Software Improvement Group

A benchmark based approach to determine language verbosity

Hans Kuijpers & Lodewijk Bergmans

Software Improvement Group

Three different implementations of the same algorithm to determine prime numbers

Algorithm: 'Sieve Of Eratosthenes', generates prime numbers

Haskell

```
primes = sieve [ 2.. ]
  where
    sieve (p:x) = p : sieve [ n | n <- x, n `mod` p > 0 ]
```

Python

```
def eratosthenes(n):
  sieve = [ True for i in range(n+1) ]
  def markOff(pv):
    for i in range(pv+pv, n+1, pv):
      sieve[i] = False
  markOff(2)
  for i in range(3, n+1):
    if sieve[i]:
      markOff(i)
  return [ i for i in range(1, n+1) if sieve[i] ]
```

- How can we compare the volume of these programs?
- How much effort has been spent on developing these?
  - How much on design? On coding?
- Can we estimate the typical effort per Line of Code?

JavaScript

```
function Eratosthenes(element, top) {
  var all = new Uint8Array(new ArrayBuffer(top));
  var idx = 0;
  var prime = 3;
  var x, j;

  element.innerHTML = "1 ";
  while(prime <= top) {
    var flag = true;
    for(x = 0; x < top; x++) {
        if(all[x] == prime) {
            flag = false;
            break;
        }
    }
    if(flag) {
        element.innerHTML += prime + " ";
        j = prime;
        while(j <= (top / prime)) {
            all[idx++] = prime * j;
            j += 1;
        }
    }
    prime += 2;
  }
  element.innerHTML += "<br>";
  return;
}
```

Table of Contents

Who is Who

Hans Kuijpers

8 years at SIG

Managing Consultant

Certified Scope Manager, Professional Scrum Master I
FPA: Nesma, IFPUG, Cosmic, Use Case Points

Lodewijk Bergmans

8 years at SIG

Sr. Researcher

Areas of expertise: software metrics, measuring development practices, software economics: fact-based decision support for software developers, architects and management.

Build quality drives TCO and functional quality of software

Software functionality

The majority of software development work is not visible, SIG can make this transparent:

beyond ISO

Table of Contents

Motivation

- **Important requirement: assess total volume of a system with multiple programming languages**
  - Must be able to aggregate parts of a system in different languages
  - Must yield a comparable value among systems (e.g. for benchmarking)

- Previously, the ‘SPR table’ was used at SIG as a basis for translating LOC to volume (effort needed)
  - i.e. ‘language (productivity) factors’
  - In our maintainability model, this is used to compare/aggregate/weigh code in different languages ‘fairly’
  - In the SIG Cost Estimation Model, this is used to translate coding efforts (creating/modifying) to financial costs (through efforts in terms of person years (PY)

- But SPR now has several issues:
  - Contains entries that are not intuitive
  - Outdated (>10 years since last update):
    - Languages and libraries have evolved
    - Does not include new languages

Goals & context

Goal: Create a new set of factors that can be used to:

1. Compare and aggregate the size of applications in different languages
2. Calculate the ‘average’ effort needed
    - To write 1000 Lines of Code in a specific programming language (our ‘Volume’ measurement)

Considerations

- Comparing size of programs should be related to the technical effort[1] involved in creating 1000 Lines of Code
- Approach must be applicable to all programming languages we encounter
- Use a fact- or measurement-based approach, e.g. using the SIG code warehouse or other data sources
- Preferably not depending on external parties
- Experience has shown that it is *really* difficult to find consistent, trustworthy, and *comparable* effort data from actual projects, especially for new, rare and custom programming languages.

[1] Technical effort is related to all the work a developer does; technical design, coding and testing.

