs is challenged. Recursive refinement is claimed to be semantically as simple, and superior for programming ease and clarity. Some programming examples are offered to support this claim. The relation between the semantics of predicate transformers and "least fixed point" semantics is presented.

Eric C. R. Hehner, *Acta Informatica*, 11(4), 287-304, 1979

"**do** Considered **od**" Considered Odder than "**do** Considered **ob**"

David Harel

IBM T. J. Watson Research Center

Yorktown Heights, NY 10598

begin

Noticing that the exact reflection of "**do**" is not "**od**" but "**ob**", we suggest in this one sentence paper that the popular proposal of using symmetric pairs of keywords in programming languages be followed to the letter, so to speak.

niged

References (Note: reference [1] is not cited in the text.)

[1] E. C. R. Hehner, "**do** Considered **od**: A Contribution to the Programming Calculus", *Acta Informatica 11*, pp. 287-304 (1979).

David Harel, *ACM SIGPLAN Notices*, 15(4), 1980

A Further Note on Symmetric Keyword Pairs

Richard Hamlet
Department of Computer Science
University of Maryland
College Park 20742

David Harel's **do ... ob** paper[1] is compelling, but difficult to implement on limited-font equipment. Without true character-reversing the scheme is case- and font-sensitive: neither **DO ... OB** nor ***do ... ob*** are satisfactory. In this note I will stick to what most terminals can type.

Harel does not go far enough toward symmetry. Once the keyword pairs are truly symmetric, what of the material enclosed? Is not

```
do
    x := 0
    0 =: x
ob
```

attractive? (And note that a proper choice of variables and constants yields true symmetry.) Anyone can see where these remarks are heading: we should be writing palindromes over an alphabet whose symbols are their own reversals. (Accidental reversal pairs like **d - b** are a complication that should be ignored.) The effect of such a change on parsing courses and textbooks in computer science should be entirely salutory.

While we are at it, putting the uglier of two keywords at the *end* of a construction has always seemed to express a foolish optimism (start with the good, perhaps the bad will not happen). Programmers should be pessimists, and prefer (say)

BASIC

```
TXEN i = 1 TO 10
  ...
NEXT i
```

FORTRAN

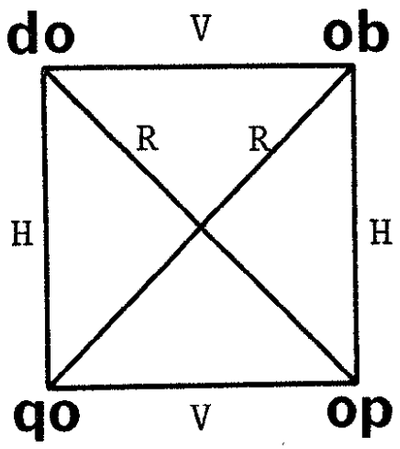
```
      EUNITNOC 100 I = 1,10
        ...
001   CONTINUE
```

Richard Hamlet, *ACM SIGPLAN Notices*, 15(6), 1980

do CONSIDERED REFLECTIVELY: A CONTRIBUTION TO GROUP PROGRAMMING

LeRoy Johnson
School of Computer Science
University of New Brunswick
Fredericton, N.B., Canada

In this one paragraph, we do extend the reflective work of Harel [2]. Insightful as this work is we feel that Harel in his rush to publication was too quick to do the obvious; we further notice that the top down reflection of do is qo and the compound Harel-Johnson reflection gives op. The relationships of these do reflections are expressed in Fig. 1(a). It is not surprising that previous work is unnecessarily restricted to one dimension since current use of the do is essentially based on the linear line, however, we believe two dimensional programming is now next in line for development, and this motivated our present paragraph. Extension to higher dimensions does not appear fruitful if we require closure in the Roman alphabet, thus, the remainder of this paragraph is restricted to two dimensions. The observation so far might be mere curiousity except for the fact that horizontal and vertical reflection generate the Klein 4 group when it is realized that their product on do is equivalent to a rotation of 180° on do. Interestingly, the result of an even number of reflections on any element of Fig. 1(a) is independent of the order of reflection. In other words you can even do programming without logical thinking, unless you are odd.



