# THE BEAUTY AND THE BEAST: SEPARATING DESIGN FROM ALGORITHMS

Dmitrijs Zaparanuks and Matthias Hauswirth

# Modularization of a program


- Do I extract this code into a separate method?
- Do I implement this traversal iteratively or recursively?
- Do I use a visitor pattern or do I place the computation in the structure itself? …

A good modularization ease understandability and maintainability

# Modularization of a program

```
private static int fac(int n) {
    int f=1;
    for (int i=1; i<=n; i++) {
        int p = 0;
        for (int j=1; j<=i; j++) {
            p=p+f;
        }
        f=p;
    }
    return f ;
}
```

```
private static int fac(int n) {
    int f=1;
    for (int i=1; i<=n; i++) {
        f=mul(f, i);
    }
    return f ;
}
private static int mul(int a, int b) {
    int p = 0;
    for (int j=1; j<=a; j++) {
        p=p+b;
    }
    return p ;
}
```

# Modularization of a program

```
private static int fac(int n) {
        int f=1;
        for (int i=1; lessOrEqual(i, n); i=addOne(i)) {f=mul(f, i);}
         return f ;
}
private static int mul(int a, int b){
        int p = 0;
        for (int j=1; lessOrEqual(j, a); j=addOne(j)) {p=add(p, b);}
        return p;
}
private static int add(int a, int b) { return a+b; }
private static boolean lessOrEqual ( int a , int b) { return a<=b;}
private static int addOne(int a) { return add(a, 1); }
```

