Grace

An open-source educational programming language

Michael Homer

Why?

Principles

- Simple programs should be simple
- Understandable semantic model
- Support different teaching orders
- Be a general-purpose language

Simple programs should be simple

Incantations

```
package user;

class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

Simple programs should be simple
No Incantations

```
print "Hello world"
```

Understandable semantic model

Method requests

people.add(person)

print "Hello, world!"

```
// Implicit receiver
```

$((x + y) > z)$ && !q

```
// Operators are methods
```

obj.x := 2

```
// Accessor methods
```

5.between(3)and(8)

```
// Multi-part method name
```

Understandable semantic model

Control structures

```
if (x < 0) then {
  print "x is negative"
} else {
  print "x is non-negative"
}


while {x > 0} do {
  x := x − 1
}
```

Support different teaching orders

Objects and classes

```
object {
  def x is public = 5
  var y is public := 7
  method distanceTo(other) { ... }
}
class point.x(x') y(y') {
  def x is public = x'
  var y is public := y'
  method distanceTo(other) { ... }
}
```

Support different teaching orders

Classes are factories

```
class point.x(x’) y(y’) {
  def x is public = x’
  var y is public := y’
  method distanceTo(other) { ... }  }
```

means exactly

```
def point = object {
  method x(x’)y(y’) {
    object {
      def x is public = x’
      var y is public := y’
      method distanceTo(other) { ... }
    }  }  }
```

Support different teaching orders

Types

```
method sum(a : Number, b : Number) ->
  Number {
  return a + b
}
var score : Number := sum(5, 10)
```

Support different teaching orders

Types are optional

```
method sum(a : Number, b : Number) ->
  Number {
   return a + b
}
var score : Number := sum(5, 10)
```

```
method sum(a, b) {
   return a + b
}
var score := sum(5, 10)
```

Tough choices

- Visibility: supporting simpler programming or correct engineering?
- Inheritance: it’s hard.
- Uniformity or variation?

Dialects

```
dialect "beginner"
...
```

- A single line to pick one
- On a *per-module* basis

Embracing variation

Engineering

Standard Grace

. . .

. . .

Static

Dynamic

. . .

Classes

Objects

Procedural

Engineering first

Functional objects

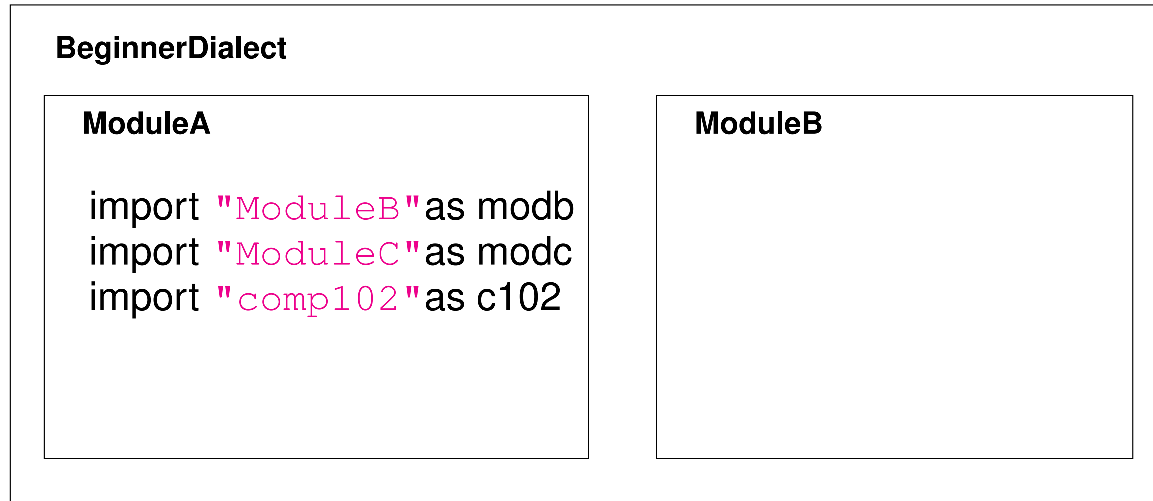
Functional

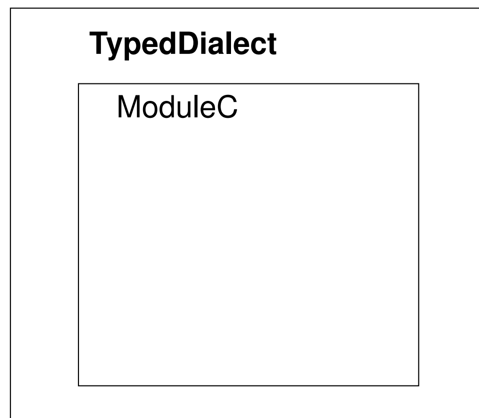
. . .

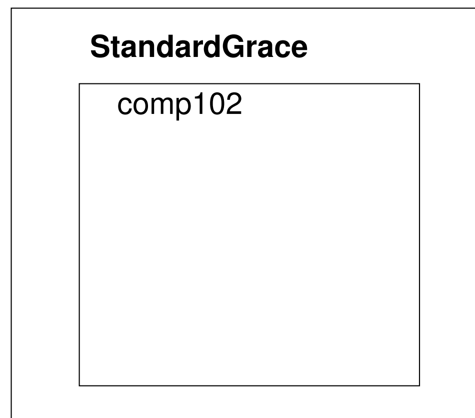
Static Functional

Dynamic Functional

Nesting







My favourite Java error

