Do people in non-English-speaking countries code in English? [closed]

94

☆

103

I've heard it said (by coworkers) that everyone "codes in English" regardless of where they're from. I find that difficult to believe, however I wouldn't be surprised if, for most programming languages, the supported character set is relatively narrow.

Have you ever worked in a country where English is not the primary language?

If so, what did their code look like?

programming-languages spoken-languages

share edited Jun 23 '15 at 9:43 community wiki
6 revs, 4 users 48%
Damovisa

108 Answers

active oldest **votes**

1 2 3 4 next

88

I'm from Canada, but live in the States now.

It took me a while to get used to writing boolean variables with an "Is" prefix, instead of the "Eh" suffix that Canadians use when programming.

For example:

```
MyObj.IsVisible

MyObj.VisibleEh
```

share edited Oct 25 '11 at 21:35 community wiki
2 revs, 2 users 94%
AlishahNovin

21 This is a joke, right? – Philip Jun 10 '11 at 19:09

21 It better be, but its pretty funny. – HelloFictionalWorld Sep 16 '11 at 2:44

6 @Philip "This is a joke eh?" – rlemon Jun 26 '13 at 13:17

add a comment

Things about programming

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Where do we begin?

Where do we begin?

Starting is a tiny part of the process

The important part is the ongoing process

Some better questions:

- What is the process?
- What does day 17 look like?

What is the process?

- Remembering what we did on day 16
- Reading code we wrote before
- Problem solving to find out what needs changing where
  - Analyzing what the code does, and making decisions about what it needs to do
- Writing code

- What parts are the hardest / most time consuming?
  - And how can we make them faster / more efficient?

Efficiency

- Useless optimizations:
  - Typing speed
  - Adderall and IQ-juice™
  - Memory/brain exercises
- Actual optimizations:
  - Reading code:
    - Reading less, finding things quickly
    - Getting an understanding quickly
  - Problem solving:
    - Having standard solutions, routines
    - Recognizing patterns
    - Recognizing mistakes quickly

Efficiency

What parts of the process do we modify to get these traits?

- The code (mostly)
- Our way of thinking about code
  - Writing code for other humans, not for the computer

How do we create code that has these traits?

(Efficiently human-readable, modify-able)

Naming unnamable qualities

(aka code examples)

```
int fn(int a, int b, int c){
   return a + b - c;
}
```

```
int fn(int a){
   return a + b - c;
}
```

```
int fn(int a){
   return a + b() - c;
}
```

```
int b(void){
   return c++;
}
```

```
int requests_above(void){
    for(int f = floor+1; f < N_FLOORS; f++){
        for(int btn = 0; btn < N_BUTTONS; btn++){
            if(requests[f][btn]){
                return 1;
            }
        }
    }
    return 0;
}

int requests_below(void){
    for(int f = 0; f < floor; f++){
        for(int btn = 0; btn < N_BUTTONS; btn++){
            if(requests[f][btn]){
                return 1;
            }
        }
    }
    return 0;
}
```