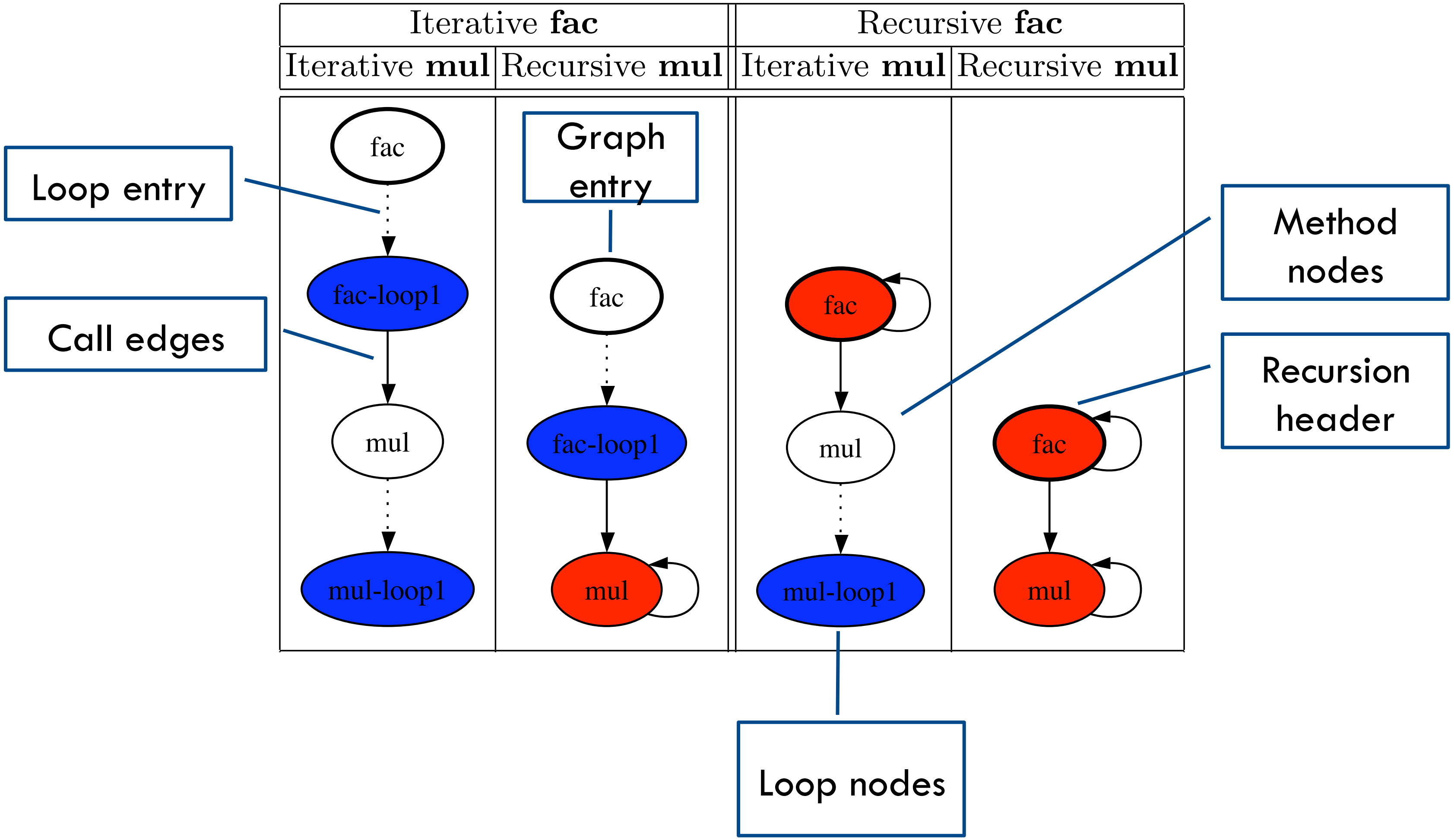
# Aims of the paper

- Propose a metric to quantify the algorithmically essential parts of a program
  - Localizable
  - Intuitive
  - Stable
  - Language independent
- Consider loops and recursive calls
- The metric is computed using three different representations
  - Call graph
  - Control-flow
  - Loop call graphs

# Overall approach

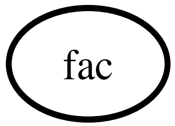
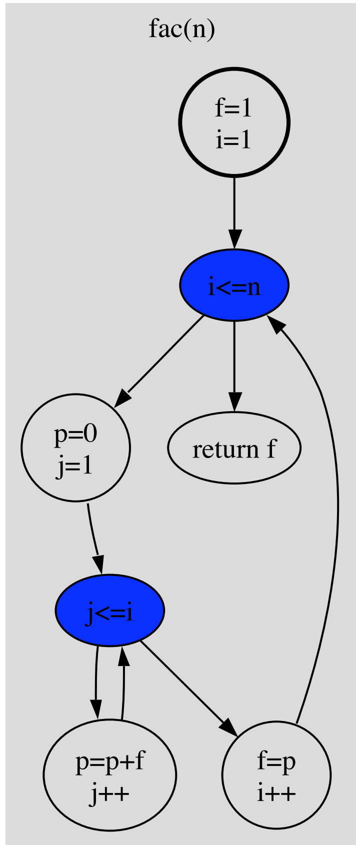
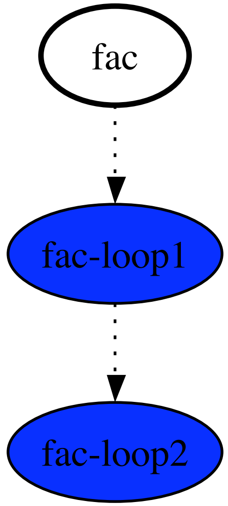
1. Build a control-flow graph
2. Identify loop forests in control-flow graphs
3. Build call graph
4. Identify recursion forests in call graph
5. Combine loop forests & call graph into loop call graph
6. Compute metrics

# Loop call graph



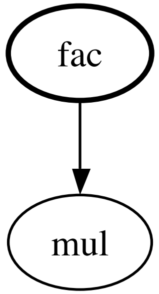
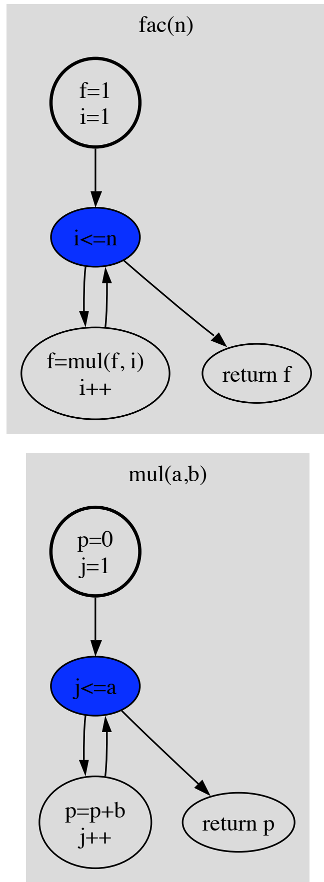
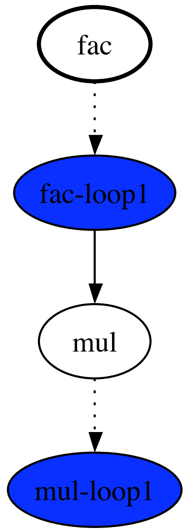
# Example 1

```
private static int fac(int n) {
    int f=1;
    for (int i=1; i<=n; i++) {
        int p = 0;
        for (int j=1; j<=i; j++) {
            p=p+f;
        }
        f=p;
    }
    return f ;
}
```