```
 1 class Counter {
 2     int total = 0;
 3     int add(int n) {
 4         return (total += n);
 5     }
 6     int addAllNegative(Iterable<Integer> all) {
 7         for (int n : all)
 8             if (n < 0)
 9                 int tot = add(-n);
10         return total;
11     }
12 }
```

My favourite Java error

```
 1 class Counter {
 2     int total = 0;
 3     int add(int n) {
 4         return (total += n);
 5     }
 6     int addAllNegative(Iterable<Integer> all) {
 7         for (int n : all)
 8             if (n < 0)
 9                 int tot = add(−n);
10         return total;
11     }
12 }
```

```
Counter.java:9: error: '.class' expected
                int tot = add(-n);
                    ^
```

Pattern matching

```
match(x) // x : 0 | String | Student

  // Match against a literal
  case { 0 -> print "Zero" }

  // Typematch, binding a variable
  case { s : String -> print(s) }

  // Destructuring match
  case { _ : Student(name, id)-> print(name)}
```

Pattern matching

```
match(x) // x : 0 | String | Student

  // Match against a literal
  case { 0 -> print "Zero" }

  // Typematch, binding a variable
  case { s : String -> print(s) }

  // Destructuring match
  case { _ : Student(name, id)-> print(name)}
```

Pattern matching

```
match(x)

  // Nested patterns
  case { p : Point(0,y) -> print "(0,{y})" }

  // Pattern operators
  case { p : Point(0, _) | Point3D(0, _, _)
      -> print(p) }

