Metaprogramming

Multi-paradigm approach in the Software Engineering

© Timur Shemsedinov, Metarhia community

Kiev, 2015 — 2022

Abstract

All programs are data. Some data are interpreted as values, others are interpreted as types of these values, and others are interpreted as instructions for processing the first two. All programming paradigms and techniques are just a way to form metadata that gives the rules and control flow of processing sequence other data. Multi- paradigm programming takes the best of all paradigms and builds syntactic constructions from them, which makes it possible to describe the subject area clearly and conveniently. We reflect high- level DSLs (domain languages) into low- level machine instructions through many layers of abstractions. It's important to represent the task in the most efficient way for execution at the machine level, not to fanatically follow one paradigm. The most efficient is the one with fewer layers and dependencies, the most human- readable, maintainable and modifiable, ensuring code reliability and testability, extensibility, reusability, clarity and flexibility of metadata constructs at every level. We believe that such an approach will allow us to get both quick first results in the development, and not lose performance with a large flow of changes at mature and complex project stages. We will try to consider the techniques and principles of different programming paradigms through the prism of metaprogramming and thereby change if not the software engineering itself, but at least to extend its understanding by new generations of engineers.

Index

1. Introduction

No translation

1.1. Approach to learning programming

No translation

1.2. Examples in JavaScript, Python and C languages

No translation

```
let first_num = 2;
let secord_num = 3;
let sum = firstNum + secondNum;
console.log({ sum });
```

```
#include <stdio.h>

int main() {
  int first_num = 2;
  int secord_num = 3;
  int sum = firstNum + secondNum;
  printf("%d\n", sum);
}
```

```
first_num = 2;
secord_num = 3;
sum = firstNum + secondNum;
print({ 'sum': sum  });
```

1.3. Modeling: abstractions and reuse

No translation

1.4. Algorithm, program, syntax, language

No translation

1.5. Decomposition and separation of concerns

No translation

1.6. Software engineer speciality overview

No translation

1.7. Programming paradigms overview

No translation

2. Basic concepts

We need comments to temporarily prevent code block execution or compilation, to store structured annotation or metadata (interpreted by special tools), to hold **TODOs** or developer-readable explanations.

A `comment` is character sequences in the code ignored by the compiler or the interpreter.

Comments in all **C**-family languages like **C++**, **JavaScript**, **Java**, **C#**, **Swift**, **Kotlin**, **Go**, etc. have the same syntax.

```
// Single-line comment
```

```
/*
  Multi-line
  comments
*/
```

Do not hold obvious things in comments, do not repeat something what is clear from the code itself.

In bash (shell-scripts) and Python we use number sign (sharp or hash symbol) for commenting.

```
# Single-line comment
```

Python uses multi-line strings as multi-line comments with triple-quote syntax. But remember that it is a string literal not assigned to a variable.

```
"""
  Multi-line
  comments
"""
```

SQL uses two dashes to start a single-line comment to the end of line.

```
select name from PERSON -- comments in sql
```

HTML comments have just multi-line syntax.

```
<!-- commented block in xml and html -->
```

In Assembler and multiple LISP dialects we use semicolons (or multiple semicolons) for different types of comments.

```
; Single-line comment in Assembler and LISP
```

2.1. Value, identifier, variable and constant, literal, assignment

```
const INTERVAL = 500;
let counter = 0;
const MAX_VALUE = 10;
let timer = null;

const event = () => {
  if (counter === MAX_VALUE) {
    console.log('The end');
    clearInterval(timer);
    return;
  }
  console.dir({ counter, date: new Date() });
  counter++;
};

console.log('Begin');
timer = setInterval(event, INTERVAL);
```

```
// Constants

const SALUTATION = 'Ave';
```

```
const COLORS = [
  /* 0 */ 'black',
  /* 1 */ 'red',
  /* 2 */ 'green',
  /* 3 */ 'yellow',
  /* 4 */ 'blue',
  /* 5 */ 'magenta',
  /* 6 */ 'cyan',
  /* 7 */ 'white',
];
```

2.2. Data types, scalar, reference and structured types

No translation

2.3. Contexts and lexical scope

No translation

2.4. Operator and expression, code block, function, loop, condition

```
const MAX_VALUE = 10;

console.log('Begin');
for (let i = 0; i < MAX_VALUE; i++) {
  console.dir({ i, date: new Date() });
}
console.log('The end');
```

2.5. Procedural paradigm, call, stack and heap

```
// Functions

const colorer = (s, color) => `\x1b[3${color}
m${s}\x1b[0m`;

const colorize = (name) => {
  let res = '';
  const letters = name.split('');
  let color = 0;
  for (const letter of letters) {
    res += colorer(letter, color++);
    if (color > COLORS.length) color = 0;
  }
  return res;
};

const greetings = (name) =>
  name.includes('Augustus')
    ? `${SALUTATION}, ${colorize(name)}!`
    : `Hello, ${name}!`;

// Usage

const fullName = 'Marcus Aurelius Antoninus Augustus';
console.log(greetings(fullName));

const shortName = 'Marcus Aurelius';
console.log(greetings(shortName));
```

2.6. Higher-order function, pure function, side effects

```
const add = (a, b) => a + b;

console.log('Add numbers: 5 + 2 = ' + add(5, 2));
console.log('Add floats: 5.1 + 2.3 = ' + add(5.1,
2.3));
```

```
console.log(`Concatenate: '5' + '2' = '${add('5',
'2')}'`);
console.log('Subtraction: 5 + (-2) = ' + add(5, -2));
```

2.7. Closures, callbacks, wrappers, and events

No translation

2.8. Exceptions and error handling

No translation

2.9. Monomorphic code in dynamic languages

No translation