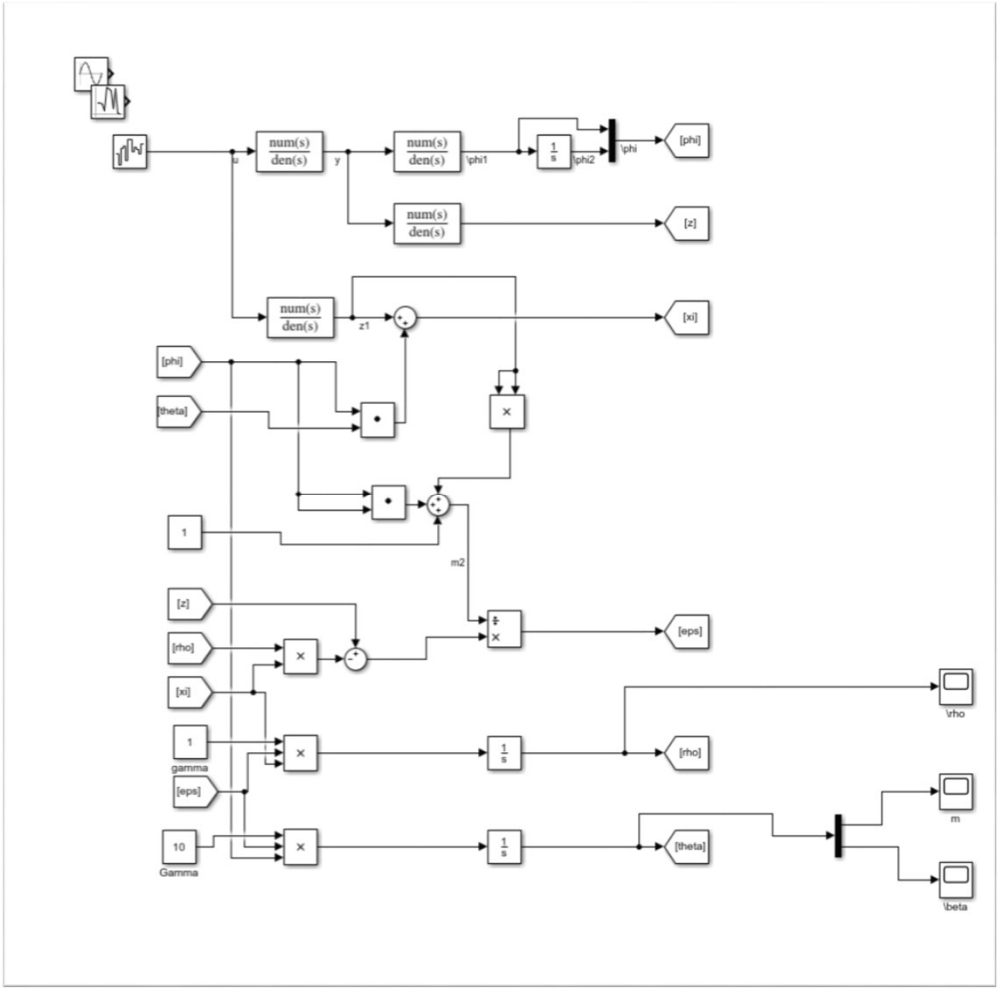
```
static int floor;
static int direction;
static int requests[N_FLOORS][N_BUTTONS];
static int fsmState;
```

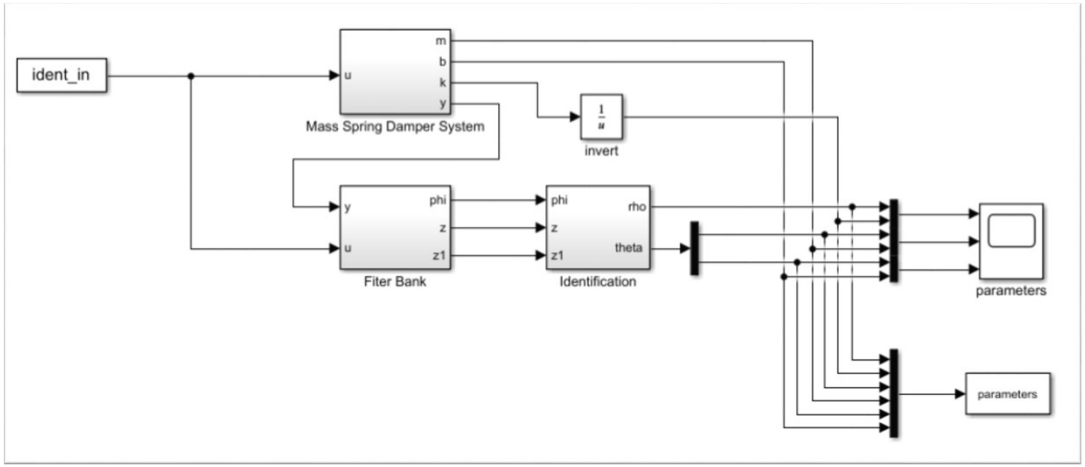
```
int requests_above(Elevator e){
    for(int f = e.floor+1; f < N_FLOORS; f++){
        for(int btn = 0; btn < N_BUTTONS; btn++){
            if(e.requests[f][btn]){
                return 1;
            }
        }
    }
    return 0;
}

int requests_below(Elevator e){
    for(int f = 0; f < e.floor; f++){
        for(int btn = 0; btn < N_BUTTONS; btn++){
            if(e.requests[f][btn]){
                return 1;
            }
        }
    }
    return 0;
}
```

