THE UNIVERSITY OF HONG KONG
DEPARTMENT OF COMPUTER SCIENCE

WHEN IMPORTLESS[1] BECOMES MEANINGFUL: A CALL FOR TAG-BASED NAMESPACES IN PROGRAMMING LANGUAGES

TOMAS TAUBER
October 20, 2014

1. Void of meaning. [Obs.] Shak. (Webster's Dictionary 1913 Edition)

WHY IS MY ACCENT SO STRANGE?

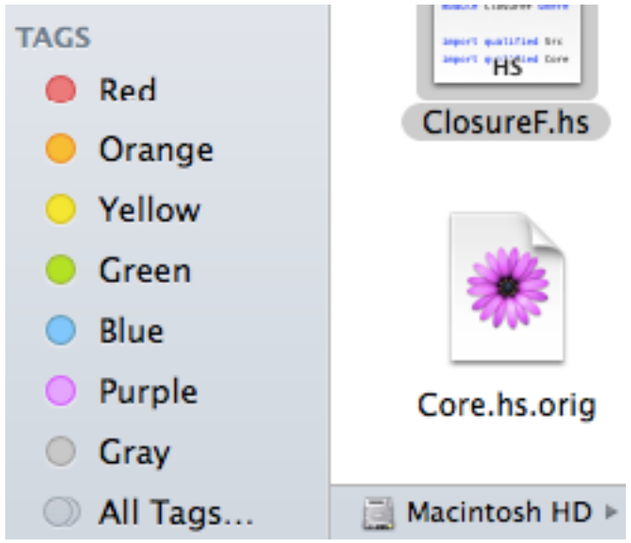
Slavic + Scottish + Cantonese accents combined

OUTLINE

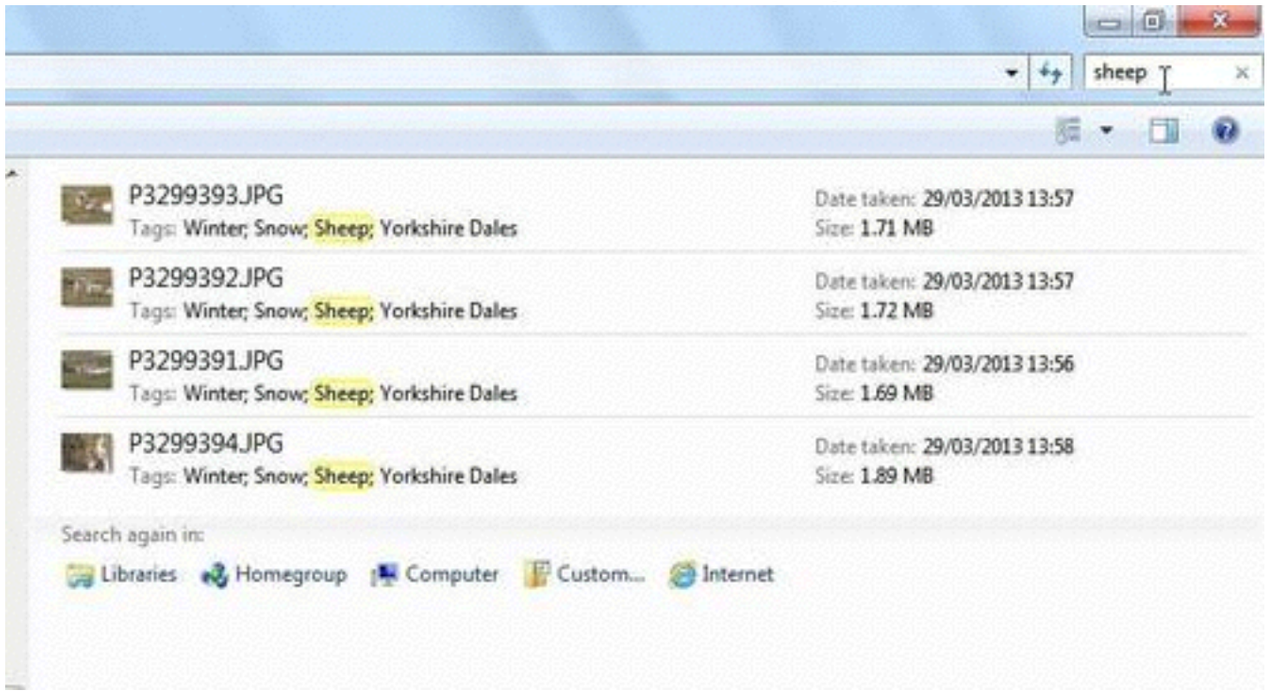
- Current filesystem trends in applications and OS
- Status quo in programming languages
- New namespace organization
- Related work
- Future plans and discussion

CURRENT TRENDS

MacOS Finder



Windows File Explorer



How are your emails stored?