Table of Contents

Implementations of the same algorithm in different programming languages

Two 'Eratosthenes' code samples revisited:

- Haskell is a functional language, very much suited for compact expression of algorithms:

Haskell

```
primes = sieve [ 2.. ]
  where
    sieve (p:x) = p : sieve [ n | n <- x, n `mod` p > 0 ]
```

- Python is a bit more generic OO language:

Python

```
def eratosthenes(n):
  sieve = [ True for i in range(n+1) ]
  def markOff(pv):
    for i in range(pv+pv, n+1, pv):
      sieve[i] = False
  markOff(2)
  for i in range(3, n+1):
    if sieve[i]:
      markOff(i)
  return [ i for i in range(1, n+1) if sieve[i] ]
```

We observe that one language is more **verbose** than the other

- Or, in reverse: the other language is more **expressive**

The **SPR table** defines for each language its **language level**:

- This has an inverse relation to the number of source statements to implement 1 Function Point
- In other words: how much/little code is needed to express a certain amount of functionality

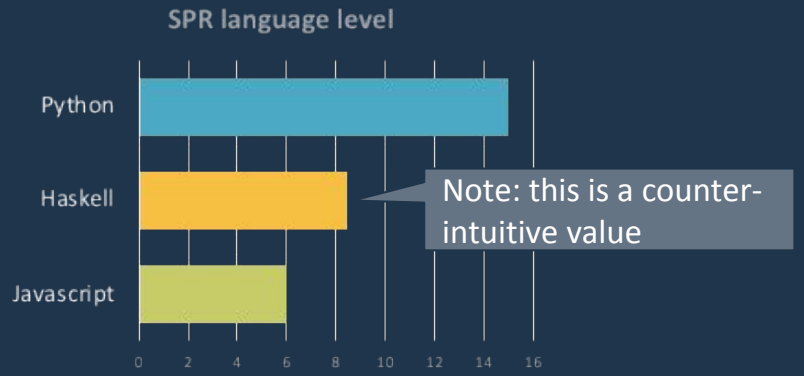


Table of Contents

Our premise: a program is the encoding of information in source code

information → Source code

Domain knowledge

requirements

Design idiom & patterns

programming language

```
Column
position 1-------10--------20--------30--------40--------50--------60--------70


         IDENTIFICATION DIVISION.
          PROGRAM-ID. SAMPLE5.
         *>  THIS SAMPLE PROGRAM IS IN FREE FORMAT. THE PROGRAM MUST BE
         *>  COMPILED WITH THE SRF COMPILER OPTION. THE SRF COMPILER OPTION
         *>  SPECIFIES THE SOURCE FORMAT TYPE. SRF(FREE,FREE) TELLS THE
         *>  COMPILER THAT THE SOURCE PROGRAM AND COPYBOOKS ARE IN FREE FORMAT
         ENVIRONMENT DIVISION.
          CONFIGURATION SECTION.
           SPECIAL-NAMES.
             SYSERR IS MESSAGE-DEVICE.
             :
         DATA DIVISION.
          FILE SECTION.
          FD  MASTER-FILE.
          01  MASTER-RECORD.
              02  GOODS-RECORD.
                  03  GOODS-CODE       PIC X(4).
                  03  GOODS-NAME       PIC X(30).
                  03  GOODS-PRICE      PIC 9(4)  BINARY.
             :
         PROCEDURE DIVISION USING PARAMETER.
         *>  (1) DETERMINE WORK-FILE NAME
             IF PARAMETER-LEN = 0
               DISPLAY "NOT SPECIFIED PARAMETER."-
               "PLEASE SPECIFY PARAMETER."
                 UPON MESSAGE-DEVICE
               GO TO TERM-PROC.
             :
          TERM-PROC.
             EXIT PROGRAM.
         END PROGRAM SAMPLE5.
```

So programming is about distilling the necessary information and expressing it using source code.