  case { s : Seq & Dog
      -> s.bark(s.size)}
```

Extensible patterns

```
if (Point.match(x)) then {
    ...
}
```

```
method match(o : Any)
        -> SuccessfulMatch | FailedMatch { ... }
```

Implementation

Minigrace

- Written in Grace
- Supports everything here, targets C and JavaScript

Compiler source code (in Grace): `github/mwh/minigrace`

Tarballs (pregenerated C code): `ecs.vuw.ac.nz/~mwh/minigrace/`

Client-side web front-end: `ecs.vuw.ac.nz/~mwh/minigrace/js`

Hopper

- Written in concurrent JavaScript: `github/zmthy/hopper`
- Had its own talk on Wednesday

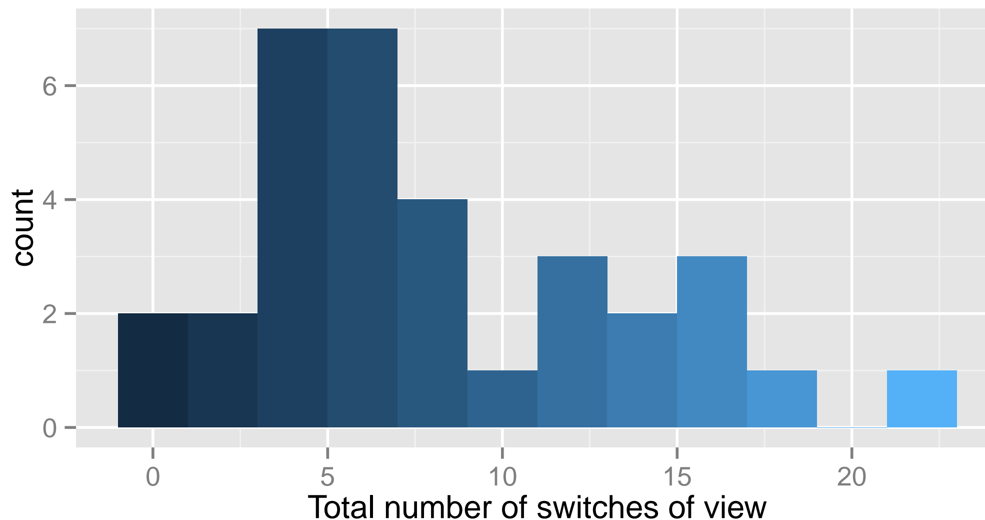
All links, and more, available from

mwh.nz/LCA2015

Live demo

Tiled Grace experiment

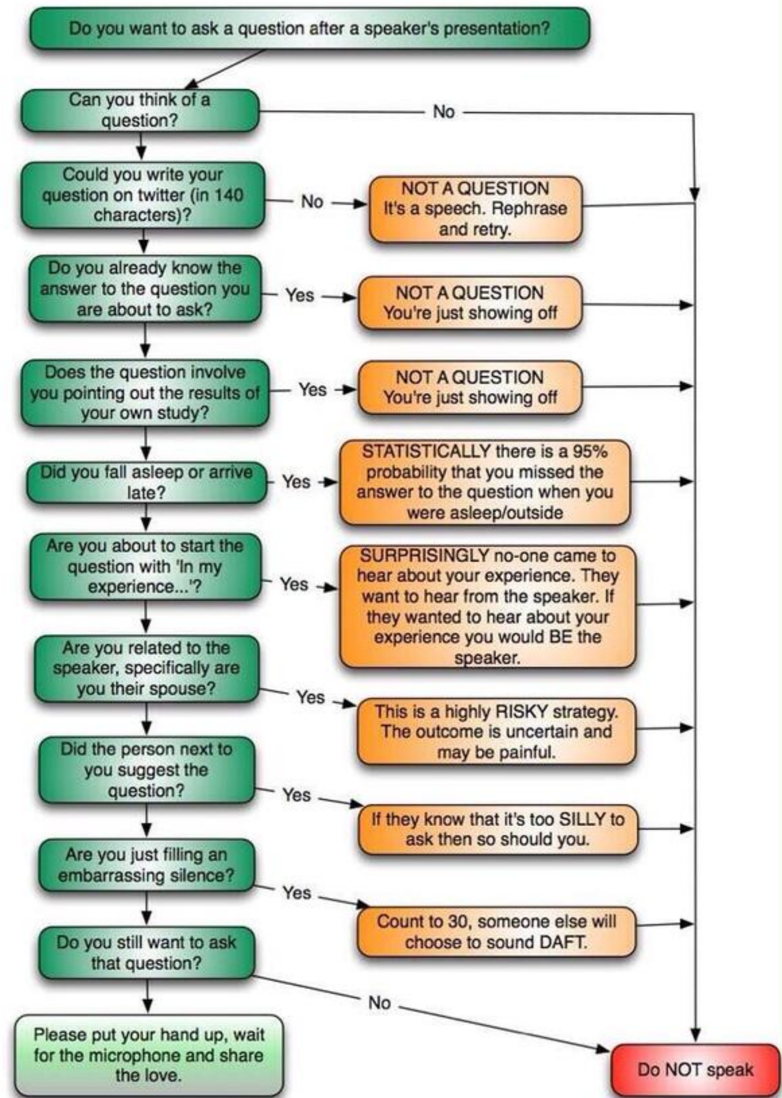
- 33 participants, mostly students
- 5 tasks, fully instrumented



Grace

An open-source educational programming language

Michael Homer


Do you want to ask a question after a speaker's presentation?
Can you think of a question?
No
Could you write your question on twitter (in 140 characters)?
No
NOT A QUESTION It's a speech. Rephrase and retry.
Do you already know the answer to the question you are about to ask?
Yes
NOT A QUESTION You're just showing off
Does the question involve you pointing out the results of your own study?
Yes
NOT A QUESTION You're just showing off
Did you fall asleep or arrive late?
Yes
STATISTICALLY there is a 95% probability that you missed the answer to the question when you were asleep/outside
Are you about to start the question with 'In my experience...'?
Yes
SURPRISINGLY no-one came to hear about your experience. They want to hear from the speaker. If they wanted to hear about your experience you would BE the speaker.
Are you related to the speaker, specifically are you their spouse?
Yes
This is a highly RISKY strategy. The outcome is uncertain and may be painful.
Did the person next to you suggest the question?
Yes
If they know that it's too SILLY to ask then so should you.
Are you just filling an embarrassing silence?
Yes
Count to 30, someone else will choose to sound DAFT.
Do you still want to ask that question?
No
Please put your hand up, wait for the microphone and share the love.
Do NOT speak

Additional slides

Extra details that may be helpful

Loop invariants

Module "**loopinvariant**":

