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

Simple vs easy

The rules and the ethic

The elements of code quality

- "Advice", "rules", "best practices", "checklists"
- Sometimes seemingly contradictory
  - Splitting vs grouping
  - Completeness vs extensibility
- Often appears to be arbitrary
  - Number of parameters
  - Lines of code
- Usually involves maturity or experience
  - Finding good names
- Always contains jargon
  - "Consistent abstraction", SOLID

Ethic and rules

There is nothing wrong with rules

- Rules are often necessary preconditions for discipline

Rules conflict and don't always apply

- There has to be an underlying ethic that generates the rules
- The universe possesses *some quality* that makes the rules (more or less) valid

Respecting and adhering to this "universal quality" or ethic is more important than respecting the rules

Identifying and understanding this ethic is paramount

Identifying the universal

- The "ultimate quality" must be:
  - Valid across the widest possible range of software artifacts
    - Apply to different projects in different languages
  - Valid at all levels of resolution within an artifact
    - Apply to line, function, module, file, etc
  - Quantifiable, with minimal room for interpretation
    - Be measurable, to resolve disagreement
  - Qualifiable, to resonate with the programmers
- Basically a pragmatist approach

The act of programming

- Good code comes from good programmers
  - Note: good, clever, smart, intelligent are all different words
- Criticism of code is criticism of the person that wrote it
  - Pay attention to how personally you take things
  - There is more to life than being a good programmer
    - But for some of you it's part of it
  - Pay attention to work-life balance etc etc

For today and later

Today: Just programming

- What does it mean for code to be good?
- What is the value hierarchy of good code?
- And how do we quantify and qualify it?
- What are the ideals, archetypes and axioms of good code?

Not today: The underlying ethics

- What is "good"?
- What is the value hierarchy of "good"?
- What are the ideals, archetypes, and axioms of X philosophy?
- Where do values come from?
- And why is it 42?

The common rule

Previously: My rules

Data and function

- Separate functionality into core and shell
- Disassociate data and function, make pure functions
- Separate data based on who writes to it
- Align data flow and control flow
- Unmix happy path and error path

Process and communication

- Processes are useful if they are stateful
- Send messages (just values) instead of invoking behavior
- Messages are accepted or rejected based on protocols and rules
- Express these rules with state machines

Finding common factors

Data and function:

- Separate, disassociate, unmix, align, purify
- Straightness, lack of interweaving

Process and communication:

- Manage state with rules, spread only values
- Containment, boundaries, orderliness

In common:

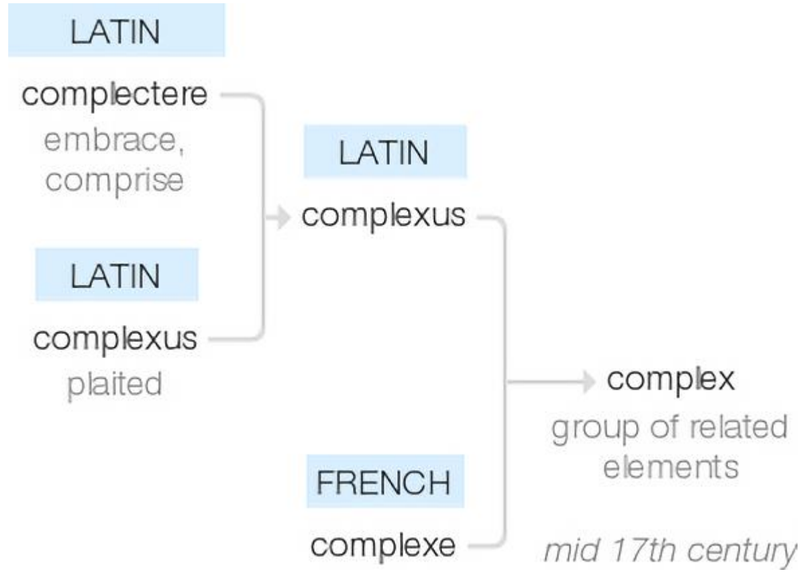
- Simplicity

Defining Simple

Simple:

sim- plex: one fold/twist

vs complex: many-fold / braided together



Simple means:

One role / task / concept / dimension

But not one instance / operation

It's about interleaving, not cardinality

Simple is objective, not subjective

(At least pragmatically objective - everyone reasonable can agree on it)

The pain

What drives us away from simplicity?