Our approach to measure verbosity/expressiveness

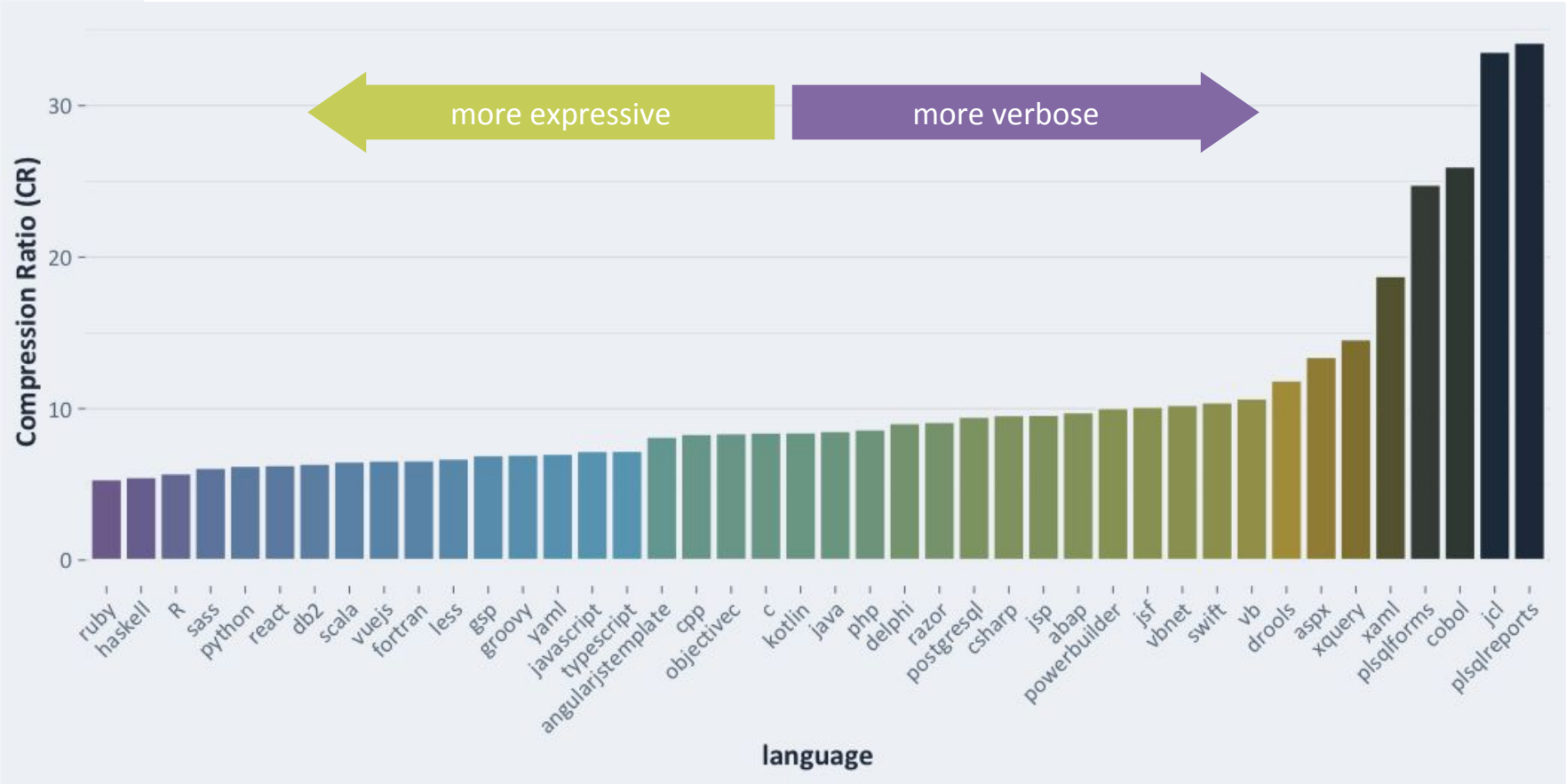
We adopt a *similar* approach to SPR:

Measure how much/little code is needed to express a certain amount of *information*