```
method for(it : Iterable) invariant(inv : Block<Boolean>)do(blk :
  Block) {
    for (it) do {i->
        if (!inv.apply) then {
            InvariantFailure.raise "Loop invariant not
              satisfied."
        }
        blk.apply(i)
    }
    if (!inv.apply) then {
        InvariantFailure.raise "Loop invariant not
          satisfied."
    }
}
```

Loop invariants client code

```
dialect "loopinvariant"

var sum : Number := 0

for (1..10) invariant { sum > 0 } do { item :
  Number ->
    sum := sum + item
}
```

http://ecs.vuw.ac.nz/~mwh/minigrace/js/#sample=loopinvariant_example

Pluggable checkers

```
import "StandardPrelude" as StandardPrelude
inherits StandardPrelude.new
def CheckerFailure = Exception.refine "CheckerFailure"

method checker(nodes) {
    for (nodes) do {n->
        if (n.kind == "vardec") then {
            CheckerFailure.raiseWith("var declarations
              are not allowed at the top level", n.
              name)
        }
    }
}
```

Dialect-support dialect

```
dialect "dialect"
import "StandardPrelude" as StandardPrelude
inherits StandardPrelude.new

fail "var declarations not allowed"
    when { v : VarDec -> true }

method checker(l) {
    check(l)
}
```

Similar: http://ecs.vuw.ac.nz/~mwh/minigrace/js/#sample=dialect_example

DSLs: Object associations

```
dialect "object-associations"
def Attends = Relationship<Student, Course>
def Teaches = Relationship<Course, Faculty>
def Prerequisites = ReflexiveRelationship<Course>
// Set up or obtain our data objects
def james = student (...)
...
Attends.add(james, cs102)
...
for (Attends.to(cs102)) do { each -> ... }
```

http://ecs.vuw.ac.nz/~mwh/minigrace/js/#sample=ObjectAssociations_example

DSLs: Finite State Machines

```
dialect "fsm"
def startState = state { print "Starting" }
def runState = state { print "Running" }
def endState = state { print "Done" }
in ( startState ) on("A") goto(runState)
in (runState)
    on("A") goto(runState)
    on("B") goto(endState)

method process(symbol : String) {
    transition (symbol)
}
```

http://ecs.vuw.ac.nz/~mwh/minigrace/js/#sample=fsm_example

The extreme: GrAPL

```
dialect "grapl"
N ← [1, 2, 3, 4]
print (N)
print (N + 2)
print (+/N)
// Standard Lotto example
print (L[⍋(L ← (n 6 ? 40))])
// Calculate primes up to 20 - note that the /
// function has its parameters reversed here,
// because of Grace's evaluation order.
print ((P ← (n 1 ↓ ι 20))/ ∼(P∈(P∘.∗P)))
```

http://ecs.vuw.ac.nz/~mwh/minigrace/js/#sample=grapl_example

What is pattern-matching?

Take an object.

Do “something” if it’s an object the pattern matches.

Otherwise, try the next pattern or error.

What does pattern-matching mean?

Take an object.

Do “something” if it’s an object the pattern matches.

Otherwise, try the next pattern or error.

Pattern-matching is applying a partial function.

$$\begin{aligned} f(x) &= -x \quad \text{when } x < 0 \\ f(x) &= x \quad\;\;\; \text{otherwise} \end{aligned}$$

Match results