Is this presentation a file or a directory?

PROGRAMMING LANGUAGES NAMESPACES

"Solution" 1:

Prefixes

```
myNS_foo();
yourNS_foo();
```

nesting in identifiers

querying?

all names fully qualified

Solution 2:

Emulate (in) Hierarchical Filesystem

```
src/
    com/
        google/
            android/
                gms/
                    common/
                        api/
                            ...
```

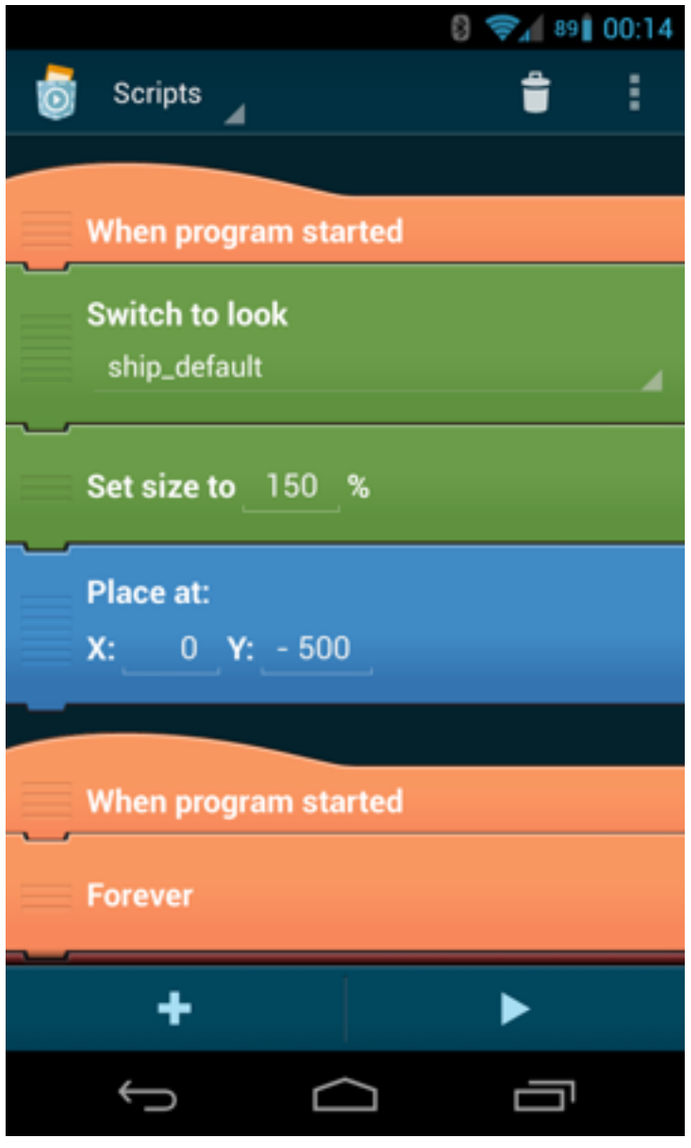
Better,

but is it

always suitable?

OTHER OBSERVATIONS I

Pocket Code
(demo at SPLASH'14)



Should we care how
these programs are stored
in the filesystem?

OTHER OBSERVATIONS II

Typical Utility module:

util.py (original)

```
import sys
import inspect
import heapq, random

"""
Data structures useful for implementing SearchAgents

class Stack:
  "A container with a last-in-first-out (LIFO) queuing policy."
  def __init__(self):
    self.list = []

  def push(self,item):
    "Push 'item' onto the stack"
    self.list.append(item)

  def pop(self):
    "Pop the most recently pushed item from the stack"
    return self.list.pop()

  def isEmpty(self):
    "Returns true if the stack is empty"
    return len(self.list) == 0

class Queue:
  "A container with a first-in-first-out (FIFO) queuing policy."
  def __init__(self):
    self.list = []

  def push(self,item):
    "Enqueue the 'item' into the queue"
    self.list.insert(0,item)

  def pop(self):
    """
      Dequeue the earliest enqueued item still in the queue. This
      operation removes the item from the queue.
    """
    return self.list.pop()

  def isEmpty(self):
    "Returns true if the queue is empty"
    return len(self.list) == 0

class PriorityQueue:
  """
    Implements a priority queue data structure. Each inserted item
    has a priority associated with it and the client is usually interested
    in quick retrieval of the lowest-priority item in the queue. This
    data structure allows O(1) access to the lowest-priority item.

    Note that this PriorityQueue does not allow you to change the priority
    of an item.  However, you may insert the same item multiple times with
    different priorities.
  """
  def  __init__(self):
    self.heap = []

  def push(self, item, priority):
      pair = (priority,item)
      heapq.heappush(self.heap,pair)

  def pop(self):
      (priority,item) = heapq.heappop(self.heap)
      return item

  def isEmpty(self):
    return len(self.heap) == 0

class PriorityQueueWithFunction(PriorityQueue):
  """
  Implements a priority queue with the same push/pop signature of the
  Queue and the Stack classes. This is designed for drop-in replacement for
  those two classes. The caller has to provide a priority function, which
  extracts each item's priority.
  """
  def __init__(self, priorityFunction):
    "priorityFunction (item) -> priority"
    self.priorityFunction = priorityFunction      # store the priority function
    PriorityQueue.__init__(self)      # super-class initializer

  def push(self, item):
    "Adds an item to the queue with priority from the priority function"
    PriorityQueue.push(self, item, self.priorityFunction(item))


def manhattanDistance( xy1, xy2 ):
  "Returns the Manhattan distance between points xy1 and xy2"
  return abs( xy1[0] - xy2[0] ) + abs( xy1[1] - xy2[1] )

"""
  Data structures and functions useful for various course projects

  The search project should not need anything below this line.
"""

class Counter(dict):
  """
  A counter keeps track of counts for a set of keys.

  The counter class is an extension of the standard python
  dictionary type.  It is specialized to have number values
  (integers or floats), and includes a handful of additional
  functions to ease the task of counting data.  In particular,
  all keys are defaulted to have value 0.  Using a dictionary:
```

different pieces of functionality

Should they be in separate files?

Why do we care?