(a) do symmetries in the alphabet

	λ	V	H	R
λ	λ	V	H	R
V	V	λ	R	H
H	H	R	λ	V
R	R	H	V	λ

(b) Klein 4 group

Figure 1 V - vertical reflection
H - horizontal reflection
R - rotation 180°

Leroy Johnson, *ACM SIGPLAN Notices*, 15(12), 1980

do CONSIDERED obVIOUSLY odd IN THREE DIMENSIONS

LeRoy Johnson
School of Computer Science
University of New Brunswick
Fredericton, N.B., Canada

In this two paragraph paper, we hasten to extend both our results and an appology to Harel for chiding him on his lack of reflection. The pot has called the kettle black, and, indeed, potted we were undoubtly when we penned our previous paper. In Johnson [4], we claimed that extension of the do results to higher dimensions was not fruitful, for instance a rotation of 90° takes do out of the Roman alphabet. This incorrect assumption was due to a lack of reflection on our part. On rereading the interchange between Harel [2] and Hehner [3] we were able to decouple the confounding of reflection in the interchange and obtain more solid results. Indeed no reflection is required to see that od is the interchange of do.

In Fig. 1 the interchange and reflection operations are properly related in three dimensions. Clearly, do by prior use should occupy the zero axis, thus an application of the mapping of Fig. 1. to do will generate the relationships depicted by the cube of Fig. 2. It is obvious that we can oddly enough also form the Klein 4 group of Johnson [4] on od and that these two groups are related by interchange; this then naturally extends the do mappings to three dimensions. Extensions to dimensions higher than 3 appear computationally intractable on paper, and we leave this to other researchers as an open problem, for we can do no more.

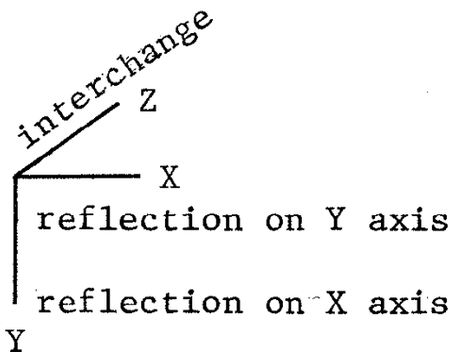


Fig. 1

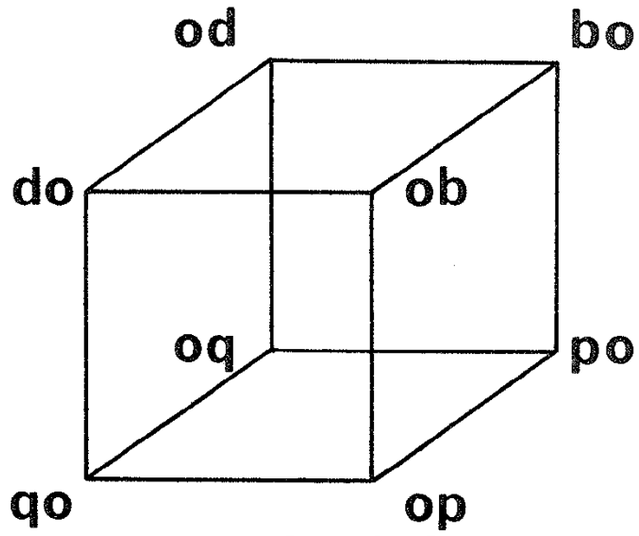


Fig. 2

Leroy Johnson, *ACM SIGPLAN Notices*, 15(12), 1980

Bracketing Programme Constructs

J. G. Hunt

FRTA-4233, LGZ Landis Gyr Zug AG, CH-6301 Zug.

