How good are you at programming?

A CEFR-like approach to measure programming proficiency

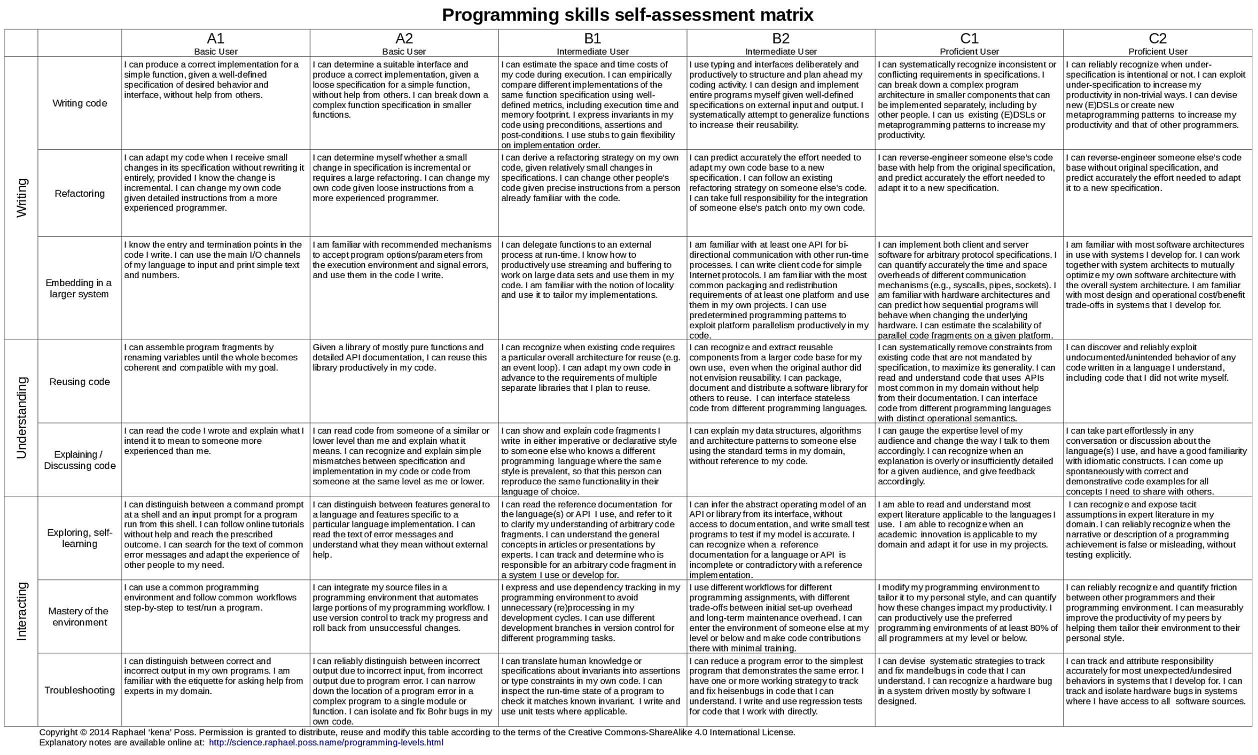
Raphael ‘kena’ Poss

July 2014

modified:	2018-09-03
category:	Programming
slug:	programming-levels
subtitle:	A CEFR-like approach to measure programming proficiency
tags:	methodology