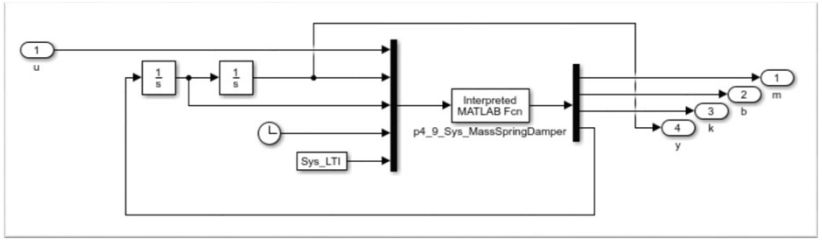
```
typedef enum {
    Idle,
    DoorOpen,
    Moving
} FSMState;

typedef struct {
    int         floor;
    Dirn        direction;
    int         requests[N_FLOORS][N_BUTTONS];
    FSMState    fsmState;
} Elevator;
```

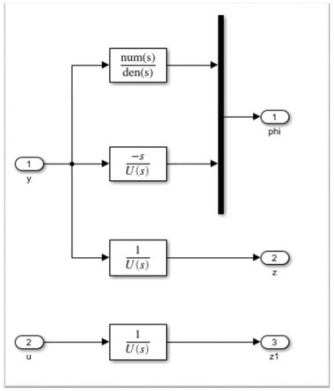
```
for(int i; i<len; i++){
    for(int j; j<2; j++){

        int ID = associatedOrders[i][j].ID;

        if(ID > 0  &&  !(ID in elevators)){
            associatedOrders[i][j].ID = -1;
        }

        if(ID != 0){
            btn.button      = j ? Containers.ButtonType.DOWN : Containers.ButtonType.UP;
            btn.floor       = i;
            bestID          = Algorithm.SelectBestElevator(btn, elevators, thisPeerID);
            timedOutLevel1  = (Clock.currTime - associatedOrders[i][j].time) > timeoutLevel1;
            timedOutLevel2  = (Clock.currTime - associatedOrders[i][j].time) > timeoutLevel2 + rnd;

            if((thisPeerID == bestID  &&  !timedOutLevel1  &&  ID == -1)    ||
                (thisPeerID == bestID  &&  timedOutLevel1)                  ||
                (timedOutLevel2)){
                    associatedOrders[i][j].ID    = thisPeerID;
                    associatedOrders[i][j].time  = Clock.currTime;
                    send(NetworkInterface, Messages.CheckIn(btn.floor, btn.button, thisPeerID));
                    send(LocalElevator,    Messages.CheckIn(btn.floor, btn.button, thisPeerID));
            }
        }
    }
}
```

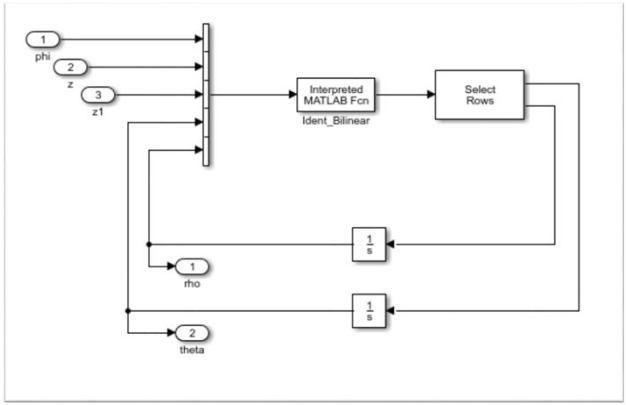
```
package main

import (
    "./Ressurs"
    "./fsm"
    "./network"
    "time"
)

func main() {
    ipchan               := make(chan string, 1)
    DoorTimer            := time.NewTimer(3*time.Second)
    DoorTimer.Stop()
    OrderChan            := make(chan Ressurs.ButtonOrder, 1)
    SendToNetwork        := make(chan string, 1)
    CostChan             := make(chan Ressurs.MSG, 2)
    ReadChan             := make(chan Ressurs.MSG, 2)
    floorArrivalTimeout  := make(chan bool, 1)
    stop_timer           := make(chan bool, 10)
    BackUpQueChan        := make(chan Ressurs.ButtonOrder, 1)

    go fsm.RunElevator(OrderChan, DoorTimer, floorArrivalTimeout, stop_timer, BackUpQueChan)
    go network.RunNetwork(DoorTimer, OrderChan, SendToNetwork, CostChan, ReadChan, ipchan,
                          floorArrivalTimeout, stop_timer, BackUpQueChan)

    select {}
}
```


