Kdy se kód čte jako kniha?

Kar(t)el Kočí

29.2.2020

Je čitelnost důležitá?

Proč se vlastně zabývat čitelností kódu?

- protože kód píšeme jednou, ale čteme mnohem mnohem vícekrát
- protože čitelný kód je verifikovatelný kód
- protože čitelný kód je rozšiřitelný kód
- protože nikdo vám nepřispěje i když to je open-source pokud to nepřečte
- protože zkrátíte review proces
- protože vás budou mít rádi jak aktuální i budoucí kolegové
- protože vás bude mít rádo vaše budoucí já

Základní koncepty

- Kód má nejenom fungovat, ale hlavně sdělovat myšlenku jak má fungovat
- Každý a i prázdný znak sděluje myšlenku
- Segmentace do celků
- Naznačování důležitosti pomocí pojmenování a omezení přístupu
- Exaktní a trefné pojmenování
- Dodržování stylu jazyka

Pojmenujte nepojmenované

```
configure() {
        sed -i "s/@$1@/$2/g" "$3"
        dbg "Applied '$1=$2' on $3"
}
```

```
configure() {
        local variable="$1"
        local value="$2"
        local file="$3"
        sed -i "s/@$variable@/$value/g" "$file"
        dbg "Applied '$variable=$value' on $file"
}
```

Nepřehánějte to ale s vatou

```
get_home() {
        local user="$1"
        awk -F : -v "user=$user" '$1 == user { print $7 }' /etc/passwd
}
```

```
get_home() {
        awk -F : -v "user=$1" '$1 == user { print $7 }' /etc/passwd
}
```

Nepřipravujte se na třetí světovou

```
case "$operation" in
        add) prepare_add "$@" ;;
        rem|remove) prepare_rem "$@" ;;
        *) fail "Invalid mode specified:" "$mode" ;;
esac
echo "Beginning operation: $operation"
case "$operation" in
        add) do_add "$@" ;;
        rem|remove) do_rem "$@" ;;
        *) fail "Internal error: Unknown mode:" "$mode" ;;
esac
```

Jmenujme obecně a exaktně

```
char *read_file(const char *path);
```

je lepší než:

```
char *slurp(const char *path);
```

Protože Perl..

Čtenář není kompilátor aneb char není int8_t

```
char *receive_message(socket_t);
```

```
void receive_message(socket_t, uint8_t **data, size_t *len);
```

Pořadí funkcí

```
_compile() {
        ...
}
compile_tools() { _compile tools/compile toolchain/compile; }
compile_target() { _compile target/compile; }
compile_packages() { _compile package/compile; }
compile() {
        compile_tools
        compile_target
        compile_packages
}
```

Protože knihu také nečteme od konce

Segmenty je lepší pojmenovat

```
##############################################
print_help() {
...
##############################################
operation_add() {
...
```

```
#Argument parsing ############################
print_help() {
...
# Operation ADD ##############################
operation_add() {
...
```

Segmenty je lepší zapouzdřit

```
struct config *parse_args(int argc, const char **argv) {
        ...
}

int main(int argc, const char **argv) {
        struct config *conf = parse_args(argc, argv);
        ...
}
```

Omezte definiční obor

```
char *scan_file(const char *fpath, const char *regexp) {
        FILE *file;
        regex_t preg;
        regcomp(&preg, regexp, 0);
        ...
        file = fopen(fpath, "r");
        ...
        fclose(file);
        return result;
}
```

Nejlepší kód není třeba dokumentovat?

```
# Parse arguments
struct conf *parse_arguments(argc, argv);
# Load configuration
load_config(conf);
# Invoke appropriate handler
switch (conf->operation) {
```

Ano, ale to neplatí o API!

```
int pop_int(stack_t, int flags);
const char *pop_string(stack_t, int flags);
```

Dokumentujte globální proměnné

```
# Possible values:
# running: container is running
# stopped: container is stopped
# none: container does not exists
STATE="none"
update_state() { STATE="$(container status)"; }
access() {
        case "$STATE" in
                running) container sh ;;
                *) fail "Can't access container in state: $STATE" ;;
        esac
}
```

Duplicitní dokumentace a změny

S několika úrovněmi dokumentace je snadné aby se neshodovali.

- Implementační na úrovni funkce
- Implementační na úrovni API
- Na rozhraní jazyků
- Vývojářská
- Uživatelská

Jak tomu předejít?

- Nechte si drobečky (This is user level function, don't forget to update user's docs)
- Dokumentujte na nejvyšší vrstvě a odkazujte se (This function is documented in API doc)
- Navažte testy na dokumentaci (Testy dokumentovaného chování)
- Kopejte kolegy do všech částí těla pokud dokumentaci ignorují

Deduplikujte kód

```
        uing8_t hight = bytes[i] >> 4;
        hexastring[2*i] = hight < 10 ? val + '0' : val + 'a';
        uing8_t low = bytes[i] & 0xf;
        hexastring[2*i + 1] = low < 10 ? low + '0' : low + 'a';
```

```
#define VAL2HEX(VAL) ((VAL) < 10 ? (VAL) + '0' : (VAL) + 'a')
        hexastring[2*i] = VAL2HEX(bytes[i] >> 4);
        hexastring[2*i + 1] = VAL2HEX(bytes[i] & 0xf);
#undef VAL2HEX
```

Zalamujeme argumenty

```
struct wait_id id = event_run_util(events, result_callback, NULL,
                &result_data, 0, NULL, -1, -1, "cp", "-f", old, new,
                (const char *)NULL);
```

```
struct wait_id id = run_util(events,
                result_callback, NULL, &result_data,
                0, NULL,
                -1, -1,
                "cp", "-f", old, new, (const char *)NULL);
```

Použijme popisnější argumenty

```
lxc launch images:alpine/3.11 myalpine -e -n default \
        -p default -s default
```

```
lxc launch images:alpine/3.11 myalpine \
        --ephemeral \
        --network default \
        --profile default \
        --storage default
```

Magické konstanty a jejich konstantnost

```
#define TOKEN_BITS 64
#define TOKEN_BYTES 8
bool verify(uint8_t *token) {
        uint64_t value = *token;
        return value % 42;
```

```
#define TOKEN_BYTES 8
#define TOKEN_BITS (TOKEN_BYTES * 8)
#if TOKEN_BYTES != 8
#error Function verify() is implemented to work only with 8 bytes tokens
#endif
bool verify(uint8_t *token) {
```

A jaké máte tipy pro zlepšení čitelnosti kódu vy?

Děkuji za pozornost.

`git.cynerd.cz`