Contents

The table



Programming skills self-assessment matrix

		A1 Basic User	A2 Basic User	B1 Intermediate User	B2 Intermediate User	C1 Proficient User	C2 Proficient User
Writing	Writing code	I can produce a correct implementation for a simple function, given a well-defined specification of desired behavior and interface, without help from others.	I can determine a suitable interface and produce a correct implementation, given a loose specification for a simple function, without help from others. I can break down a complex function specification in smaller functions.	I can estimate the space and time costs of my code during execution. I can empirically compare different implementations of the same function specification using well-defined metrics, including execution time and memory footprint. I express invariants in my code using preconditions, assertions and post-conditions. I use stubs to gain flexibility on implementation order.	I use typing and interfaces deliberately and productively to structure and plan ahead my coding activity. I can design and implement entire programs myself given well-defined specifications on external input and output. I systematically attempt to generalize functions to increase their reusability.	I can systematically recognize inconsistent or conflicting requirements in specifications. I can break down a complex program architecture in smaller components that can be implemented separately, including by other people. I can us existing (E)DSLs or metaprogramming patterns to increase my productivity.	I can reliably recognize when under-specification is intentional or not. I can exploit under-specification to increase my productivity in non-trivial ways. I can devise new (E)DSLs or create new metaprogramming patterns to increase my productivity and that of other programmers.
	Refactoring	I can adapt my code when I receive small changes in its specification without rewriting it entirely, provided I know the change is incremental. I can change my own code given detailed instructions from a more experienced programmer.	I can determine myself whether a small change in specification is incremental or requires a large refactoring. I can change my own code given loose instructions from a more experienced programmer.	I can derive a refactoring strategy on my own code, given relatively small changes in specifications. I can change other people's code given precise instructions from a person already familiar with the code.	I can predict accurately the effort needed to adapt my own code base to a new specification. I can follow an existing refactoring strategy on someone else's code. I can take full responsibility for the integration of someone else's patch onto my own code.	I can reverse-engineer someone else's code base with help from the original specification, and predict accurately the effort needed to adapt it to a new specification.	I can reverse-engineer someone else's code base without original specification, and predict accurately the effort needed to adapt it to a new specification.
	Embedding in a larger system	I know the entry and termination points in the code I write. I can use the main I/O channels of my language to input and print simple text and numbers.	I am familiar with recommended mechanisms to accept program options/parameters from the execution environment and signal errors, and use them in the code I write.	I can delegate functions to an external process at run-time. I know how to productively use streaming and buffering to work on large data sets and use them in my code. I am familiar with the notion of locality and use it to tailor my implementations.	I am familiar with at least one API for bi-directional communication with other run-time processes. I can write client code for simple Internet protocols. I am familiar with the most common packaging and redistribution requirements of at least one platform and use them in my own projects. I can use predetermined programming patterns to exploit platform parallelism productively in my code.	I can implement both client and server software for arbitrary protocol specifications. I can quantify accurately the time and space overheads of different communication mechanisms (e.g., syscalls, pipes, sockets). I am familiar with hardware architectures and can predict how sequential programs will behave when changing the underlying hardware. I can estimate the scalability of parallel code fragments on a given platform.	I am familiar with most software architectures in use with systems I develop for. I can work together with system architects to mutually optimize my own software architecture with the overall system architecture. I am familiar with most design and operational cost/benefit trade-offs in systems that I develop for.
Understanding	Reusing code	I can assemble program fragments by renaming variables until the whole becomes coherent and compatible with my goal.	Given a library of mostly pure functions and detailed API documentation, I can reuse this library productively in my code.	I can recognize when existing code requires a particular overall architecture for reuse (e.g. an event loop). I can adapt my own code in advance to the requirements of multiple separate libraries that I plan to reuse.	I can recognize and extract reusable components from a larger code base for my own use, even when the original author did not envision reusability. I can package, document and distribute a software library for others to reuse. I can interface stateless code from different programming languages.	I can systematically remove constraints from existing code that are not mandated by specification, to maximize its generality. I can read and understand code that uses APIs most common in my domain without help from their documentation. I can interface code from different programming languages with distinct operational semantics.	I can discover and reliably exploit undocumented/unintended behavior of any code written in a language I understand, including code that I did not write myself.
	Explaining / Discussing code	I can read the code I wrote and explain what I intend it to mean to someone more experienced than me.	I can read code from someone of a similar or lower level than me and explain what it means. I can recognize and explain simple mismatches between specification and implementation in my code or code from someone at the same level as me or lower.	I can show and explain code fragments I write in either imperative or declarative style to someone else who knows a different programming language where the same style is prevalent, so that this person can reproduce the same functionality in their language of choice.	I can explain my data structures, algorithms and architecture patterns to someone else using the standard terms in my domain, without reference to my code.	I can gauge the expertise level of my audience and change the way I talk to them accordingly. I can recognize when an explanation is overly or insufficiently detailed for a given audience, and give feedback accordingly.	I can take part effortlessly in any conversation or discussion about the language(s) I use, and have a good familiarity with idiomatic constructs. I can come up spontaneously with correct and demonstrative code examples for all concepts I need to share with others.
Interacting	Exploring, self-learning	I can distinguish between a command prompt at a shell and an input prompt for a program run from this shell. I can follow online tutorials without help and reach the prescribed outcome. I can search for the text of common error messages and adapt the experience of other people to my need.	I can distinguish between features general to a language and features specific to a particular language implementation. I can read the text of error messages and understand what they mean without external help.	I can read the reference documentation for the language(s) or API I use, and refer to it to clarify my understanding of arbitrary code fragments. I can understand the general concepts in articles or presentations by experts. I can track and determine who is responsible for an arbitrary code fragment in a system I use or develop for.	I can infer the abstract operating model of an API or library from its interface, without access to documentation, and write small test programs to test if my model is accurate. I can recognize when a reference documentation for a language or API is incomplete or contradictory with a reference implementation.	I am able to read and understand most expert literature applicable to the languages I use. I am able to recognize when an academic innovation is applicable to my domain and adapt it for use in my projects.	I can recognize and expose tacit assumptions in expert literature in my domain. I can reliably recognize when the narrative or description of a programming achievement is false or misleading, without testing explicitly.
	Mastery of the environment	I can use a common programming environment and follow common workflows step-by-step to test/run a program.	I can integrate my source files in a programming environment that automates large portions of my programming workflow. I use version control to track my progress and roll back from unsuccessful changes.	I express and use dependency tracking in my programming environment to avoid unnecessary (re)processing in my development cycles. I can use different development branches in version control for different programming tasks.	I use different workflows for different programming assignments, with different trade-offs between initial set-up overhead and long-term maintenance overhead. I can enter the environment of someone else at my level or below and make code contributions there with minimal training.	I modify my programming environment to tailor it to my personal style, and can quantify how these changes impact my productivity. I can productively use the preferred programming environments of at least 80% of all programmers at my level or below.	I can reliably recognize and quantify friction between other programmers and their programming environment. I can measurably improve the productivity of my peers by helping them tailor their environment to their personal style.
	Troubleshooting	I can distinguish between correct and incorrect output in my own programs. I am familiar with the etiquette for asking help from experts in my domain.	I can reliably distinguish between incorrect output due to incorrect input, from incorrect output due to program error. I can narrow down the location of a program error in a complex program to a single module or function. I can isolate and fix Bohr bugs in my own code.	I can translate human knowledge or specifications about invariants into assertions or type constraints in my own code. I can inspect the run-time state of a program to check it matches known invariant. I write and use unit tests where applicable.	I can reduce a program error to the simplest program that demonstrates the same error. I have one or more working strategy to track and fix heisenbugs in code that I can understand. I write and use regression tests for code that I work with directly.	I can devise systematic strategies to track and fix mandelbugs in code that I can understand. I can recognize a hardware bug in a system driven mostly by software I designed.	I can track and attribute responsibility accurately for most unexpected/undesired behaviors in systems that I develop for. I can track and isolate hardware bugs in systems where I have access to all software sources.

