The Beauty and the Beast: Separating Design from Algorithm

Dmitrijs Zaparanuks and Matthias Hauswirth

University of Lugano
zaparand@usi.ch
Matthias.Hauswirth@unisi.ch

Abstract. We present an approach that partitions a software system into its algorithmically essential parts and the parts that manifest its design. Our approach is inspired by the notion of an algorithm and its asymptotic complexity. However, we do not propose a metric for measuring asymptotic complexity (efficiency). Instead, we use the one aspect of algorithms that drives up their asymptotic complexity – repetition, in the form of loops and recursions – to determine the algorithmically essential parts of a software system. Those parts of a system that are *not* algorithmically essential represent aspects of the design. A large fraction of inessential parts is indicative of "overdesign", where a small fraction indicates a lack of modularization. We present a metric, *relative essence*, to quantify the fraction of the program that is algorithmically essential. We evaluate our approach by studying the algorithmic essence of a large corpus of software system, and by comparing the measured essence to an intuitive view of design "overhead".

1 Introduction

Given today's large software systems, consisting of tens or hundreds of thousands of classes, wouldn't it be nice to be able to automatically distinguish between their essential and non-essential parts? More specifically, wouldn't it be nice to be able to quantify the amount of algorithmically essential code and the amount of code that primarily serves design? And wouldn't it be nice to have a tool that automatically locates the essential code? In this paper we present an approach towards this goal.

A software system implements a set of interacting algorithms to achieve a function. As software systems have gotten more complex, software developers have tried to find techniques to combine and implement the algorithms in such a way as to not only achieve efficiency of the running code, but also structure in the software code. The structure is generally referred to as design. There has been a lot of effort to develop effective design methods. Many of them are based on Parnas' fundamental principle of "information hiding" [13]. The expectation is that good design will impose an overhead (often in the form of extra indirection) during execution, but that it will provide benefits in terms of understandability and maintainability of the software.

In his seminal "No Silver Bullet – Essence and Accident in Software Engineering" [3], Brooks provides an *intensional* definition of essential complexity ("complex conceptual structures that compose the abstract software entity") and accidental complexity ("*representation* of these abstract entities in programming languages"). His definition of essential complexity refers to the algorithms to be implemented in a system. Any implementation of such algorithms in a given programming language will necessarily entail design decisions such as: Do I extract this code into a separate method? Do I implement this traversal iteratively or recursively? Do I use polymorphism or a switch statement? Do I introduce a facade? Do I use delegation or inheritance? Do I use a visitor pattern or do I place the computation in the structure itself? All of these design decisions affect the resulting code. None of them (should) affect the essential algorithms implemented by the system.

Our goal is to identify the essential parts of a software system by analyzing its code. To automate this task, we have to provide an *operational* definition of essence. We do this by focusing on *repetitions*. Repetitions are essential for the algorithmic complexity of a program. If there are no repetitions, the asymptotic computational complexity is constant, or $O(1)$. While *conditionals* affect computational complexity, adding a conditional statement to a structured program can only reduce its computational complexity. Thus, to keep our analysis tractable, we focus on the most important algorithmic aspect in code: repetitions.

In most programming languages, there are two forms of repetition: loops and recursion. *Loops* correspond to cycles in the *control flow graph* of a method, while *recursions* correspond to cycles in the *call graph* of a program. Our analysis thus identifies the loops in all methods and it determines the recursive cycles in the program's call graph. The duality between loops and recursions allows us to use the same algorithm for both analyses.

More precisely, our analysis creates *forests of nested cycles* for each of these graphs. For control-flow graphs, such a forest is known as a "loop nest tree". A cycle is a set of nodes so that each node in the set is reachable from each other node in the set. Moreover, each cycle has a set of *header nodes*, through which execution can enter the cycle. In control-flow graphs directly compiled from structured programs, loops contain only a single header node. However, loops in control flow graphs of unstructured programs, and recursive cycles in the call graph, can have multiple headers.

Our approach to determine the algorithmically essential parts of a program identifies all header nodes of iterative and recursive cycles. Nodes in cycles that are not headers are not deemed algorithmically essential. This relatively straightforward approach allows us to identify the (algorithmically inessential) methods introduced by design decisions such as the use of collections, facades, iterators, encapsulation, or visitors.

The result of our analysis is a representation we call the *loop call graph*, in which we highlight essential parts of the program. Moreover, based on that representation, we compute metrics that quantify the algorithmic essence of a system (or subsystem) as well as its design "overhead".

1.1 Rationale

Our approach is based on the following design rationale:

Localizable. We do not just want to have a global overall metric measuring essence, but we want an approach that explicitly identifies essential and inessential parts in programs.

Intuitive. The program parts identified as essential need to correspond to a programmer's intuition of what is essential in the underlying algorithm.

Stable. When analyzing two implementations of the same algorithm, the essential parts in both implementations should correspond to each other.

Language-independent. Our approach should not depend on a specific programming language. Two equivalent programs written in different languages should result in equivalent essential parts.

The remainder of this paper is structured as follows. Section 2 presents a motivating example. Section 3 explains our approach and analysis. Section 4 discusses the tool that implements our approach for Java programs. Section 5 uses our approach to characterize a large body of Java programs. Section 6 discusses the relationship between essence and design. Section 7 connects essence to related work. Section 8 discusses usage scenarios for our metric and limitations of our approach, and Section 9 concludes.

2 Motivation

Consider the three implementations[1] of the factorial function in **Listings 1.1, 1.2**, and **1.3**. They vary in their amount of indirection: Listing 1.1 expresses the computation as two nested for loops, Listing 1.2 factors out the multiplication into a separate method, and Listing 1.3 introduces separate methods for even more basic computational steps. However, all three implementations perform the same essential computation.

Many developers probably would consider Listing 1.2 as the best of the three designs. We believe this is because neither the call graph nor the control flow graph is excessively big. **Figure 1** shows those two graphs (left and center column) for each of the three listings (top to bottom). Listing 1.1 has a relatively large control-flow graph, while Listing 1.3 has a similarly large call graph. Unfortunately, we cannot use this balance between call graph and control-flow graph size as guidance for a design metric: the size of the call graph is unbounded because it grows with the size of the program.

However, if we identify the repetitions (in these iterative programs, the loops), we can use them to measure the amount of algorithmically essential code versus the amount of design-related code. We do this in our new representation, the

[1] For the sake of a simple example, assume the programming language does not have a multiplication operator, which requires the developer to implement multiplication explicitly.