num(s)
den(s)
u
y
\phi1
1
s
\phi2
\phi
[phi]
[z]
z1
[xi]
[theta]
m2
[eps]
[rho]
gamma
10
Gamma
\rho
m
\beta


ident_in
u
m
b
k
y
Mass Spring Damper System
invert
Fiter Bank
phi
z
z1
rho
theta
Identification
parameters
parameters


u
Interpreted MATLAB Fcn
p4_9_Sys_MassSpringDamper
Sys_LTI
m
b
k
y


num(s)
den(s)
phi
y
z
u
z1


phi
z
z1
Interpreted MATLAB Fcn
Ident_Bilinear
Select Rows
rho
theta

Complexity
Intertwined-ness
Simultaneity

Simple vs Easy

Simple:

sim- plex: one fold/twist

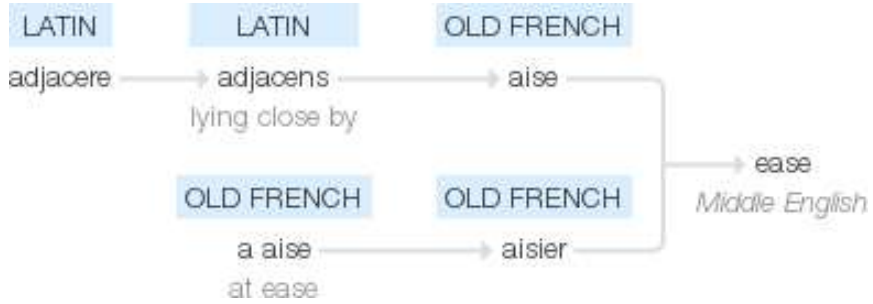
vs complex: many-fold / braided together

Easy:

lie near / be nearby

with little effort

vs hard: requiring endurance/effort



Simple

Simple means:

One role / task / concept / dimension

But not one instance / operation

It's about interleaving, not cardinality

Simple is objective

Easy

Easy means:

- Near / right there
- Already exists, quick/little effort to get started
- Near to our understanding, familiar

- Easy is subjective and relative
  - (What is easy to you might not be easy to others)

Complexity & understanding

We can only make something reliable if we understand it

We can only consider a few things at a time (~7 ±2)

Intertwined things must be considered together

Complexity undermines understanding

And therefore also quality, reliability, and long-term productivity

Easy and speed

- Easy is a short-term tactical decision
  - which quickly leads to local maxima
- Simple is a long term process
  - which requires effort, experimentation, and iteration
- Easy gives us a lot of speed in the start
  - Just add a global variable
  - Just add another parameter
  - Just add a new library
  - Just add more WD-40 and duct tape

Two more words

Complect:

To interleave/weave/twine

(archaic)

Compose:

To put/place together

Identifying complexity

But first

How to achieve easy?

Location: putting in our toolkit, installing stuff

Familiarity: Learning new stuff

Both of the above we can control

Mental capacity: Can't move our brain closer to the complexity

But we can make things nearer to our understanding by simplifying

And then

How to achieve simple?

Make things modular!

- Lecture over!

Of course not:

- There can still be lots of interconnections
- Don't be fooled by the trivialities of code organization and layout
- What matters is the kind of connections: abstractions

Modularity does not imply simplicity, but it does enable it

Oh by the way

Abstract:

To draw (something) away

To consider something separately from (sth else)

Abstracting does not mean "hiding stuff"

Abstracting means creating a new context where we can be absolutely precise

Car windows + windshield wipers + heater means you don't have to care about rain

Step 1: Choose simple things

Complex	Simple
state, objects	values, data
methods	functions
inheritance	templates
imperative loops	set-functions
conditionals	rules
inconsistency	consistency

Step 1: Choose simple things

- State is never simple:
  - Complects value and time
  - Now implicitly interleaves time with the rest of the program
    - And you can't get rid of it, or even really mitigate it
- Inheritance ties different types and their methods together
  - Templates let you define data structures and general-purpose functions separately
- Imperative loops (while, for) complect what and how
  - Set functions (map, filter) define how once and for all in the reusable function

Step 1: Choose simple things

Conclusion:

- Use raw data
- Use pure (stateless, share-nothing) functions

- Avoid all this object-oriented nonsense

Step 2: Create new simple things

Abstract for simplicity

The "surface area" of a new piece of code should be small and precise

Ask yourself: "Who, what, when, where, why, how?"