What inspires us to choose the path to bad solutions?

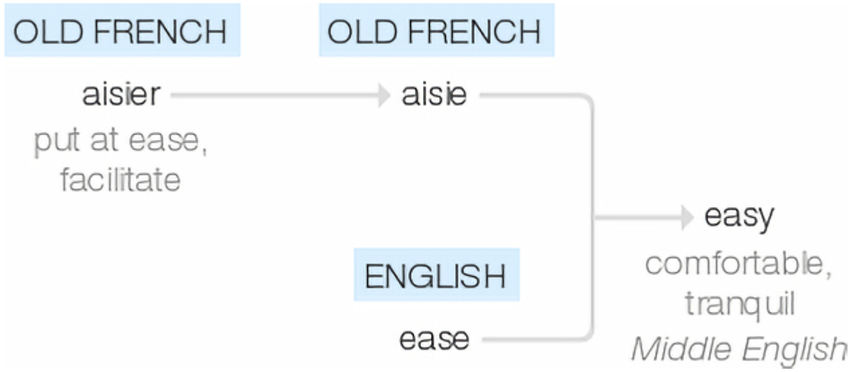
It's easy

Literally

Defining Easy

Easy:

from same origin as adjacent: lie near / be nearby



Easy means:

Near, right there

Already exists, quick to get started

Near to our understanding, familiar

Easy is subjective and relative

(what is easy to you might not be easy to others)

Two kinds of complexity

Inherent complexity:

- The problem itself is complected
- This sets the lower bound for the complexity in the code

Accidental complexity:

- It's your fault

Code quality "rules" exist to manage accidental complexity

- Rules are self-imposed limits
- This limit is the limit of human perception and comprehension

The speed

- Easy is a short term tactical decision
  - Quickly reaches the maximum complexity we can manage
- Simple is a long term process
  - Which requires effort, discipline, 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

Analyzing some rules

Rules about modules

Composition of several, low coupling between, high cohesion within

Use without knowing internals, maintain without knowing externals

Minimize accessibility to internals

Hide implementation details (and "hiding" in general)

Minimize direct calls to other modules

Fewer than X data members and Y parameters

I have questions about modules

How do we know that we don't need to know internals when we wrote it and therefore already know it?

How do we know if we don't need to know the externals if the externals is why we wrote it in the first place?

How do we "minimize", generally?

How large is X and Y?

Why do we want to hide things? What are we afraid of?

Why do we want to minimize the number of things?

Rules about functions

Do one thing and do it well

Defensive, programmatically asserts assumptions

Less than X parameters

Good name, based on what it outputs

I have questions about functions

What is "one thing", and what is "doing it well"?

What is X?

What is a "good name", or "good" in general?

Rules are hard

These rules seem awfully vague and incomplete...

Following these rules is hard

- And won't necessarily lead to good code

These rules are too much "up to interpretation"

- Hard to make something everyone agrees is good

You must understand the underlying ethic
to understand the rules it generates

Making simple easy

How to achieve simple?

- Make things modular
  - Lecture over!

- Of course not:
  - The problem was interweaving
  - Modules can have lots of tight connections
  - What matters is the *kind* of connections

- Modularity does not imply simplicity, but does enable it

- Instead, start with choosing simple things

Step 1: Choose simple things

Simple	Complex	Complects
values, data	state, objects	everything that touches it & time
functions	methods	function & state (& time)
templates	inheritance	types
set-functions	imperative loops	what & how
rules	conditionals	the program's behavior & structure
consistency	inconsistency	different kinds of falsehoods, time

Allowing inherent complexity

- The code must manifest the inherent complexity
- State/Input/output is always necessary:
  - Complects value and time
  - Now implicitly interleaves time with the rest of the program
- Inconsistency is often necessary:
  - Complects value-at-time and space (or just time and space)

Step 2: Ask the 5W1H questions

Let's think about programming a new thing

Let's ask all the obvious questions:

Who, what, when, where, why, how?