Listing 1.1. Too little indirection

```
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;
}
```

Listing 1.2. Enough indirection

```
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;
}
```

loop call graph (right-most column in Figure 1), which adds loop nodes into the call graph. Having a single representation allows us to combine information about recursion with information about loops.

Figure 1 shows that each of the three implementations contains two loops. The loop call graph shows that, at runtime, the multiplication loop will be nested inside the factorial loop, no matter which implementation we use. Moreover, it shows that the number of inessential nodes in the loop call graphs differs significantly. The *relative essence*, or the number of essential nodes divided by the number of method nodes, changes from 2/1 to 2/2 to 2/5. As these values show, a high relative essence is not necessarily positive: it may indicate the absence of any modularization. However, an essence close to 0 is an indication of “overdesign”, or too much indirection.

3 Approach

Our approach to identifying essential and accidental parts of computation is to look for repetition in the computation. This repetition manifests itself either as loops (cycles in control-flow graphs) or recursions (cycles in call graphs).

Listing 1.3. Too much indirection

```
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);
}
```

To reason about repetition overall, we have to combine information from both of these graphs. In theory, we could build a whole-program control-flow graph. In this way, call graph cycles would result in cycles in the whole-program control-flow graph. However, given the size of modern software, the size of the resulting whole-program control-flow graph would grow too big for efficient analysis.

Instead of combining call graph and control-flow graph, we propose a multi-level analysis. We analyze each control-flow graph individually to find loops, and we analyze the call graph to find recursions. Each loop and each recursion header represents a computationally essential part of the program, while all other call graph nodes represent accidental parts. Then we combine the results to present the essential and accidental parts of the program and to measure its essence.

3.1 Overview

Our overall approach can be summarized as follows:

1. Build control-flow graphs
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

Building control-flow graphs. Control-flow graphs of programs written in languages supporting exception handling often contain a substantial number

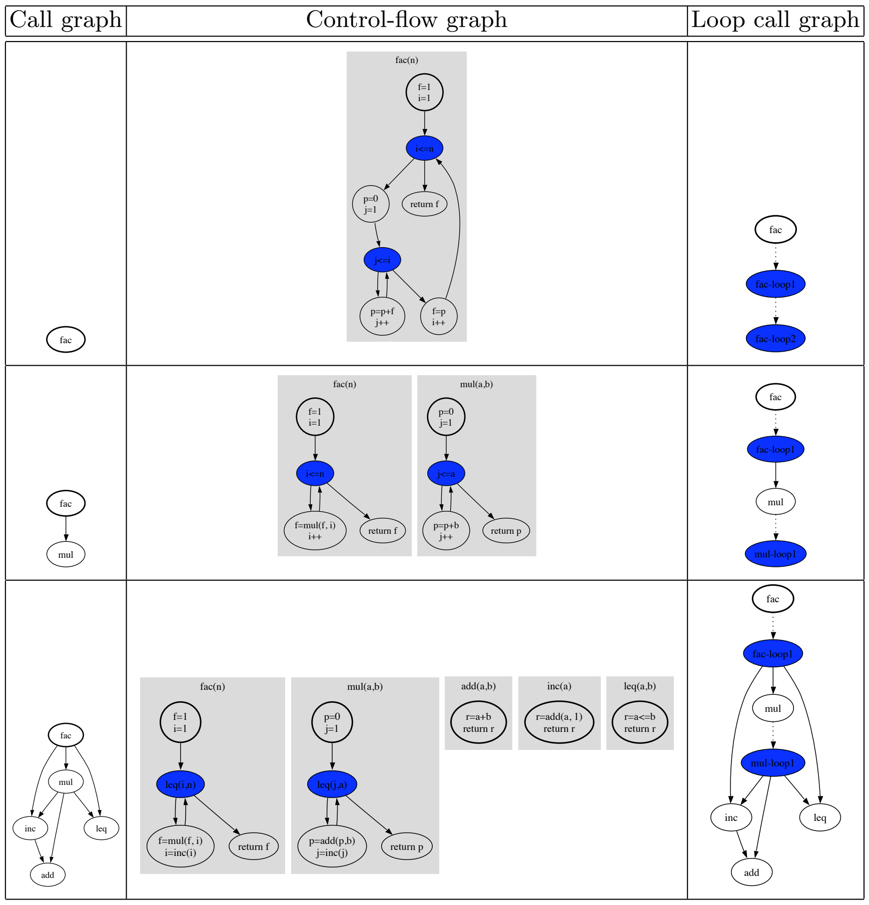


Fig. 1. Call graph, control-flow graph, and loop call graph

of exception edges. We expect the control-flow graphs to include those edges, so that we are able to analyze the whole program, including all its exception handlers.

Identify loop forests in control flow graphs. We use an approach to loop detection that works on arbitrary (even un-structured) control flow graphs. Section 3.2 describes that approach in detail.

Build call graph. We expect the call graph to include all feasible call edges (e.g. including those due to polymorphic invocations and calls through function pointers). Naively, this can be satisfied by adding edges between each pair of methods. However, such an extremely conservative approach would lead to large numbers of cycles in the call graph and would greatly overstate the amount of recursion in the program.

For strongly-typed languages with polymorphic method calls, efficient analyses such as class-hierarchy or rapid type analysis [1] are able to eliminate most infeasible call edges. For weakly-typed languages, more expensive flow-sensitive analyses or pointer analyses are necessary to get a reasonably precise call graph.

Our recursion detection approach requires one node of the call graph to be denoted as the *entry* node, and it requires all other nodes to be reachable from that node. When analyzing entire programs, this usually corresponds to the main method. When analyzing libraries, we introduce an artificial entry node that points to all public API methods provided by the library.

Sometimes we are interested in analyzing an application but to exclude (some of) the libraries. In that case we do not create any nodes for the excluded library methods. Call sites in the application that point to excluded library methods do not lead to any call edges. Frameworks, like libraries, are called by the application, but they also call back into the application. When analyzing frameworks in isolation, we create neither of these (incoming or outgoing) call edges, and thus we may miss recursive cycles that cross between application and framework[2].