```
if (Point.match(x)) then {
   ...
}

def matchResult = Point.match(x)

def values : Tuple<Number, Number> =
  matchResult.bindings

def p : Point = matchResult.result
```

Exceptions

- Want a hierarchy of errors…
- …but they all have the same type.
- Pattern-matching!

Exceptions as patterns

```
def MyError = Error.refine "MyError"
def NegativeError = MyError.refine "
  NegativeError"

try {
   if (value < 0) then {
     NegativeError.raise "{value} < 0"
   }
} catch {e : MyError -> print "Error: {e}"
  }
```

Blocks

Are objects:

```
def welcome = { n-> print "Hello {n}" }
```

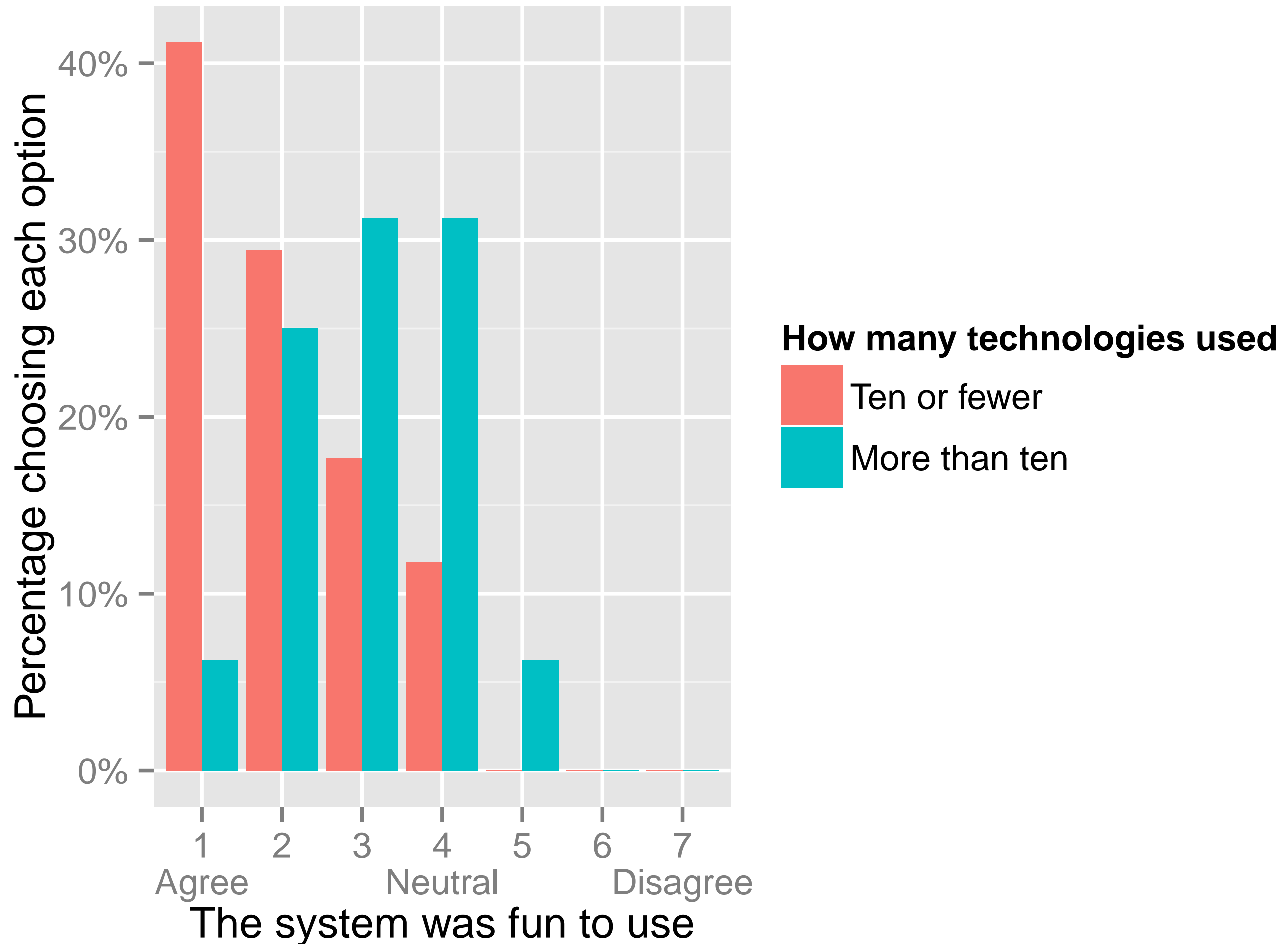
```
welcome.apply "World"
```

