Competence

Nikita Sharov v. 06/11/2019

Based on the Programmer Competency Matrix by Sijin Joseph, oldie but goldie.

Knowledge

- Has heard of upcoming technologies in the field;
- knows about alternatives to popular and standard tools.

Data Structures

- Knows basic database concepts, normalization, ACID, transactions;
- knows the difference between clustered and non-clustered indexes;
- understands how data and indexes are stored internally;
- knows how hashtables can be implemented and how to handle collisions;
- knows space and time tradeoffs of arrays, linked lists, dictionaries, priority queues etc.

Systems Programming

- Understands the entire programming stack (compilers, linkers and interpreters; just-in-time compilation; static and dynamic linking; binary and assembly code; garbage collection, heap, stack, memory addressing);
- understands static / dynamic typing, weak / strong typing, type inference, lazy evaluation;
- understands what assembly code is and how things work at the hardware level (CPU / memory / cache / interrupts / microcode);
- knows about virtual memory and paging;
- understands kernel / user mode, multi-threading and synchronization primitives;
- understands how networks work, understanding of network protocols and socket level programming.

Experience

10+ years of professional experience.

- Has used more than one framework in a professional capacity and is well-versed with the idioms of the frameworks;
- has written libraries that sit on top of the APIs to simplify frequently used tasks and to fill in gaps in the API;
- has written automated unit tests; has written code in TDD manner.

Languages

- C (compiled, curly-brace, imperative, procedural, system programming language with manual, deterministic memory management);
- C++ (compiled, curly-brace, imperative, metaprogramming, multiparadigm, object-oriented [class-based, single dispatch], procedural, system programming language with manual, deterministic memory management);
- **C#** (compiled [into IL], curly-brace, functional [impure], imperative, interactive mode, iterative, garbage collected, multiparadigm, object-oriented [class-based, single dispatch], procedural, reflective language);
- **CSS** (style sheet language);
- **HTML** (markup language);

- **JavaScript** / ECMAScript (curly-brace, embeddable [in source code, client- and server-side], extension, functional [impure], interactive mode, interpreted, garbage collected, multiparadigm, object-oriented [prototype-based], procedural, reflective, scripting language);
- **Mardown** (markup language);
- PHP (curly-brace, embeddable [in source code, server-side], imperative, interpreted, imperative, iterative, garbage collected [combined with automated reference counting], multiparadigm, object-oriented [class-based, single dispatch], reflective, scripting language);
- **PowerShell** (command line interface, curly-brace, extension, garbage collected, imperative, interactive mode, interpreted, multiparadigm, procedural, reflective, scripting language);
- **SQL** (data-oriented, declarative, fourth-generation, little language [serving a specialized problem domain]);
- **SVG** (markup language);
- **TypeScript** / JavaScript / ECMAScript (class-based superset of JavaScript);
- VBA (compiled [into IL], extension, procedural, scripting language with automated reference counting);
- **XML** (meta markup language).

Platforms

- Linux (Debian);
- **Microsoft Azure**;
- Microsoft DOS (6.22);
- **Microsoft Windows** (CE / XP / Server / Vista / 7 / 8 / 10).

Capacity

- Able to understand the complete picture;
- able to recognize and code dynamic programming solutions;
- able to communicate thoughts / design / ideas / specs in an unambiguous manner;
- able to effectively use the IDE using menus and keyboard shortcuts for most used operations.

Configuration Management

- Able to setup automated functional, load / performance and UI tests;
- able to setup continuous delivery and deployment;
- proficient in using centralized (CVS / SVN / TFS) and distributed version control systems (Git / Mercurial);
- knows how to branch and merge, setup repository properties etc.

Systems Decomposition

- Able to break up problem space and design solutions, that span multiple technologies / platforms;
- able to visualize and design systems with multiple product lines and integrations with external systems;
- able to design operations support systems like monitoring, reporting etc.

Problem Decomposition

- Able to break up problems into multiple functions;
- able to come up with reusable functions / objects that solve the overall problem;
- able to write generic / object-oriented code, that encapsulate aspects of the problem that are subject to change, using appropriate data structures and algorithms.

Data Processing

- Able to design good and normalized database schemas keeping in mind the queries that will have to be run;
- knows basic sorting, searching and data structure traversal and retrieval algorithms;
- proficient in use of ORM tools;
- can do basic database administration, performance / index / query optimization, write advanced select queries;
- understands how databases can be mirrored, replicated etc.

Modus Operandi

Follows Kent Beck's four rules of Simple Design (in Martin Fowler's interpretation), in order of importance:

1. Passes the tests (including non-automated / manually executed tests)
2. Reveals intention (purpose is easy to understand)
3. Contains no duplication (everything should be said "once and only once")
4. Minimizes the number of elements (anything that doesn't serve the three prior rules should be removed)

Requirements

- Comes up with questions regarding missed cases / areas that need to be speced;
- suggests better alternatives / flows to given requirements based on experience, feedback, tests and measurements;
- adjusts communication as per the context; peers can understand what is being said.

Defensive Coding

- Asserts critical assumptions in code;
- checks externally set arguments;
- makes sure to check return values and check for exceptions around code that can fail;
- comes up with good unit test cases for the code that is being written;
- writes unit tests that simulate faults.

Error Handling

- Comes up with guidelines on exception handling for the entire system;
- codes to detect possible exception before;
- maintains consistent exception handling strategy in all layers of code;
- ensures that errors / exceptions leave the program in a well-defined state; resources, connections and memory is all cleaned up properly.

Delivery

Source Tree Organization

Relates to the entire set of artifacts that define the system.

- No circular dependencies; binaries, libs, docs, builds, third-party code all organized into appropriate folders;
- physical layout of the source tree matches logical hierarchy and organization; directory naming and organization provide insights into the design of the whole system.

Code Organization Across Files

- Related source files are grouped into folders;
- each physical file has a unique purpose (for e.g. one class definition, one feature implementation etc.);
- code organization at the physical level closely matches design, and looking at file names and folder distribution provides insights into design.

Code Organization Within a File

- Methods are grouped logically and / or by accessibility;
- code is grouped into regions and well commented with references to other source files;
- consistent white space usage; files look beautiful.

Code Readability

- Good names for files, folders, variables, classes, methods etc.;
- no long functions; comments explaining unusual code, bug fixes and code assumptions;
- code flows naturally; no deep nesting of conditionals or methods.