Identify recursion forests in call graph. Skiena [16] proposed to identify recursion in call graphs with the purpose of optimizing non-recursive code. He labels all nodes that are involved in a recursion cycle. However, not all methods involved in a recursion cycle are essential for the computational complexity of the program. Analog to loop analysis in control-flow graphs, we want to identify the "headers" of recursion cycles, and we want to properly handle nested recursion cycles. Doing so requires us to determine recursion forests, forests consisting of trees of nested recursion cycles ("loops"). The duality between control-flow graphs and call graphs allows us to use the same approach we use for detecting loop forests in control-flow graphs.

Combine loop forests and call graph into loop call graph. To present the results of our analysis in a single structure, we combine the loops and the call graph into a *loop call graph*. A loop call graph contains two kinds of nodes: method nodes represent methods and loop nodes represent loops. The graph contains two kinds of edges: call edges represent method calls and loop entry edges represent loop entries. Call edges point from a method or a loop to a method (the callee). Loop entry edges point from a method or a loop to a loop. Loop entry edges originating from a loop represent loop nesting (entry of an inner loop from an outer loop). Call edges only originate in a method if the call site is not located in a loop (otherwise they originate in the innermost loop containing the call site).

Figure 2 shows the loop call graphs for four variations of the "best" factorial example in Listing 1.2. The left-most graph corresponds to the original example, the other graphs replace one or both methods (`fac` or `mul`) with recursive implementations. Loop entry edges are rendered with dotted lines, while call edges are solid. Loop nodes are filled in blue. Method nodes that are recursion headers are filled in red. The method node that corresponds to the graph entry (`fac`) has a bold outline.

[2] We have observed such cycles in frameworks, such as GUI toolkits, that contain recursively invokable event dispatch methods.

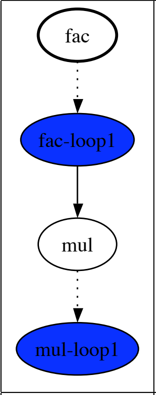
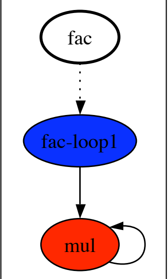
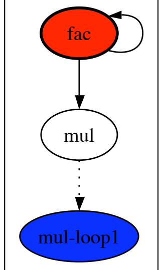
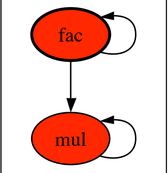
Iterative **fac**		Recursive **fac**	
Iterative **mul**	Recursive **mul**	Iterative **mul**	Recursive **mul**
			
$N_N = 2$	$N_N = 1$	$N_N = 1$	$N_N = 0$
$N_R = 0$	$N_R = 1$	$N_R = 1$	$N_R = 2$
$N_L = 2$	$N_L = 1$	$N_L = 1$	$N_L = 0$
$E = 2$	$E = 2$	$E = 2$	$E = 2$
$e = \frac{2}{2+0} = 1$	$e = \frac{2}{1+1}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

Fig. 2. Factorial Loop Call Graphs and Metrics

Compute metrics. Given the loop call graph, we compute the metrics that quantify the algorithmic essence of the program. First, we define three direct absolute-scale metrics to count the different kinds of nodes in the loop call graph:

$N_N = |\text{non-recursive method nodes}|$

$N_R = |\text{recursive method nodes}|$

$N_L = |\text{loop nodes}|$

Second, we compute the *absolute essence* E, a simple indirect metric that counts the essential nodes in the loop call graph:

$E = N_L + N_R$

Finally, we compute indirect ratio-scale metrics to be able to reason about essence independently of program size, culminating in our measure of *relative essence* e.

$recursiveness = \frac{N_R}{N_N+N_R}$

$loopyness = \frac{N_L}{N_N+N_R}$

$e = \frac{E}{N_N+N_R}$

Figure 2 shows that E is the same (2) for all four implementations of the program. This means that E is agnostic to the choice between recursive or iterative implementations. Moreover, Figure 1 shows that E is not affected by the design of the program: no matter what degree of indirection is added, E stays the same (2). N_N, however, significantly changes with the design of the program: for Figure 1 it varies between 1 and 5, and it clearly correlates with the amount of indirection introduced by the programmer. Our main metric, e, also stays the same (1) for all four implementations in Figure 2. It does clearly show, though, the differences between the three implementations in Figure 1 (2, 1, and 0.4; for the “underdesigned”, nice, and ”overdesigned” implementations).

3.2 Forest Construction

Two steps in our approach require the construction of forests representing the cycle nesting structure in a graph. We need to find *loop forests* in control-flow graphs and *recursion forests* in loop call graphs.

Reducibility. Loop identification and loop forest construction are standard analyses in optimizing compilers. The control-flow graphs compilers operate on are usually derived from structured source programs, and thus they are usually *reducible*[3] (and even if they were not, optimizers could just decide to skip loop optimizations on irreducible control flow graphs). For this reason, the classic loop identification algorithms used in compilers are unable to properly identify loops in irreducible graphs. Unfortunately, the graphs we encounter in our static analysis, in particular the call graphs (for example those in Figures 5 and 6), are not necessarily reducible. Moreover, unlike a compiler optimization, we cannot just bail out if we encounter an irreducible graph (especially because many realistic call graphs are irreducible, and analyzing the call graph is essential). Thus, we cannot use the classical natural-loop identification algorithms for constructing our recursion forests.

Forests in irreducible graphs. We base our approach on prior work on loop identification in irreducible flow graphs. Ramalingam [14] provides an axiomatic framework for this problem and generalizes prior approaches by Sreedhar et al. [18], Havlak [7], and Steensgaard [19]. **Algorithm 1** represents the core of loop forest construction. It iteratively removes specific edges from the graph until the graph contains no more cycles. FindStronglyConnectedComponents corresponds to Tarjan's algorithm, and returns the set of strongly-connected components (the set of loop bodies). In the inner loop, the algorithm iterates over each loop body to determine its header nodes. Unlike in reducible graphs, in irreducible graphs loops can have multiple headers. IdentifyHeadersSteensgaard thus returns a set of header nodes for each loop. The algorithm adds each newly identified loop L into the loop forest F. RemoveLoopbackEdges in **Algorithm 3** then removes all *loopback* edges (edges that point to a header from within the body) from the graph. Finally, the algorithm repeats by again finding strongly-connected components on the now smaller graph.

Header identification. We use Steensgaard's variant of Ramalingam's approach, because it produces loops with headers that most closely correspond to the intuitive understanding of the recursion structure in programs. With Steensgaard's variant, the headers of a loop correspond to all entry nodes (all nodes pointed to from outside the loop). This is more intuitive than Havlak's or Sreedhar's variants, which consider subsets of the entry nodes to be headers.