- **Advantage**: this also applies to code that is not directly implementing functionality:
  - Non-functionals such as performance, security, reliability.
    - This can easily be 30% of the code [1]
  - But also the code for structuring software (e.g. all declarations of methods, classes, interfaces, inheritance, ..)!
- **How** can we measure the amount of information? → the theory of **Kolmogorov complexity**
  - "The amount of information in a document equals the size of the shortest possible program that can produce the document"
  - A practical approximation: take the length of the compressed document
  - We use the term Information Point (IP) to designate a single unit of information
- **Application**: calculate the Compression Ratio (CR) of programs in a language: CR = code_size / compressed_size
  - The average* compression ratio for a language indicates the average amount of code per unit of information (IP)
  - We built a benchmark of average compression ratio's based on large amount of scoped industrial systems analyzed by SIG.

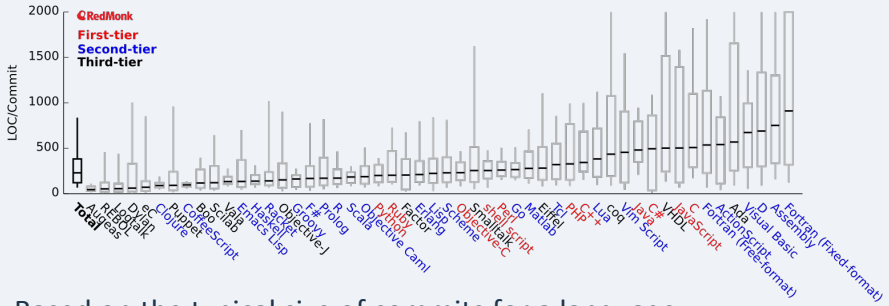
Table of Contents

Compression ratios for subset of languages



Preliminary validation: comparing with gearing factors, and Redmonk language ranking

Comparison to Redmonk 'language ranking by expressiveness'[1]



Based on the typical size of commits for a language:

- Assumption is that 1 commit ~ 1 functionality/behavior
- We only tested against the ranking, not actual values

The correlation between Redmonk ranking and Compression Ratios (CR) is strong (Spearman correlation =0.75)

In other words: our notion of language verbosity corresponds closely to the redmonk notion of language expressiveness

Comparison to language gearing factors of SPR[2] and QSM[3]

These are expressed as SS/FP: source statements per FP

We found NO significant correlations between CR and SPR/QSM. SS/FP is not exactly the same as verbosity/CR:

- E.g. Reusing functionality from libraries provides extra FPs but does need little coding effort
- The CR-based approach counts non-functional code
- SPR is 15+ years old; (usage of) languages evolved

[1] D. Berkholz, "Programming languages ranked by expressiveness", redmonk, 2013: *https://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness/*

[2] SPR, "SPR programming languages table, PLT2007c", 2007

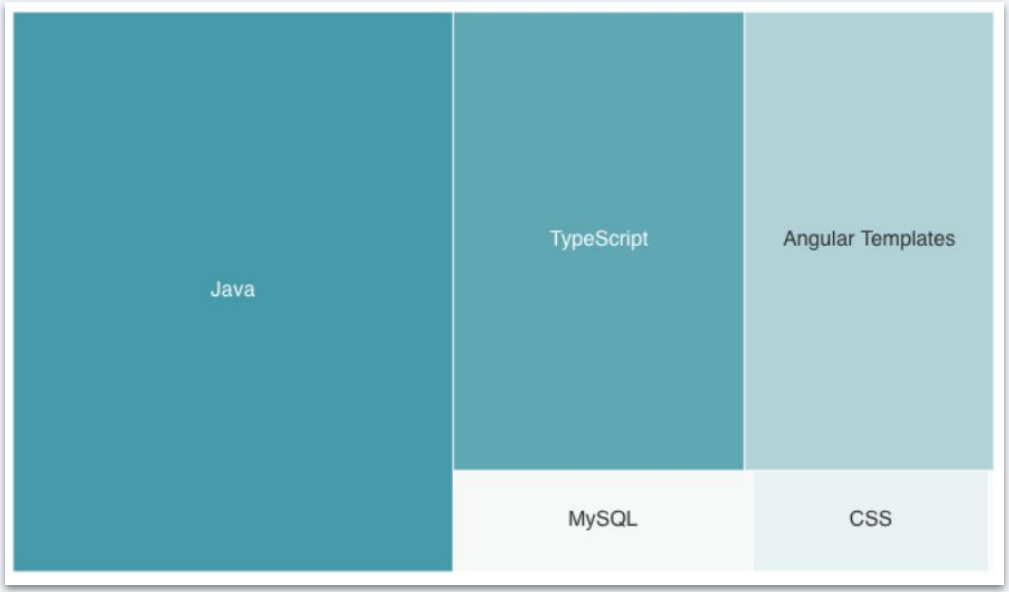
[3] QSM, "Function Point Languages Table, version 5.0", *https://www.qsm.com/resources/function-point-languages-table*