Comments

with **useful** information

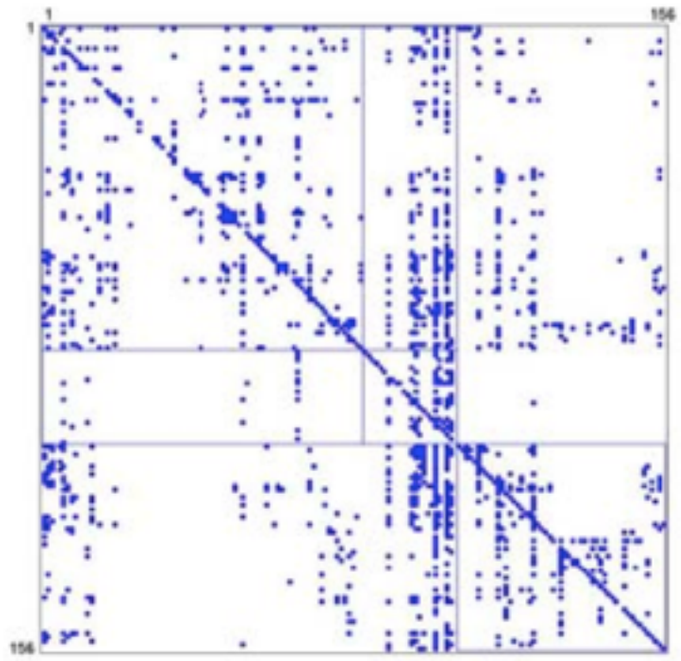
and **keywords**

completely ignored

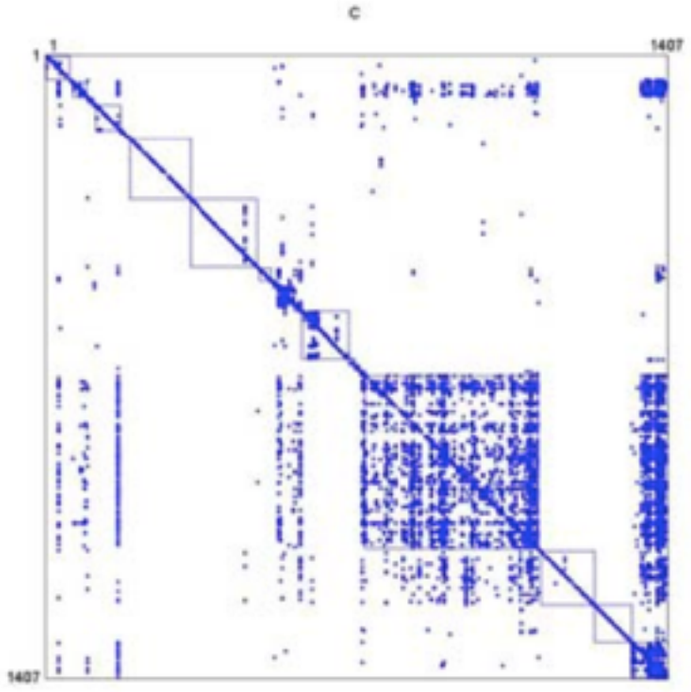
by the compiler / interpreter

OTHER OBSERVATIONS III

Version A



Version C



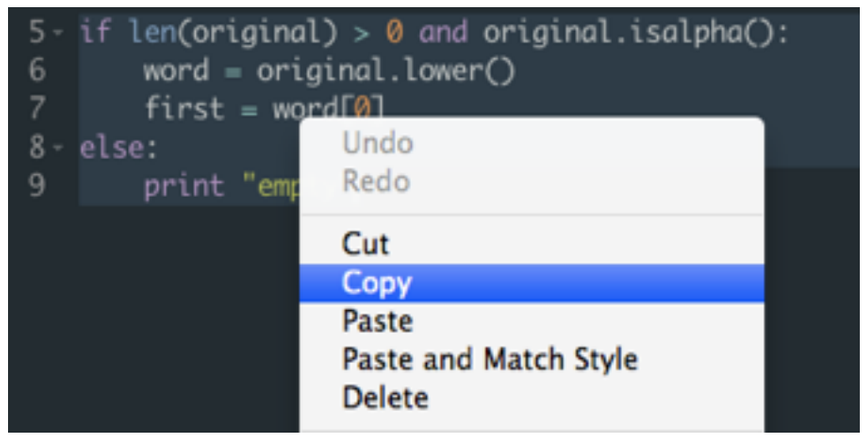
Software evolves,
interactions in it **change**,
original **directory structure**
makes less sense…

Should we periodically waste time
on directory structure
refactoring?

Design Structure Matrices from
"*The Impact of Component Modularity on Design Evolution*" by MacCormack, Rusnak, and Baldwin

MISSING FEATURES

- Seamless copy-pasting of code pieces to REPL or between edited files?



- Easy source code queries?

  E.g. "Give me all John Doe's functions that transform my data structure"

- Multiple code versions in environment?

PYTHON EXAMPLE

May not be able to copy-paste or move definitions without imports

```
from weather import get_data
from numpy import sum
  (…)
# this calculates average
temperature from recent
weather forecasts

def avg_temp():
  data = get_data()
  avg = sum(data) / len(data)
  return avg
  (...)
```

Cannot have multiple versions

Contains useful info, completely ignored by the interpreter

REMOVING IMPORTS

- What if we remove all imports and auto-import any identifies?

- We may then move around and copy-paste code seamlessly

- What about conflicts?

IMPORTLESS PYTHON

```
# this calculates #average
#temperature from recent
#weather forecasts
# #@#('.../site-packages/
numpy/', #ndarray)
def avg_temp():
  data = get_data()
# #summation #ndarray
#version:1.8.1
  avg = sum(data) / len(data)
  return avg
```

Marking keywords for guiding auto-import

Local tag addition

Gets auto-imported

Needs extra information due to conflicts

TAGGING

- **Per global scope definitions** (functions, variables, classes): guides auto-import in them + "exposes" them

- **Per local scope block**: guides auto-import for all statements in them

- **Per statement**: guides auto-import for a particular statement

INTERPRETER

- *Pre-processing files*: inserting explicit local imports
- Lookup of identifiers using the *standard Boolean model*
- If two or identifiers match the lookup with the same score: **error** (need to **disambiguate using more tags**)
- For dynamic code or REPL, exception hook is added to intercept *NameErrors* and try lookups

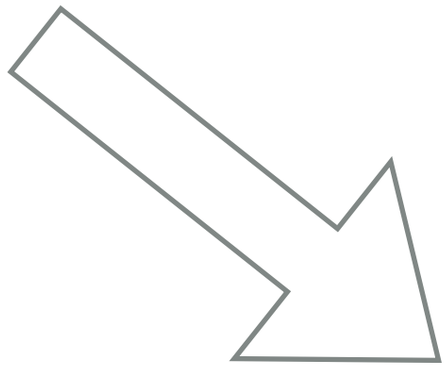
STORING

```
# this calculates #average #temperature from recent #weather forecasts

# #@#('.../site-packages/numpy/', #ndarray)

def avg_temp():
  data = get_data()
# #summation #ndarray #version:1.8.1

  avg = sum(data) / len(data)
  return avg
```



I	I	average
2	I	temperature
3	I	weather
. . .		