[3] In a reducible flow graph [8], each strongly-connected component can only have one entry edge.

Algorithm 1 ConstructLoopForest

```
Require: graph $G = \langle N, E \rangle$
Ensure: loop forest $F = \langle \mathbb{L}, \mathbb{C} \rangle$
  {S}CC set $S$, loop $L$, loop body $B$, loop headers $H$
  {n}ode to loop map $M = \langle N, \mathbb{L} \rangle$
  $S \leftarrow$ FindStronglyConnectedComponents(G)
  while $S \neq \emptyset$ do
    for all $B \in S$ do
      $H \leftarrow$ IdentifyHeadersSteensgaard$(G, B)$
      $L \leftarrow (H, B)$
      $L_p \leftarrow$ FindParentLoop$(F, L, M)$
      $C \leftarrow (L_p, L)$
      $F \leftarrow F \cup \{(L, C)\}$
      RemoveLoopbackEdges(G, L)
    end for
    $S \leftarrow$ FindStronglyConnectedComponents(G)
  end while
  return $F$
```

4 Implementation

We implemented a static program analysis tool to measure the essence of Java programs. Our tool uses ASM [12] to statically analyze Java class files. We build control-flow graphs that include all exception edges and use class hierarchy analysis [1] to statically resolve polymorphic call targets. We implemented the forest construction algorithm described in Section 3.2. Our tool determines the number of loops, recursion headers, and total call graph nodes, and computes our metric of essence. It also produces a visualization of the loop call graph with nodes annotated accordingly.

Algorithm 2 FindParentLoop

```
Require: loop forest $F = \langle \mathbb{L}, \mathbb{C} \rangle$, new loop $L = \langle H, B \rangle$,
  node to loop map $M = \langle N, \mathbb{L} \rangle$
Ensure: return loop $L$'s parent $L_p$, or $\emptyset$ if $L$ is a root
  $L_p \leftarrow \emptyset$
  for all $n \in B$ do
    if $\exists L_m \in \mathbb{L}$ such that $(n, L_m) \in M$ then
      $L_p \leftarrow L_m$
    end if
  end for
  for all $n \in B$ do
    $M \leftarrow M \setminus \{(n, L_p)\} \cup \{(n, L)\}$
  end for
  return $L_p$
```

Algorithm 3 RemoveLoopbackEdges

```
Require: graph $G = \langle N, E\rangle$, loop $L\langle H, B\rangle$, $H \subseteq B \subseteq N$
Ensure: $\forall h \in H\ \forall b \in B\ (b, h) \notin E$
  for all $h \in H$ do
    for all $b \in B$ do
      $E \leftarrow E \setminus \{(b, h)\}$
    end for
  end for
```

Algorithm 4 IdentifyHeadersSteensgaard

```
Require: graph $G = \langle N, E\rangle$, loop body $B \subseteq N$
Ensure: loop headers $H \subseteq B$
  $H \leftarrow \emptyset$
  for all $n \in B$ do
    if $\exists n_1 \in N$ such that $n_1 \notin B \wedge (n_1, n) \in E$ then
      $H \leftarrow H \cup \{n\}$
    end if
  end for
  return $H$
```

5 Characterization

To evaluate our new metric, we studied the essence of a large number of Java programs. Our programs consist of the 100 applications of the Qualitas Corpus [20] release 20100719r and the two dominant Java benchmark suites: SPEC JVM [17] release 2008, and Dacapo [2] release 9.12-bach. We included all systems from all three suites. For Qualitas, we used the meta data to indicate which classes to analyze. Instead of including one JRE in the Qualitas corpus, we analyzed JRE 1.6.0 for three different platforms: Linux, Mac OS X, and Windows, because the Java runtime libraries can differ significantly by platform. For SPEC JVM, we analyzed the classes indicated in other characterization studies [21]. For Dacapo, the set of analyzed packages corresponds to the set of packages used when the benchmarks are run. We label each system with a suffix, according to the corpus it comes from (Q for Qualitas, S for SPEC JVM, D for Dacapo, and we use J for the JREs).

5.1 Size

Our corpus consists of 133 systems, with a total of just over 2 million Java methods and 229536 loops. The systems range in size between 16 and 257562 methods (median: 5132). 159052 of the methods are recursion headers, and there are a total of 38845 recursive cycles. While 31213 (80%) of those recursions have a single header method, 7632 contain multiple headers (and thus lead to irreducible call graphs).

5.2 Essence

The relative essence e of the systems ranges between 0.037 and 0.66 (median: 0.206). **Figure 3** plots relative essence versus program size. The logarithmic x-axis shows program size in terms of the number of loop call graph nodes. The highlighted band shows the first, second, and third quartiles of the essence distribution. Half of the systems lie within that band. Systems outside that band are somewhat unconventional.

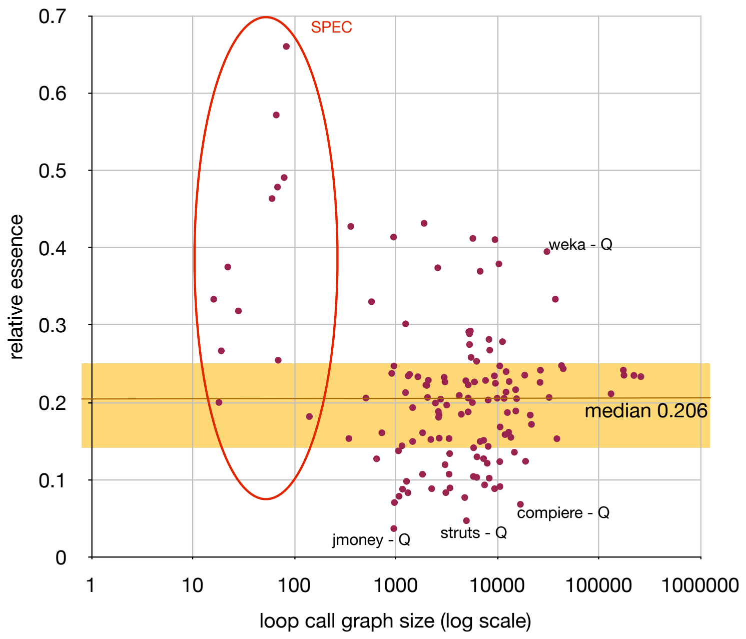


Fig. 3. Essence by program size

All the systems with fewer than 200 nodes come from the SPEC JVM suite. Many of them correspond to small, loop-driven algorithm implementations. The fact that our metric separates these systems from "normal" systems in the Qualitas corpus and the Dacapo suite corresponds to the findings of the authors of the Dacapo suite [2].