| Call graph | Control-flow graph | Loop call graph |
|---|---|---|
|  |  |  |

# Example 2

```
private static int fac(int n) {
        int f=1;
        for (int i=1; i<=n; i++) {
                f=mul(f, i);
        }
        return f ;
}
private static int mul(int a, int b) {
        int p = 0;
        for (int j=1; j<=a; j++) {
                p=p+b;
        }
        return p ;
}
```

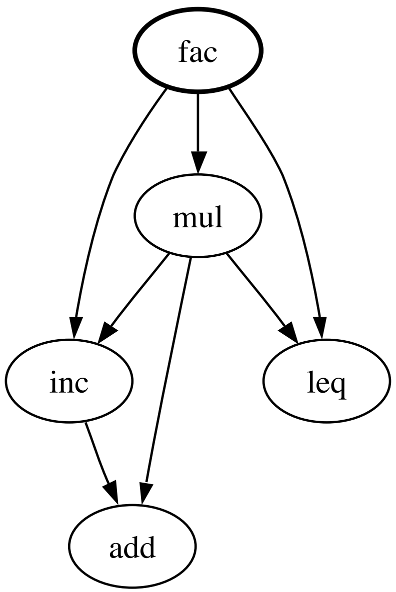
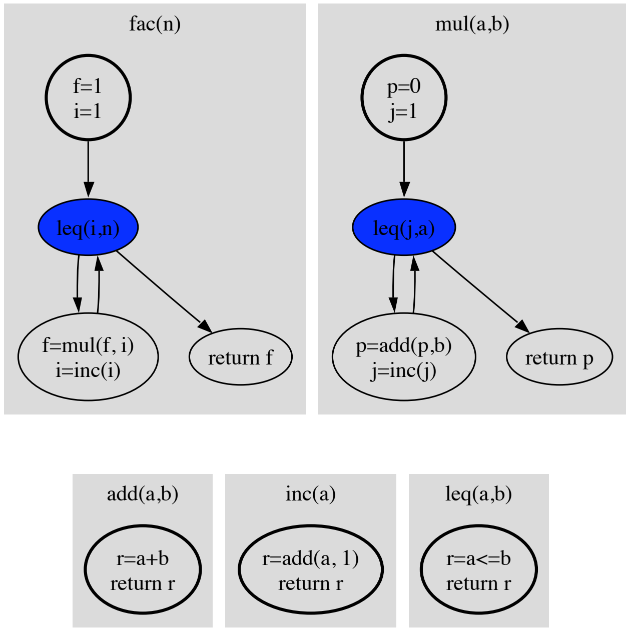
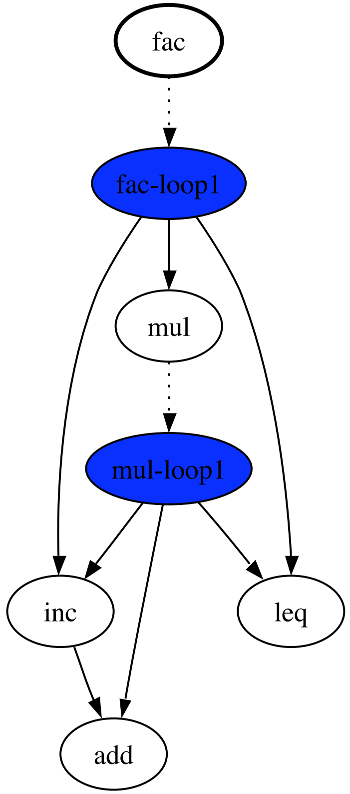
| Call graph | Control-flow graph | Loop call graph |
| --- | --- | --- |
|   |   |   |