- Avoid complecting these, as much as possible

2-1

What:

- Operations, behavior imparted on the system
  - (Functions, processes, etc)
- Use template-like things wherever useful
  - Avoids complecting data types

Who:

- Data, entities
  - Things that operations use or transform
- Compose components from sub-components directly
  - Don't invent unnecessary data types
  - Pass data as parameters explicitly
- Don't complect with other components
  - Don't have references (pointers)

2-2

How:

- The implementation code itself
- Prefer declarative over imperative (if possible)
- Don't complect with anything
  - (Function internals shouldn't leak outside the function)

When, where:

- Where you put the code, literally (lines, files)
- Very easy to complect with everything
  - If A calls B, then A must know where B is
- Use queues and channels
  - Again, don't have references

2-3

Why:

- The logic that manifests the specification
  - Often all over the place
  - (Conditionals, control flow, etc)
- Prefer rules and declarative logic systems
- Generally poor language and library support for this...

2-4 Reminder

Remember that this is not about cardinality

Do not create artificial divisions/separations based on stupid metrics like number of lines of code in a function

Divide existing things into smaller pieces
only if it provides a simplicity benefit

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

Some observations

Observation 1: Boundaries

Logic/algorithm can almost always be separated from irreversible interactions

- Exception: distributed interactions (transactions)

This is the core/shell divide

Observation 2: Sharing

- Sharing is bad because it complects
  - Either: Sharing - when things are modified from multiple places
  - Or: Duplication - leads to inconsistency
- Multiple-readers single-writer does not have this issue
  - Copies of data are just values, no references to thing-doers
  - Note that copies go out of date "instantly"

Observation 3: Objection

- Object-oriented programming is fundamentally about adding complexity
  - Objects complect data and function
  - Object interactions require references
    - This complects ownerships and responsibilities
    - (Or limits interactions to a strict hierarchy, which has poor modeling power)
  - All-or-nothing public interface complects actual and potential interactions
  - Inheritance complects super- and sub-types
  - Polymorphism complects potential outcomes from a single interaction

OOP is poorly defined

Construct	Concept	Definition
Structure	Abstraction	Creating classes to simplify aspects of reality using distinctions inherent to the problem.
	Class	A description of the organization and actions shared by one or more similar objects.
	Encapsulation	Designing classes and objects to restrict access to the data and behavior by defining a limited set of messages that an object can receive.
	Inheritance	The data and behavior of one class is included in or used as the basis for another class.
	Object	An individual, identifiable item, either real or abstract, which contains data about itself and the descriptions of its manipulations of the data.
Behavior	Message Passing	An object sends data to another object or asks another object to invoke a method.
	Method	A way to access, set, or manipulate an object's information.
	Polymorphism	Different classes may respond to the same message and each implement it appropriately.

Deborah Armstrong:
The Quarks of Object Oriented Development

"Actually I made up the term 'object-oriented', and I can tell you I did not have C++ in mind."
- Alan Kay

OOP is killing itself slowly

- Contemporary "OOP" languages keep replacing OOP constructs
  - "Composition over inheritance"
  - Java adds generics in 5.0 (2004)
  - New languages without static types are somehow "OOP" (Python, Ruby, JS)
  - Lambdas and closures added to C++11 and Java 8
  - "C++ supports OOP and other programming styles, but is deliberately not limited to any narrow view of 'Object Oriented.'" - Bjarne Stroustrup
  - Game engine programming is moving towards entity component system structure
- Contemporary object oriented programming is a Perversion of the Original Object Principles
  - or P.O.O.P. for short