The relative essence of *larger* systems usually is closer to the median of the corpus. Large systems, such as the Mac OS X JRE with 219893 methods, are rarely written by a single developer or team, and thus they constitute a mix of code contributions, each contribution with a somewhat different design style. When measuring the essence of the entire system, the different design styles are mixed together and the essence gets close to the median essence of the corpus. For this reason, large systems with a *particularly high or low* relative essence are interesting. Weka is such a large system, with 30589 nodes and a high e of

0.395. Weka is a large collection of machine learning algorithms, and thus its high algorithmic essence is not surprising. JMoney is at the opposite end of the essence spectrum. It is a money management application based on the Eclipse platform. It consists mostly of glue code that connects various Eclipse plugins. It delegates most repetitions either to existing GUI controls (such as table grids) or collection classes (such as Java's Arrays.sort).

5.3 Essence by System

Figure 4 presents details about the essence of all the systems in our corpus. We ordered the systems on the x-axis by relative essence. The top chart shows the relative essence, consisting of its two components (loops and recursions). The bottom chart shows the size of the systems in terms of their number of loop call graph nodes. Bars are stacked according to the different node types: non-recursive methods ("normal"), recursive methods, and loops. The y-axis is cropped at 50000, however six systems had well over 50000 nodes (the three JREs, Dacapo's trade benchmarks, and the Qualitas version of Eclipse).

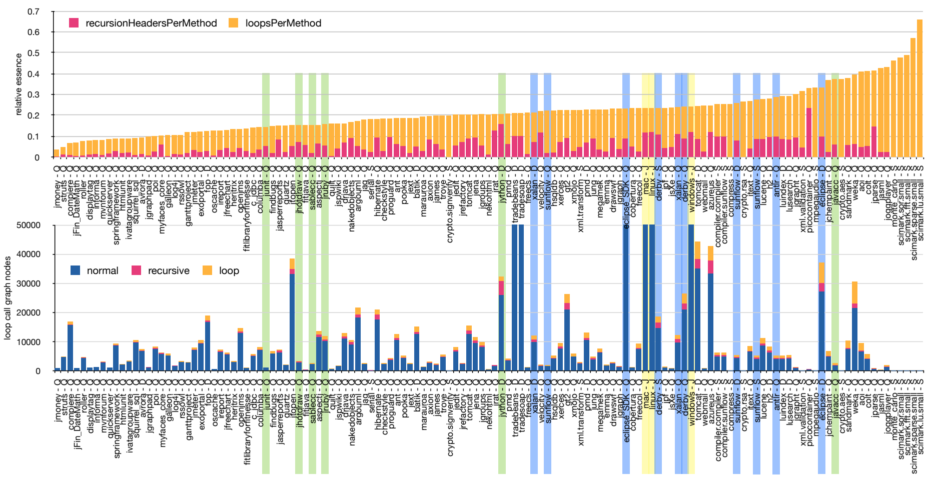


Fig. 4. Essence, recursive methods, and loops in Qualitas, SPEC, and Dacapo systems

Two of the systems in our corpus are traditionally used as good examples of design. JUnit is known for its high design pattern density, and JHotDraw is partially written by the same authors. Those two systems implement dramatically different functionality, however, their relative essence is relatively close (0.144 and 0.153, respectively). Their essence is significantly below the median of 0.206. One could use systems like these, with a design of an exemplary quality, as a reference for the amount of essence a well-designed system should have.

Our corpus contains several groups of related systems. The relative essence of each of the three JREs is above the median. The Windows and Linux versions, which are almost identical, have the same essence of 0.235. The Mac OS X version has a slightly higher essence of 0.242. That version includes additional libraries, such as the QuickTime multimedia framework, a possible source for more algorithm-dominated code. The corpus contains two versions of Derby, one in Qualitas and the other in SPEC. Given the different definitions of what constitutes part of a system, the set of analyzed classes in such cases can significantly differ. For Derby, even though the set of analyzed methods differs, the essence is nearly the same (0.235 vs. 0.242). For Xalan, the difference is relatively similar (0.213 vs. 0.240). For Sunflow, which is part of all three suites, the relative essence differs significantly (0.222, 0.258, 0.275). However, the three suites include significantly different subsets of the Sunflow classes. The essence differs for Eclipse in Dacapo (0.333) and Qualitas (0.233), but Qualitas includes a more recent version of Eclipse and a much larger set of plugins. JRuby (0.158) and Jython (0.206) are two runtime environments for dynamic languages. Jython makes much heavier use of recursion. Finally, SableCC (0.154) and Javacc (0.374) are located almost at the opposite ends of the spectrum. They are both parser generators. The SableCC documentation points out the focus on generating easy-to-maintain, object-oriented, design-pattern-based parsers. This indicates that the developers of SableCC followed the same design approach when writing SableCC itself. It is interesting to note that the relative essence of SableCC and the well-designed JHotDraw is the same.

6 Essence and Design

In this section we show that essence is directly related to object oriented design concepts, such as code smells, refactorings, and design patterns.

6.1 Essence and Code Smells

Code smells help to identify specific design problems. To understand how code smells relate to essence, we performed two tests.

First, we manually analyzed the commonly known code smells [5], to understand how they correlate with relative essence[4]. Of the 31 smells, 3 strongly indicate low relative essence, 2 weakly indicate low relative essence, 2 weakly indicate high relative essence, and 14 strongly indicate high relative essence. This means that most of the commonly known smells represent issues where the relative essence is too big. This may be an indication that in practice, most problems with software are related to the lack of modularization. Nevertheless, there are some smells (especially "Lazy Class/Freeloader", "Middle Man", and "Shotgun Surgery") that indicate that the relative essence is too low and the amount of indirection too high.