# Example 3

```
private static int fac(int n) {
    int f=1;
    for (int i=1; lessOrEqual(i, n); i=addOne(i)) {f=mul(f, i);}
     return f ;
}
private static int mul(int a, int b){
    int p = 0;
    for (int j=1; lessOrEqual(j, a); j=addOne(j)) {p=add(p, b);}
    return p;
}

private static int add(int a, int b) { return a+b; }
private static boolean lessOrEqual ( int a , int b) { return a<=b;}
private static int addOne(int a) { return add(a, 1); }
```

# Example 3

| Call graph | Control-flow graph | Loop call graph |
|---|---|---|
|  |  |  |

# Metrics

NN = |non-recursive method nodes|

NR = |recursive method nodes|

NL = |loop nodes|

E = NL + NR

$$\text{recursiveness} = \frac{NR}{NN + NR}$$

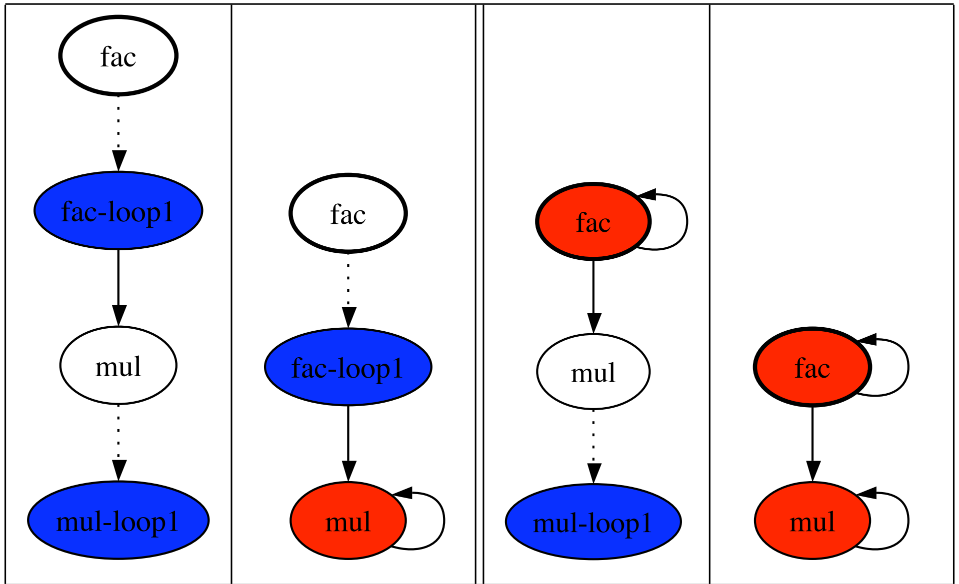
$$\text{loopyness} = \frac{NL}{NN + NR}$$

$$e = \frac{E}{NN + NR}$$

‘e’ for previous examples:

- Example 1 = 2
- Example 2 = 1
- Example 3 = 0.4

| Iterative **fac** | | Recursive **fac** | |
|---|---|---|---|
| Iterative **mul** | Recursive **mul** | Iterative **mul** | Recursive **mul** |