Amused by the levity of {3} and annoyed by the attitude of {2}, I should like to make a serious contribution to the do...od controversy.

In the realm of expressions of the traditional arithmetic kind, the use of brackets and parentheses to clarify and disambiguate generally goes unchallenged. Certainly for expressions of limited complexity the method usually works for the human reader: whilst for a machine free of problems of the psychology of perception, greater depths of nesting present no theoretical difficulty.

As Lisp and Algol-68 have shewn, the same method may be applied to bracketing other programme constructs: yet the majority of human readers tend to find this lacking in clarity. Lisp provides no alternative, and has accordingly been much abused and reviled. Algol-68 permits the use of symmetric keyword pairs, such as comment...tnemmoc: probably the one example of a symmetry which is not beautiful.

J. G. Hunt, *ACM SIGPLAN Notices*, 16(4), 1981

Bracketing Programme Constructs

J. G. Hunt

FRTA-4233, LGZ Landis Gyr Zug AG, CH-6301 Zug.

Amused by the levity of {3} and annoyed by the attitude of {2}, I should like to make a serious contribution to the do...od controversy.

In the realm of expressions of the traditional arithmetic kind, the use of brackets and parentheses to clarify and disambiguate generally goes unchallenged. Certainly for expressions of limited complexity the method usually works for the human reader: whilst for a machine free of problems of the psychology of perception, greater depths of nesting present no theoretical difficulty.

As Lisp and Algol-68 have shewn, the same method may be applied to bracketing other programme constructs: yet the majority of human readers tend to find this lacking in clarity. Lisp provides no alternative, and has accordingly been much abused and reviled. Algol-68 permits the use of symmetric keyword pairs, such as comment...tnemmoc: probably the one example of a symmetry which is not beautiful.

J. G. Hunt, *ACM SIGPLAN Notices*, 16(4), 1981

Bracketing Programme Constructs

J. G. Hunt

FRTA-4233, LGZ Landis Gyr Zug AG, CH-6301 Zug.

Amused by the levity of {3} and annoyed by the attitude of {2}, I should like to make a serious contribution to the do...od controversy.

In the realm of expressions of the traditional arithmetic kind, the use of brackets and parentheses to clarify and disambiguate generally goes unchallenged. Certainly for expressions of limited complexity the method usually works for the human reader: whilst for a machine free of problems of the psychology of perception, greater depths of nesting present no theoretical difficulty.

As Lisp and Algol-68 have shewn, the same method may be applied to bracketing other programme constructs: yet the majority of human readers tend to find this lacking in clarity. Lisp provides no alternative, and has accordingly been much abused and reviled. Algol-68 permits the use of symmetric keyword pairs, such as comment ... tnemmoc, probably the one example of a symmetry which is not beautiful.

J. G. Hunt, *ACM SIGPLAN Notices*, 16(4), 1981

Today

- What is good language *design*.
- NOT: What is a good language?
- Can we explain *good language design*?
- Approach this question from the other side
  - **language design smells**

Code Smell

A *code smell* is a hint that something has gone wrong somewhere in your code.

http://c2.com/cgi/wiki?CodeSmell

Code Smell

A *code smell* is a hint that something has gone wrong somewhere in your code.

Symptom of deeper issues (maybe)

http://c2.com/cgi/wiki?CodeSmell

Code Smell

A *code smell* is a hint that something has gone wrong somewhere in your code.

Symptom of deeper issues (maybe)

Does not affect user of software, but programmer

http://c2.com/cgi/wiki?CodeSmell

Smell	Example
"Duplicated code"	

```
public boolean checkGameOver(Graphics g2) {
  g2.setColor(Color.RED);

  if (this.player1.getHealth() < 0) {
    g2.drawString("Game Over, winner: " +
           this.player1.getName(), SCREEN_PADDING,
                  PLAYER_HEIGHT + 250);
    return true;
  }

  if (this.player2.getHealth() < 0) {
    g2.drawString("Game Over, winner: " +
           this.player2.getName(), SCREEN_PADDING,
                  PLAYER_HEIGHT + 250);
    return true;
  }
  return false;
}
```

Language Design Smell

A *language design smell* is a hint that something has gone wrong somewhere in your language design.