Second, in our corpus of 211507 real-world classes, we manually analyzed those classes that exhibited the highest and the lowest relative essence, to understand their design and to find potential code smells[5]. The 14 classes with the

[4] Table with results on `http://sape.inf.usi.ch/essence/smells`
[5] Table with results on `http://sape.inf.usi.ch/essence/outliers`

highest relative essence produce mostly “Long Method” smells, often accompanied by “Comments” to break the long method into understandable blocks. In our corpus, 125640 classes (59%) have zero relative essence. Out of that large pool, we picked two small samples. The first zero-essence sample represents the classes with the largest number of methods (and thus method nodes in the loop call graph). This sample, in which each class contains at least 90 methods, represents many “Data Classes” and “Middle Men”. However, it also contains automatically generated classes and adapters with almost empty default method implementations. To avoid the bias possibly introduced by exclusively focusing on classes with many methods, we also investigated a second, random, sample of zero-essence classes. That sample also contains some bad smells like “Data Class”, however many classes in that sample are participants in design patterns, and thus make positive use of indirection.

6.2 Essence and Refactorings

Refactoring is one way to fix the cause of a code smell. In Fowler’s “Refactoring” book [5], Beck differentiates between two kinds of refactorings: those that *add* indirection where programs are “missing one or more benefits of indirection” (such as for enabling of sharing of logic, for explaining the intent separately from implementation, for isolating change, and for encoding conditional logic through polymorphism), and those that *remove* “parasitic” indirection which isn’t paying for itself (such as left-over intermediate methods, or unused polymorphism). According to Beck, refactoring essentially maximizes the amount of design qualities while minimizing (or at least not unnecessarily increasing) the amount of indirection.

We analyzed how well-known refactorings relate to relative essence, and thus to the amount of indirection[6]. Of the 85 refactorings we studied, only 18 may increase relative essence (14 strongly, 4 of those to a lower degree). A much larger fraction, 47 refactorings, may decrease relative essence (42 strongly, 5 to a lower degree). Three pairs of refactorings most directly relate to relative essence: “Inline Method” and “Extract Method”, “Remove Middle Man” and “Hide Delegate”, and “Replace Delegation with Inheritance” and “Replace Inheritance with Delegation”. For each of these pairs, the first refactoring increases relative essence and the second refactoring (representing the inverse transformation) decreases relative essence. The first two pairs directly introduce or remove delegation. The last pair is a bit more subtle. It transforms between delegation and inheritance. Inheritance could be seen as an implicit form of delegation (dynamic method lookup instead of explicit indirection to the delegate in the application code).

6.3 Essence and Design Patterns

The presence of design patterns in a software system often is considered an indicator of good quality. It is easy to see how design patterns like a “Facade”,

[6] Table with results on `http://sape.inf.usi.ch/essence/refactorings`

an “Adapter”, a “Mediator”, or a “Bridge” introduce extra indirections and decrease the relative essence of software. The effect of some other design patterns on essence are a bit less straightforward. We thus manually analyzed this relationship[7] for the well-known “Gang of Four” [6] design patterns. None of the 23 patterns directly implies a higher relative essence. Only 4 patterns are mostly uncorrelated with relative essence. The vast majority (19) represent a design with a low relative essence. Given that design patterns generally reduce relative essence, relative essence could be seen as a measure of design pattern density.

In the remainder of this section we describe the effect of a more complicated pattern, “Visitor”, which is often used in a recursive context. As our running example, assume we want to implement a program that can evaluate abstract syntax trees that represent arithmetic expressions. **Algorithm 5** shows the essence of the solution in the form of pseudo-code. Its relative essence would be $e = N_R/N_R = 1$.

Algorithm 5 EVAL

```
Require: $n$ is a tree node, $n.type$ is its type, $n.value$ is its value, $n.left$ and $n.right$
  are its children
Ensure: return value = result of evaluating the subtree rooted in $n$
  if $n.type = LITERAL$ then
    return $n.value$
  else if $n.type = ADD$ then
    return $\text{EVAL}(n.left) + \text{EVAL}(n.right)$
  else if $n.type = SUBTRACT$ then
    return $\text{EVAL}(n.left) - \text{EVAL}(n.right)$
  else if $n.type = MULTIPLY$ then
    return $\text{EVAL}(n.left) \cdot \text{EVAL}(n.right)$
  end if
```

Listing 1.4 provides a possible implementation of this algorithm in Java, without using the visitor pattern. The tree is represented using a class hierarchy, rooted in an abstract superclass `Node`. The only method of that class, `eval()`, evaluates a node. `Node` has two subclasses, `Literal` which represents a constant value, and `BinOp`, which represents a binary operation. `BinOp` keeps references to its two operand nodes. `Add` is one of the three `BinOp` subclasses, representing an addition. `Subtract` and `Multiply` are analog to `Add`, and are omitted for brevity. In our example, the program entry point is `Main.calc()`, which receives a tree via a reference to its root node and contains a polymorphic call to `Node.eval()`.

The top of **Figure 5** shows the call graph of this program. The three essential nodes corresponding to the three `eval()` methods represent the headers of one recursive cycle. We call these kinds of recursive cycles “polymorphic recursions”, because their header nodes are implementations of a method declared in a super type, and because they are invoked polymorphically in a recursive way. The three

[7] Table with results on `http://sape.inf.usi.ch/essence/patterns`

Listing 1.4. Polymorphic implementation in Java

```
public abstract class Node {
 public abstract int eval();
}

public class Literal extends Node {
 private int value;
 public Literal(int value) {
  this.value = value;
 }
 public int eval() {return value;}
}

public abstract class BinOp extends Node {
 private Node a;
 private Node b;
 public BinOp(Node a, Node b) {
  this.a = a; this.b = b;
 }
 public Node getA() {return a;}
 public Node getB() {return b;}
}

public class Add extends BinOp {
 public Add(Node a, Node b) {
  super(a, b);
 }
 public int eval() {
  return getA().eval() + getB().eval();
 }
}

public class Main {
 public int calc(Node n) {return n.eval();}
}
```

nodes are all header nodes of this single cycle (according to Steensgaard's approach to header node identification), because there is a node outside the cycle, `Main.calc()`, which points to all of them. If we turned the call in `Main.calc()` into a monomorphic call (e.g. by changing the type of `Main.calc()`'s parameter from `Node` to `Add`), this would change: We would end up with one header node (`Add.eval`). However, the loop forest algorithm would detect a nested recursive cycle (after eliminating the loopback edges pointing to `Add.eval`), with `Subtract.eval` and `Multipy.eval` as its headers. This means that the number of recursive cycles would change, which is the reason for why we count header nodes and not recursive cycles when computing essence. With our approach, no matter whether we enter this recursive cycle via a polymorphic or a monomorphic call, we end up with three header nodes and the relative essence stays the same[8]: $e = (N_L + N_R)/(N_N + N_R) = (0 + 3)/(3 + 4) = 0.43$. The bottom of Figure 5 shows the recursion cycle forest of this program, which includes a single recursive cycle (rectangle) containing just the three header nodes.