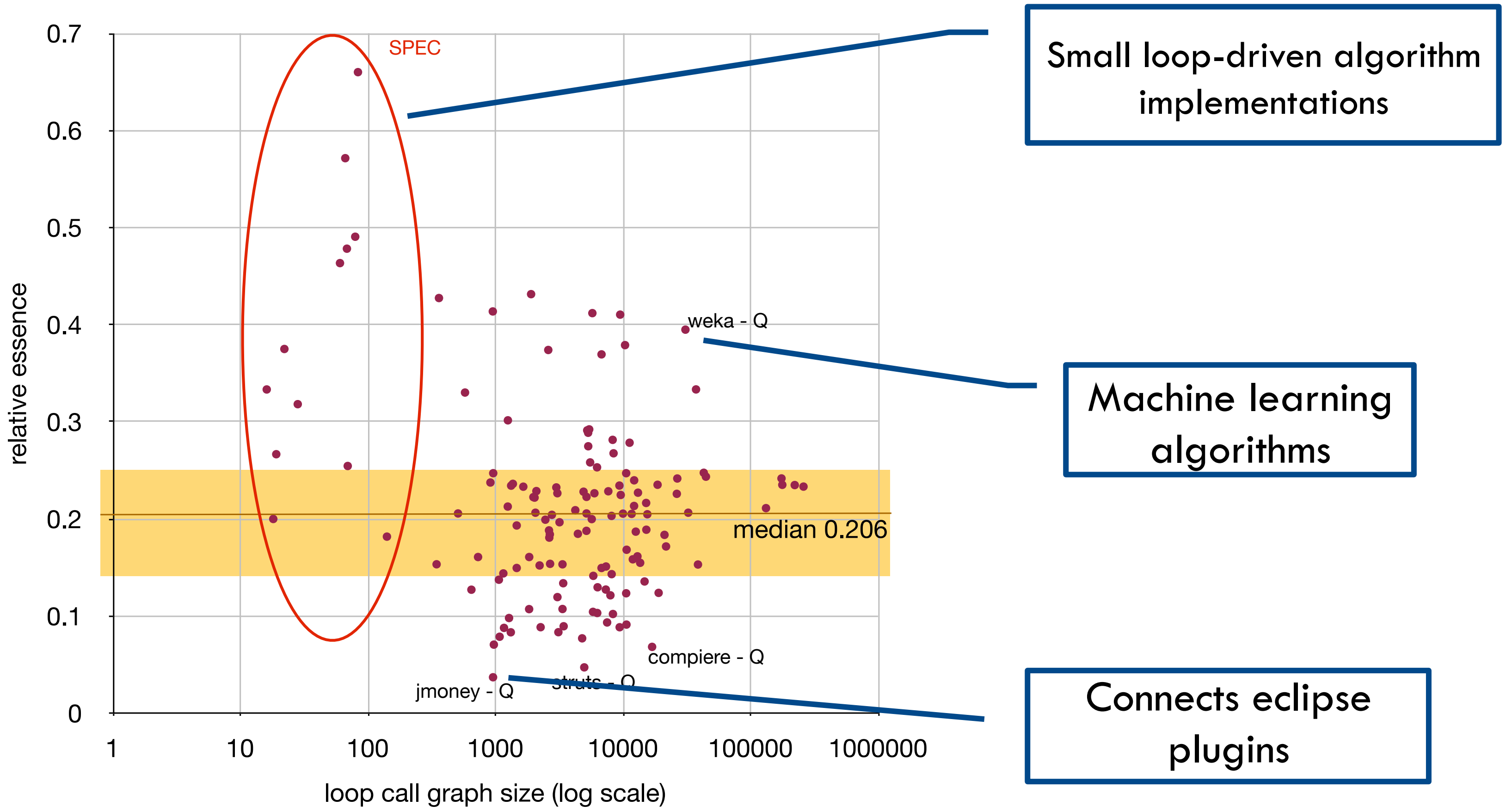

| Iterative **mul** | Recursive **mul** | Iterative **mul** | Recursive **mul** |
|---|---|---|---|
| $N_N = 2$ | $N_N = 1$ | $N_N = 1$ | $N_N = 0$ |
| $N_R = 0$ | $N_R = 1$ | $N_R = 1$ | $N_R = 2$ |
| $N_L = 2$ | $N_L = 1$ | $N_L = 1$ | $N_L = 0$ |
| $E = 2$ | $E = 2$ | $E = 2$ | $E = 2$ |
| $e = \frac{2}{2+0} = 1$ | $e = \frac{2}{1+1} = 1$ | $e = \frac{2}{1+1} = 1$ | $e = \frac{2}{0+2} = 1$ |

# Metrics

- E
  - It is not affected by the degree of indirection
  - It does not depend on the implementation (recursive or iterative)