```
object { // Almost:
  method apply(n) {
    print "Hello {n}"
  } // In fact, self
} // is unchanged
```

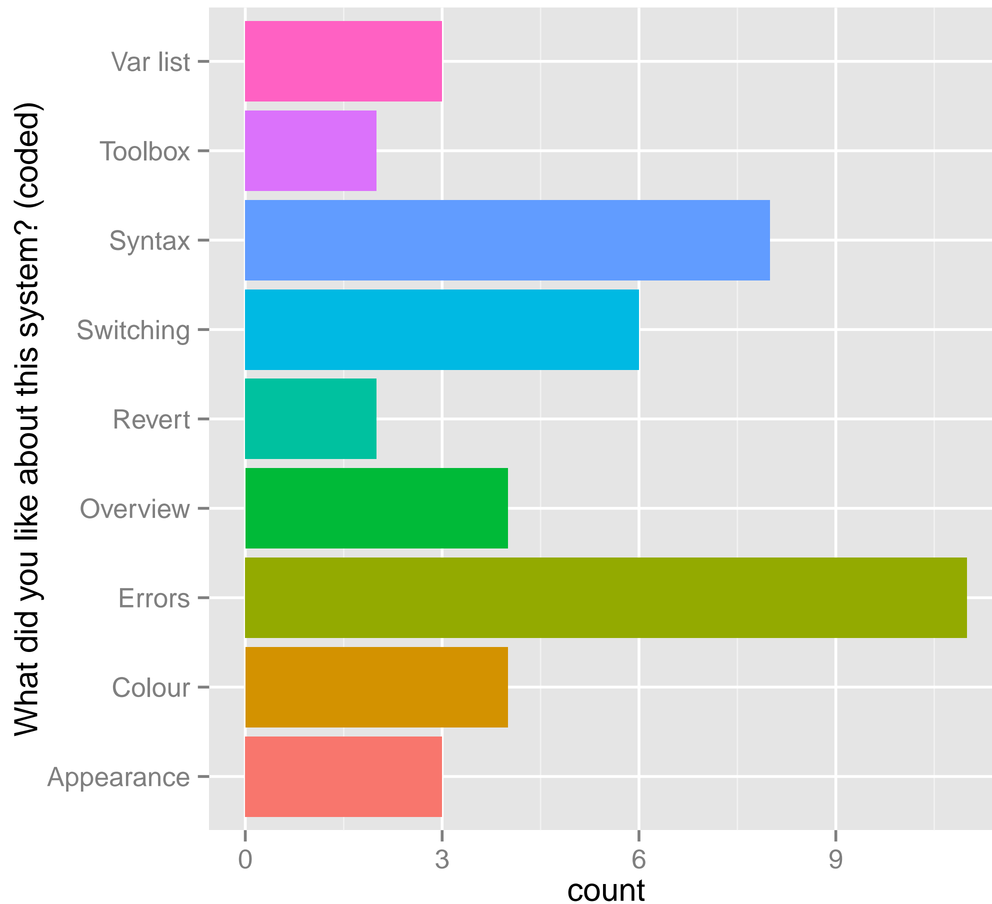
Experiment

- Anonymised data set available
- Includes instrumentation and analysis tooling
- Complete (52-page) writeup of protocol and results also available
- All of this accessible from mwh.nz/LCA2015

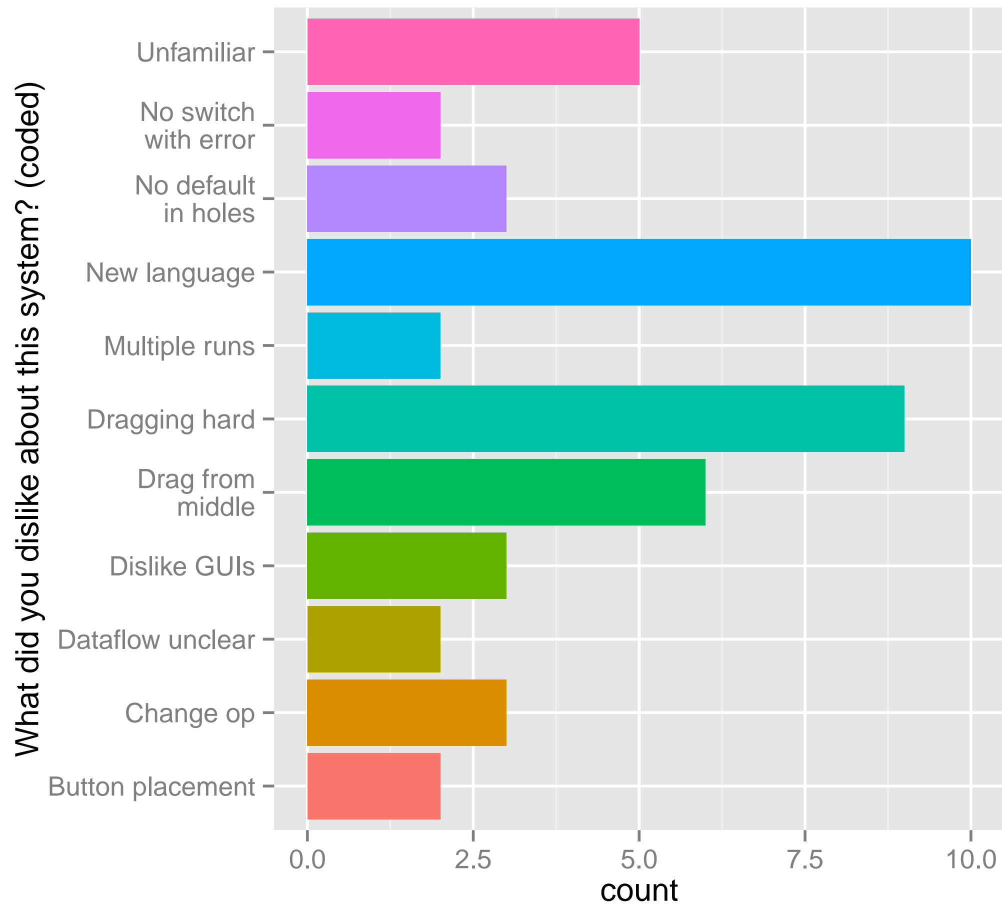
Fun by experience



Freeform likes



Freeform dislikes



Type operations

- Variants: Point | Nil

$$x : A \mid B \equiv x : A \vee x : B$$