- What does this thing do, and what does it need?

Try to conclude that "I don't know and I don't need to know"

- Needing to know means there's some interaction or assumption

Creating good abstractions is hard

- And you won't get it right the first time

Step 2: What

What:

- Operations, behavior imparted on the system
  - Functions (or sets of functions), processes/servers

Use template-like things wherever useful

- Easier in some languages than others

"Do one thing and do it well"

Don't complect with how

Step 2: Who

- Who:
  - Data, entities
    - Things that abstractions are connected to
    - Things that functions/operations use and transform

- Build components from sub-components directly
  - Don't hardwire subcomponents together
  - Pass data as parameters explicitly (not by objects-like implicit sharing)

- Don't complect with component details or other components
  - Don't have self-references, or even avoid references entirely

Step 2: How

- How:
  - Also known as implementation code
- Prefer abstractions that don't dictate how
  - Declarative is better than imperative
    - (Again, language dependent)
- Don't complect with anything

Step 2: When, where

- Very easy to complect these with everything
  - For A to call B, A must know where B is
- Queues or channels can help here

Step 2: Why

Why:

The rules and specification of the application

Often all over the place in the code

In conditionals, complected with control flow

Prefer rules and declarative logic systems

(Again, generally bad language or library support for this stuff)

Step 2: A reminder

Remember that this is not about cardinality

Do not create artificial divisions based on stupid metrics

- like "lines of code in a function must be less than x"

Divide things into smaller pieces only if it provides simplicity

- Only if it reduces the number of complections (or kinds of complections)
- Having to look all over the place to follow a straight line is dumb

These are guidelines, not rules

Step 3: Dealing with nasty things

- Two kinds of complexity:
  - Inherent complexity
    - The problem itself is complected
  - Accidental complexity
    - Latin for "It's your fault"

- How do we deal with real-world complected things?
  - Input/output
    - Complects what & when. Outputs cannot be un-done
  - Network / file systems / hardware in general
    - Things that can (and will) fail
    - Or are always unreliable, even when working normally

Observations

1: Logic/algorithms can always be separated
from irreversible interactions

No graph-traversal code needs to read from disk
No elevator direction-choosing algorithm needs to write to network

Observations

2: Encapsulation is a failure

If a thing-doer is responsible for some other thing-doer, we either get

- Cycles, where two thing-doers are responsible for (own) eachother
- A hierarchy, where two "leaves" cannot talk to eachother

(Does the order distributer own the network or vice versa?)

Which means a thing-doer must not be accessible from multiple other thing-doers

A "setter" accessible from two other outside classes is just a "global" variable

Which also means we must not send references to thing-doers as messages

Or in parameters to methods

Encapsulation in terms of ownership is a useless way of thinking

Observations

3: Sharing is bad

Sharing means either

- Things being modified from multiple places (aka "anywhere")
- Duplication of data, giving consistency problems

Multiple readers & single writer does not have this problem

Copies are unchangeable values, pure data with no references to thing-doers

But note that copies go out of date "instantly"

(especially when considering network)

Step 3: Dealing with nasty things

Conclusion:

State & stateful things is a necessary evil

Make it as simple as possible by:

- Making state changes in one (cardinal) place

  (for that one piece of state, don't put everything in one place unless it makes sense)

- Separating the logic of how state changes are made (algorithms) from where

Step 3: Dealing with nasty things

Stateless core - stateful shell

The core:

- Pure functions, stateless.
- All the stuff that is fun to write, algorithms etc.
- Lots of decisions, no dependencies

The shell:

- Thing-doers are their own processes/servers/objects
- The state the thing-doer writes to is put in that thing-doer
  - Writing has no logic, that's in the core
- Any other needed state/data are copies received/requested from other thing-doers
- Lots of dependencies, no (or very simple) decisions

Step 4: Creating entire programs

1: Start with the data

2: Find what inputs modify the data

3: Find the outputs

4: Segment the data based on what modifies it

   And create a persistent thing-doer for each

5: Wire it together with queues or channels

   And only send values over them

6: Iterate when you encounter complexities