- NN is correlated with the level of indirection

# Essence

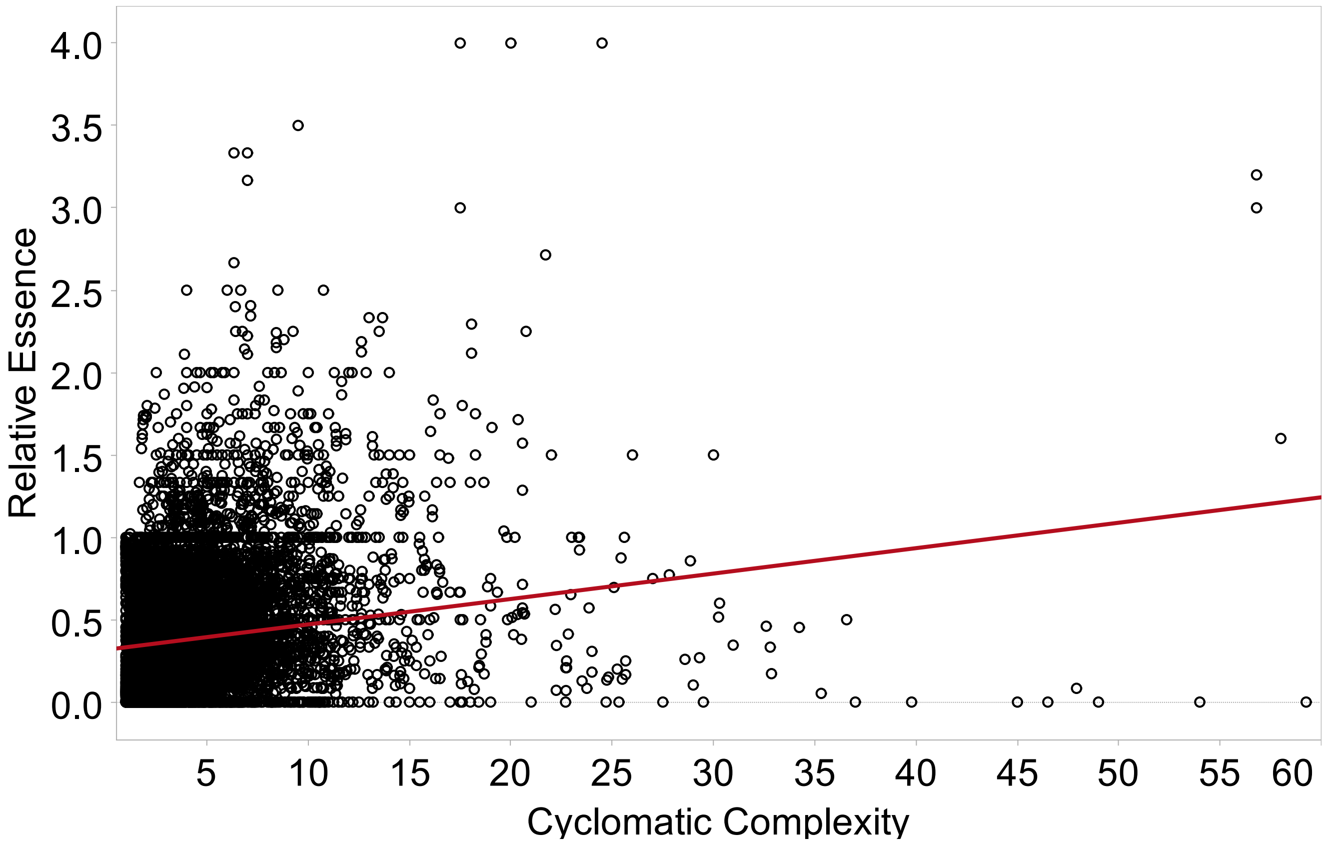


0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
relative essence
1
10
100
1000
10000
100000
1000000
loop call graph size (log scale)
SPEC
weka - Q
compiere - Q
jmoney - Q
struts - Q
median 0.206
Small loop-driven algorithm implementations
Machine learning algorithms
Connects eclipse plugins

# Essence & design

- Code smells
  - In some cases, the essence is too low but the amount of indirection too high
- Refactoring
- Design patterns
  - This technique successfully identifies that the visitor pattern introduces an extra level indirection for each method

# Cyclomatic complexity



# Usage scenarios

- Deviation from reference
  - Compare *essence* with a reference value of a system of high design quality
- Problem localization
  - Generated graphs can help to spot nodes with high or low essence
- Refactoring recommendation
  - Guide modularization of a system
- Quality and process attribute prediction
  - It might be possible to predict process attributes, such as error rates or times to fix an error

# Conclusions

- The presented metric (relative essence) provides hints on which parts of the system to *remove*, and where to *add* extra indirections
- Not all recursion and loops are necessarily required for solving the problem the program needs to solve
- This metric relates with design patterns, code smells and refactoring
- None of the existing metrics correlates with relative essence

TARTU ÜLIKOOL · UNIVERSITAS TARTUENSIS · 1632