I	avg_temp	FUN	phEqZpwoQM	/usr/local/lib/. . .	# this calculates #average #temperature from recent #weather forecasts # #@#('.../site-packages/numpy/', #ndarray) def phEqZpwoQM(): data = get_data() # #summation #ndarray #version:1.8.1
..					

EDITING

Workspace

file1

all units
matching
#weather
or
#model

conf

file1:
#weather
#model
file2:
#gui
#layout

file2

all units
matching
#gui
or
#layout

Based on the specified tags, we generate temporary files and synchronize them with corresponding entries in the database

EVALUATION

- Under this more flexible setting, we do not run into the current namespace issues (e.g. more versions can coexist)

- We can convert from a hierarchical namespace, but would there be any issues?
  - Much worse performance?
  - Worse cohesion of modules?

- To assess this, we convert a collection of packages (Anaconda 1.9.2) to be tag-based, compare the module-level LCOM4 metric and execution time of running their unit tests

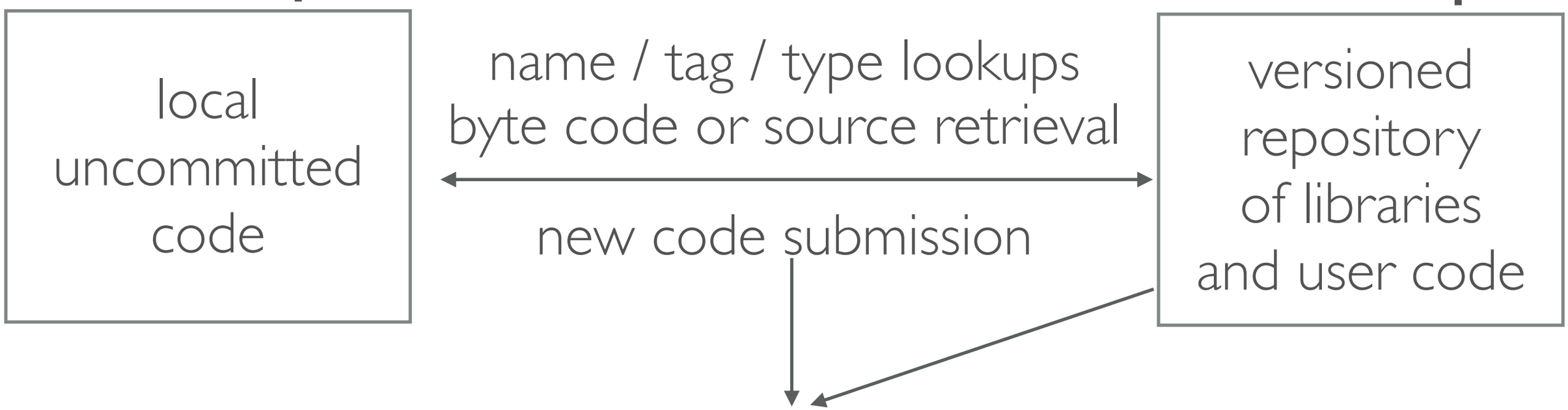
RELATED WORK

- OS FS Research: "*Hierarchical File Systems are Dead*" (Seltzer, Murphy), TagFS, WinFS, …
- "*Call by Meaning*" (Samimi et al.) @ Onward! 2014
- "*Keyword Programming in Java*" (Little and Miller)
- Attribute-oriented and intentional programming
- UpgradeJ (Bierman, Parkinson, and Noble)

FUTURE WORK

Extending to statically typed languages

Architectural changes



1. Does it break tests? If yes, reject.
2. Does it change interactions? If yes, show a warning / require confirmation.

FUTURE WORK

Extending to statically typed languages

Formalization

1. Semantics of the core meta-language (basic functionality, e.g. lookups)
2. Semantics of a more advanced (e.g. different tag co-existence restrictions) meta-language and its translation to the core language

Other possible work:
separate / global compilation,
type inference in this setting,
OOP semantics with tags…

SUMMARY

- Most programming language namespaces more or less mimic hierarchical filesystems

- That is often good, but fails in a few scenarios

- More flexible namespaces can overcome these issues

- A major challenge would be to extend them to statically typed languages