Language Design Smell

A *language design smell* is a hint that something has gone wrong somewhere in your language design.

Does affect user (= programmer)

Language Design Smell

A *language design smell* is a hint that something has gone wrong somewhere in your language design.

Does affect user (= programmer)

[Parallel to user interface design smells]

Visual Basic 6

Option Base Statement Example

This example uses the **Option Base** statement to override the default base array subscript value of 0. The **LBound** function returns the smallest available subscript for the indicated dimension of an array. The **Option Base** statement is used at the module level only.

```
Option base 1    ' Set default array subscripts to 1.

Dim Lower
Dim MyArray(20), TwoDArray(3, 4)    ' Declare array variables.
Dim ZeroArray(0 To 5)    ' Override default base subscript.
' Use LBound function to test lower bounds of arrays.
Lower = LBound(MyArray)    ' Returns 1.
Lower = LBound(TwoDArray, 2)    ' Returns 1.
Lower = LBound(ZeroArray)    ' Returns 0.
```

http://msdn.microsoft.com/en-us/library/aa266180(v=vs.60).aspx

Cf. *VBA Language Specification*, Microsoft Corporation, Release: March 15, 2010, p. 56-57

JavaScript block scoping

```
function f() {
  var i = 13;
  {
     var i = 42;
     print(i);
  }
  print(i);
}
```

https://github.com/spencertipping/js-in-ten-minutes

http://matt.might.net/articles/javascript-warts/

http://oreilly.com/javascript/excerpts/javascript-good-parts/awful-parts.html

JavaScript block scoping

```
function f() {
  var i = 13;
  {
      var i = 42;
      print(i);
  }
  print(i);
}
```

prints: 42 42

https://github.com/spencertipping/js-in-ten-minutes

http://matt.might.net/articles/javascript-warts/

http://oreilly.com/javascript/excerpts/javascript-good-parts/awful-parts.html

Smell	Example	Manifestation

"Counter intelligence"	JavaScript blocks appear to introduce new scopes, but they don't.	`function f() {`
		`  var i = 13;`
		`  {`
		`  var i = 42;`
		`  print(i);`
		`  }`
		`  print(i);`
		`}`
		`// prints 42 42`