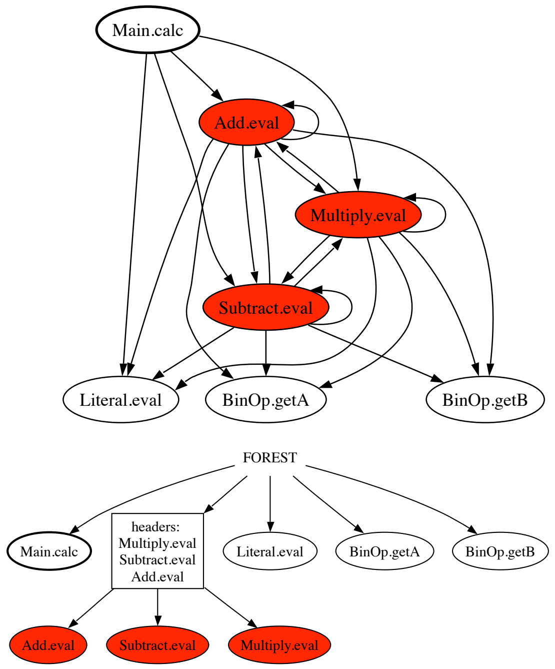


Fig. 5. OO call graph and recursion forest

For space reasons, we omit the visitor-based implementation of the traversal. We only show the resulting loop call graph at the top of **Figure 6**. As we can

[8] Note that header node identification approaches other than Steensgaard's can violate this property.

see, we still have three essential methods (the three concrete elements' `accept` methods). They form the headers of a larger recursive cycle, which also includes the visitor's `visit` methods. Our approach correctly identifies that the visitor pattern introduces an extra level indirection for each method. The number of essential nodes stays the same ($E = 3$), however, the relative essence decreases due to the extra level of indirection: $e = E/(N_N + N_R) = 3/(3 + 10) = 0.23$. Also note that the recursion forest at the bottom of the figure shows the same structure, which means that the visitor does not introduce any nested recursive cycles. This corresponds to intuition, because the introduction of the visitor pattern does not change the algorithmic aspect of the program.

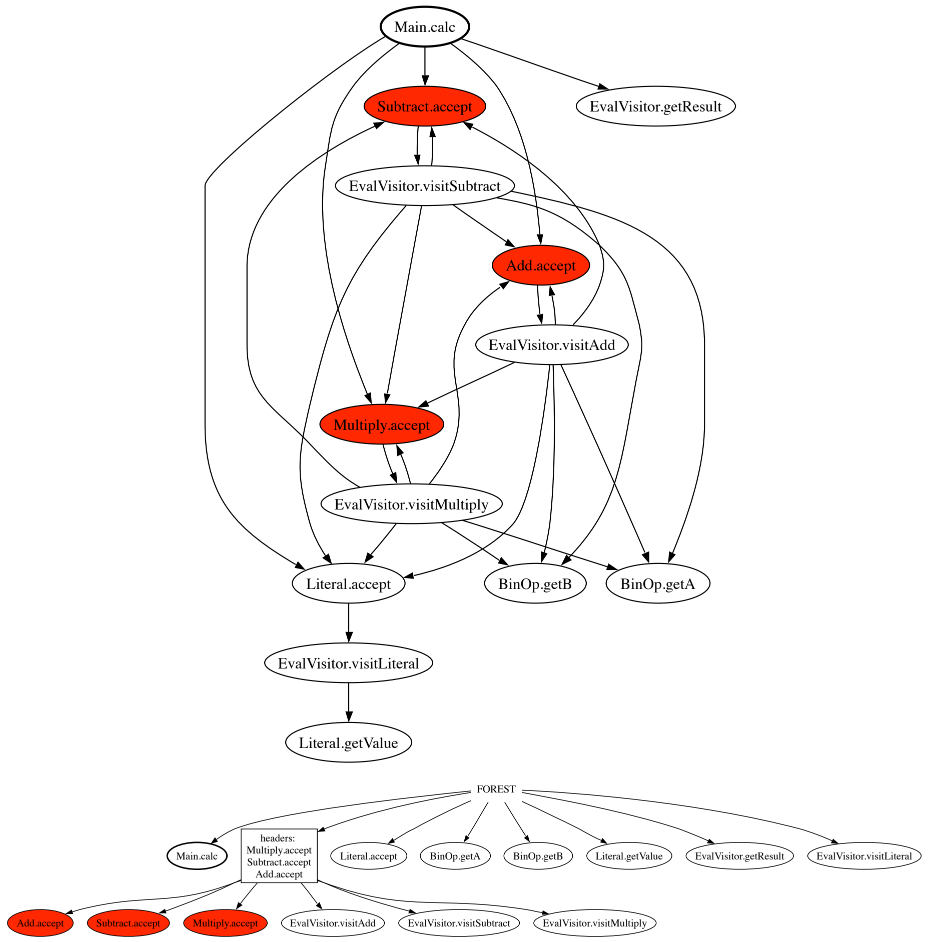


Fig. 6. Visitor call graph and recursion forest

7 Related Work

Our approach – and our metric – is related to and inspired by two seminal works in software engineering.

Our operational definition of essence, based on the loop call graph and our two metrics, is similar in spirit to Brook's intensional definition [3] of essence and accident. We start with the representation of a solution formulated in a concrete programming language, which, according to Brooks, contains essential and inessential aspects. Instead of finding an abstract model representing the complete essence (and only the essence) of a solution, we aim for the more tractable task of separating the parts of the concrete program that are likely essential from the parts that are likely accidental.

Parnas' "On the criteria to be used in decomposing systems into modules" [13] describes the concept of information hiding. The tool enabling information hiding is the use of indirection in the form of functions that hide design decisions and internal data structures. Relative essence is our approach towards quantifying the amount of such indirection.

7.1 Conceptual Relationship to Existing Metrics

Essence is a design metric. In this section and the next, we analyze whether essence just represents a different perspective on an existing design metric, or whether essence represents a design property that is not captured by existing metrics. While this section discusses the conceptual relationships between essence and intuitively related metrics, the next section empirically studies the correlation of essence with well-known existing metrics.