Explanatory notes are available online at: http://science.raphael.poss.name/programming-levels.html

Also available in PDF form.

The online test

A web application is available to assess your own skills according to this table for one or more programming languages.

How to use this table

The table characterizes the proficiency level (columns) of programmers of a particular programming language in the context of different programming activities (rows).

This table is inspired by the CEFR table of the same name, for assessing proficiency in natural languages. Like the CEFR, this table divides learners into three broad level divisions: "Basic user" (A), "Independent user" (B) and "Proficient user" (C). The broad divisions are each further divided in two levels (A1, A2, B1, B2, C1, C2) that correspond to testable milestones in language acquisition.

This table can be used in different ways, for example:

- row by row, to assess one's own level per activity (different skill levels for different activities are possible);
- column by column from left to right, to determine one's own minimum level for a programming language (the rightmost level where all requirements in the column and all columns to the left are matched);

- column by column from right to left, to determine one's most developed skill (the rightmost level where any requirement in the column is matched);
- language per language, to assess one's own relative proficiency in different programming languages.

Possible applications

- to track one's own progress while learning how to program;

  for example: *this year, I transitioned from A2 to B2 in C++. For Java, I am B1 for understanding but still A2 for writing.*
- to advertise the educational goals of a programming course;

  for example: *this Java introductory course will bring you to level A1 or A2 for all activities.*
- to advertise one's own skillset to peers or prospective employers;

  for example: *I am C1 in Python, B2 in O'Caml and A2 in Haskell.*
- to set basic level requirements for courses or professional activities:

  for example: *This course requires A2 proficiency with a language in the C family.*
- to select a programming course that best matches one's skill level;

  for example: *My current level is A2 but this course requires B1, so I will need some extra work before starting.*
- to coordinate the teaching objectives of successive programming courses in a curriculum;

  for example: *Our introductory course brings students to A2 in Java, but our follow-up program requires B1 or B2, so we need to propose a supplementary course for that level.*

Design methodology

The table was designed following the CEFR methodology:

First, recognize the different modalities (production, reception, interaction, mediation). Then group and abstract activities in each modality by clusters that share similar motivations and actors. The resulting set becomes the rows in the matrix.

The requirements for "A" levels are then phrased to identify users that can perform language acts under supervision or under dedicated/personalized guidance from peers.

The requirements for "B" levels are then phrased to identify users that can perform language acts without supervision or guidance from peers, or with minimal effort from peers.

The requirements for "C" levels are then phrased to identify users who are fully independent and who can demonstrate skills corresponding to a high expertise level according to the majority of other users of the language.

References

- *Common European Framework of Reference for Languages: Learning, Teaching, Assessment (CEFR).* Language Policy Unit, Strasbourg. Available from: http://www.coe.int/lang-CEFR
- Wikipedia: Common European Framework of Reference for Languages.

Acknowledgements

The initial idea of an assessment matrix appeared during discussions with Merijn Verstraaten about average programmer skills in online fora. The concrete form as presented here was further developed thanks to inspiration and suggestion from Philip Hölzenspies, Jan Kuper, Cosmin Dumitru, Taddeüs Kroes, and many others.

Copyright and licensing