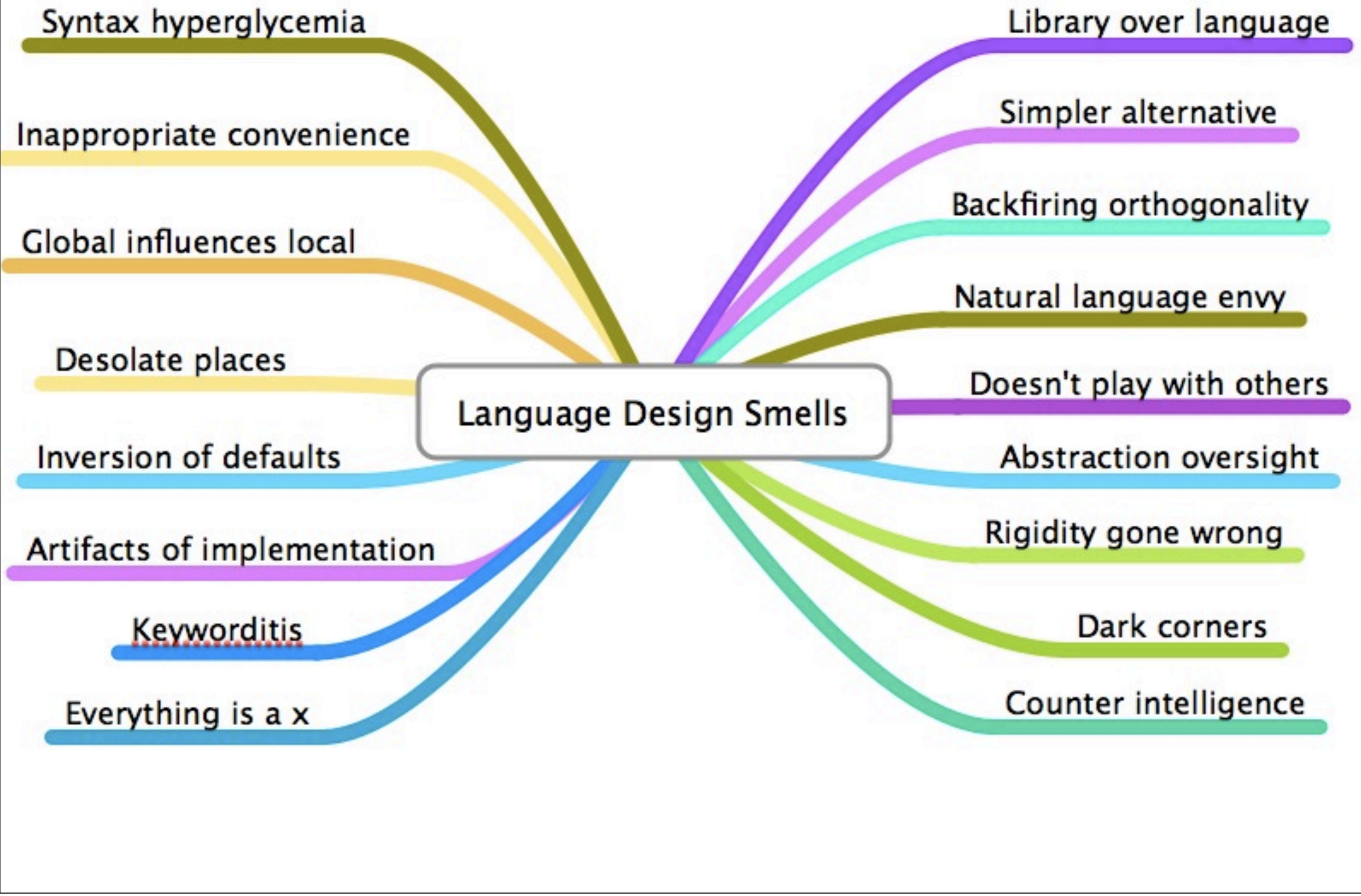
Language design smell = ...?

- Domain independent
- Language independent
- Fixable
- Unnecessary
- Debatable (trade-offs)
- Net negative
- Accidental
- Wicked
- Arbitrary
- Circumstantial
- Unintended