```
def nilValue : Nil = ...
var p : Point | Nil := nilValue // OK
...
p := CartesianPoint.new(3,4) // OK
```

- Intersubsection: T1 & T2 conforms to T1 and T2

Gradual types and inheritance

```
class x.new {
  method a {
    self.b
  }
}
class y.new {
  inherits x.new
  method b { print "B" }
}
y.new.a
```

Contents

Title slide 1

Why now? 2

Principles 3

Simple programs should be simple 4

Incantations 4

Incantations 5

Understandable semantic model 6

Method requests 6

Control structures 7

Support different teaching orders 8

Objects and classes 8

Classes are factories 9

Types 10

Types are optional 11

Tough choices 12

Dialects 13

Embracing variation 14

Nesting 15

My favourite Java error 16

The message 17

Pattern matching 18

A digression 19

Pattern matching redux 20

Extensible patterns 21

Implementation 22

Live demo 23

Tiled Grace experiment 24

Ending slide 25

Reminder 26

Additional slides 27

Dialect samples 28

Loop invariants 28

Loop invariants client code 29

Pluggable checkers 30

Dialect-support dialect 31

DSLs: Object associations 32

DSLs: Finite State Machines 33

The extreme: GrAPL 34

What is pattern-matching? 35
    Partial functions 36
Match results 37
Exceptions 38
    Exceptions as patterns 39
Blocks 40
Experiment 41
    Fun by experience 42
    Freeform likes 43
    Freeform dislikes 44
Type operations 45
Gradual types and inheritance 46

Contents