Table of Contents

Volume normalization at system level

For comparing and aggregating the volume of a system implemented with multiple languages

- Lines of Code is less representative of the amount of information and (intellectual) effort spent on the different parts

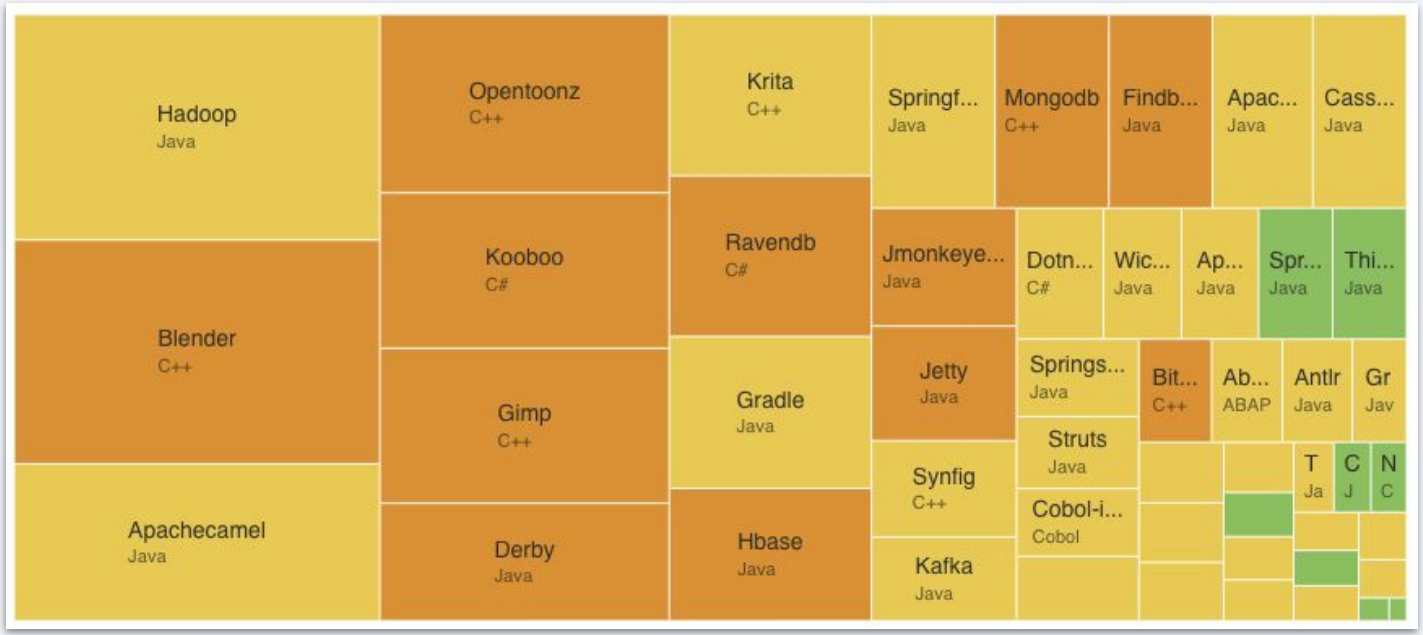


[This is a screenshot from the Sigrid® software assurance platform]