Smell vs feature

Smell → Feature

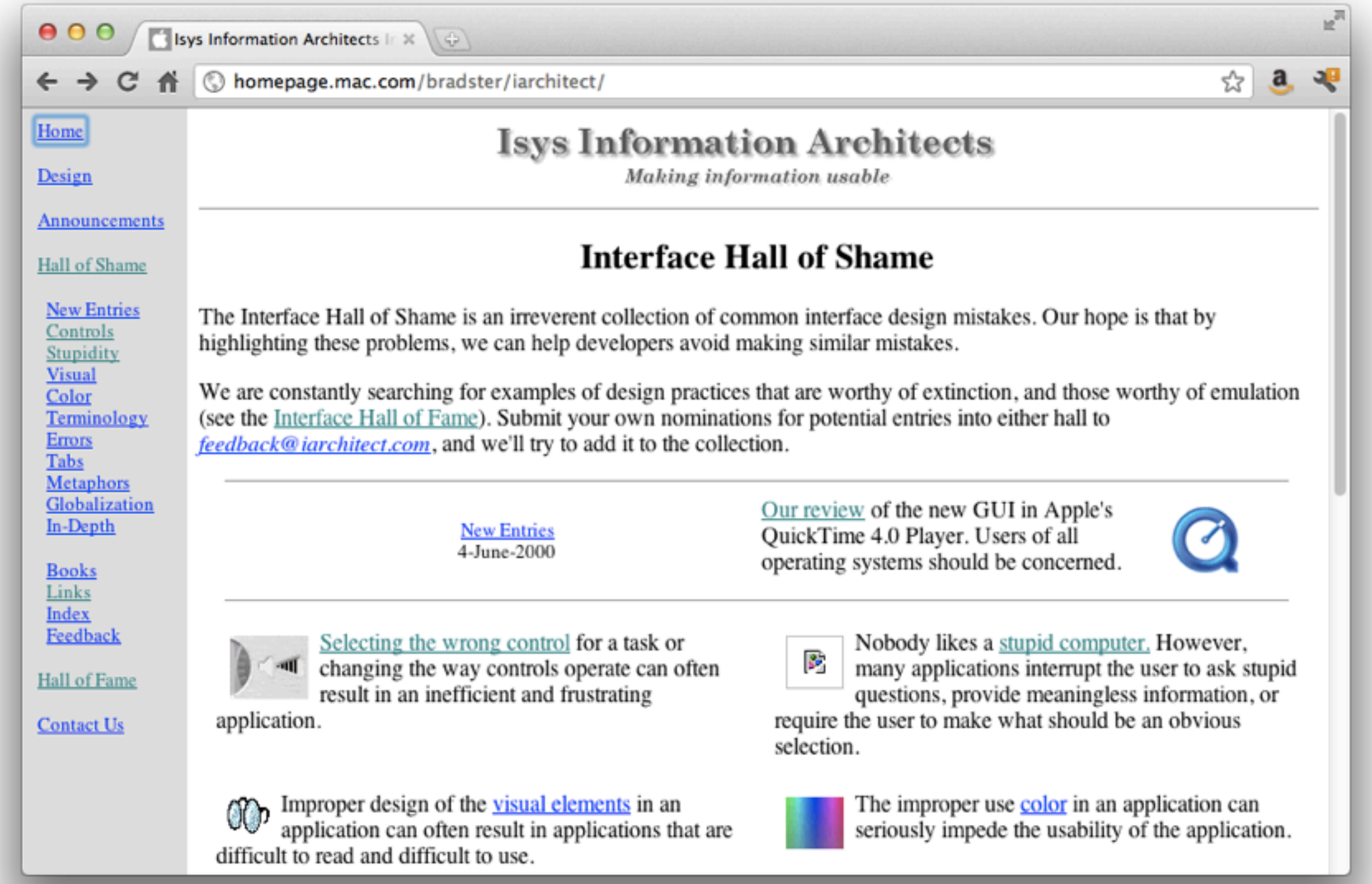
Smell	Feature
contingent	motivated
unnecessary	required
irrational	rational
accidental	essential
ad hoc	structural
arbitrary	intentional
fringe	central
consequence	decision
negative	positive


Language Design Smells
Syntax hyperglycemia
Inappropriate convenience
Global influences local
Desolate places
Inversion of defaults
Artifacts of implementation
Keyworditis
Everything is a x
Library over language
Simpler alternative
Backfiring orthogonality
Natural language envy
Doesn't play with others
Abstraction oversight
Rigidity gone wrong
Dark corners
Counter intelligence

Research agenda

- Literature survey (annotated bibliography?)
- Catalogue of smells (Wiki? Taxonomy?)
- Language design (anti-)patterns?
- Language critiques?
- Language scholarship?
- http://www.languagedesign.org

PL Hall of Shame



Further questions

- How to organize language design smells?
- Use of software quality attributes (-ilities)?
- Difference smell, bug, flaw, bad idea, ...?
- Are DSLs special?

What's your favorite language wart?

storm@cwi.nl