Volume normalization at portfolio level

For comparing the volume of systems in a portfolio; some may be implemented in totally different technologies

- This overview shows the systems in a portfolio, annotated with the *main language* per system



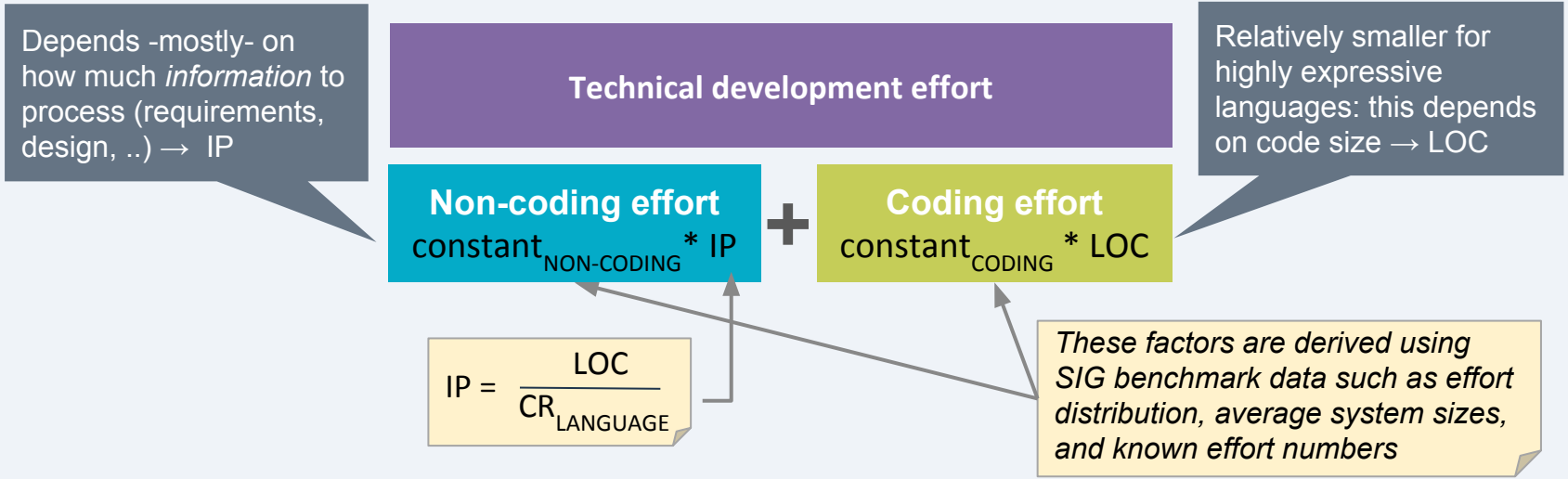
[This is a screenshot from the Sigrid® software assurance platform]

Base for cost (effort) estimations

Code-based effort estimation: effort is influenced by

- the amount of code (LOC)
- the type of language

Other (human) factors are encapsulated in the constants derived from benchmark data



Conclusion

What did we achieve with compression ratio (CR) based volume normalization?

- **CR is an objective measure of language verbosity**
  - Applicable for all text-based languages, also for new, custom and domain specific languages
  - Based on a benchmark: how languages are used in practice
  - Can be largely automated (and calibrated periodically)

- **CR allows comparing the size of source code in different languages in a consistent way**
  - You don't need FP countings, or access to the hour administration
    - But does require availability of source code
    - Does support '**real-time' observation of progress**
  - The CR-based approach is representative for all the behavior a developer implements
    - Functional behavior is not considered separately

- **CR can be used as the basis of a code-based estimation of development work**

Questions?

GETTING SOFTWARE RIGHT FOR A HEALTHIER DIGITAL WORLD