Essence is particularly related to four kinds of software metrics: design pattern density, coupling and cohesion, cyclomatic complexity, and bloat.

Design pattern density. Essence is related to Riehle's design pattern density [15]. Design pattern density determines "which parts of a design are design pattern instances, and which parts are not". It is defined as "the percentage on an object-oriented framework's collaborations that are design pattern instances". Measuring pattern density has many potential benefits, such as estimating the maturity or quantifying the ease of learning a framework. While design pattern density separates the "good" (pattern-based) parts of the design from the "bad" parts of the design, our new metric, essence, separates the overall design from the algorithmic core of a program. A second difference between our essence metric and design pattern density is that our metric is fully automatically computable from code, while design pattern density depends on the prior identification of collaborations and design patterns. While there is much active research in that direction, we are unaware of an approach to automatically and reliably identifying all the necessary collaborations and design patterns in code.

Coupling and cohesion. Essence is also related to coupling and cohesion [4]. While coupling and cohesion tell you what you should *move* around, essence tells you what you might want to *remove*, and it tells you about the absence of indirections you might want to *add*.

Cyclomatic complexity. Essence is related to McCabe's cyclomatic complexity [10]. Cyclomatic complexity characterizes method bodies consisting of basic blocks, we characterize programs consisting of methods and entire loops. Note that McCabe also defines a measure he calls "essential complexity", ev, which, however, is very different from our notion of essence. His ev measures the unstructuredness of a control-flow graph. Structured control-flow graphs, no matter how high their cyclomatic complexity, have $ev = 1$. Thus, essence is closer to McCabe's cyclomatic complexity, with the difference that we focus on repetitions (instead of *all* branches) and that we include recursion (which, as our results show, is a major contribution to the complexity of modern software systems).

Bloat. To some degree, essence could be considered a dual to bloat [11]: bloat focuses on unnecessary transformations of *data*, while we identify algorithmically inessential *code*.

7.2 Empirical Correlation with Existing Metrics

The above section discussed the relationship between relative essence and well known metrics on a conceptual level. In this section we empirically determine the correlation of relative essence with design metrics as computed by existing tools.

We believe that relative essence may be a surrogate for *design pattern density*. However, given the absence of an automatic approach to measuring design pattern density, we are not able to validate that hypothesis other than by argument. As we have shown in Section 6.3, the introduction of most design patterns reduces relative essence, and no pattern generally increases relative essence. Thus, at least for the "Gang of Four" patterns, we have reason to believe that pattern density is inversely correlated with relative essence.

Unlike design pattern density, many other design metrics can be computed automatically. However, as Lincke et al. [9] have shown, metric definitions can be ambiguous, and different metrics measurement tools often interpret metrics definitions differently. To make our comparison unambiguous we computed design metrics using four open-source tools: CyVis[9] for cyclomatic complexity, JDepend[10] for object-oriented design metrics, ckjm[11] for the Chidamber-Kemerer [4] metrics, and Dependency Finder[12] for more basic object-oriented metrics. Our study involved 137 metrics. These include metrics computed for each of the 211507 classes, each of the 11018 packages, or each of the 133 applications in our corpus[13]. We found that *none* of the existing metrics are correlated with relative essence. Cyclomatic complexity shows the closest correlation, with a Pearson's product moment correlation coefficient r=0.49. As a rule of thumb, correlation

[9] `http://cyvis.sourceforge.net/`
[10] `http://www.clarkware.com/software/JDepend.html`
[11] `http://www.spinellis.gr/sw/ckjm/`
[12] `http://depfind.sourceforge.net/`
[13] The complete results are available at `http://sape.inf.usi.ch/essence/metrics`.

coefficients with an absolute value below 0.6 or 0.7 are considered uncorrelated. **Figure 7** shows a scatterplot of cyclomatic complexity versus relative essence. It includes a regression line with a positive slope. Each class corresponds to an individual data point. The figure confirms that the correlation between cyclomatic complexity and relative essence is rather weak.

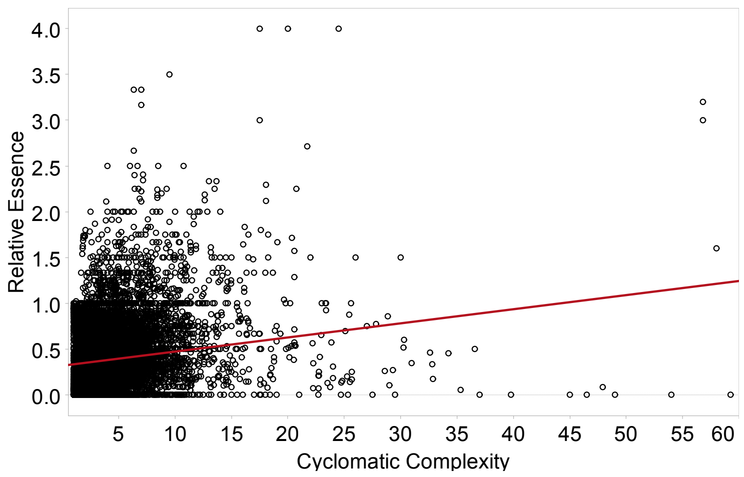


Fig. 7. Relative essence vs. cyclomatic complexity, class granularity: $r = 0.49$

The second metric, the only other metric with $|r| > 0.4$, is the number of local variables per method. The fact that classes with more local variables per method also show a higher relative essence makes intuitive sense, because methods involving loops or recursion often include local variables.

For all other metrics, $|r|$ lies below 0.3. This includes metrics like coupling and cohesion. In particular Chidamber-Kemerer's LCOM (lack of cohesion of methods) is entirely uncorrelated with relative essence (r=0.026).

8 Discussion

In this paper we introduced a novel structure, the loop call graph, and two metrics derived from that structure, absolute and relative essence. We have explained the intuition behind our approach, and we related it to code smells, refactorings, design patterns, and existing design metrics. In this section we discuss potential uses of the loop call graph and essence metrics, as well as the limitations of our approach.

8.1 Usage Scenarios

This paper lays the foundation for approaches and tools that analyze and improve systems based on the amount of indirection, modularization, or information hiding in those systems.

Deviation from reference. We believe a good use of our metric is to measure the essence of systems, and to compare it to a reference value of a system of high design quality (such as JUnit and JHotDraw). This is similar to uses of existing design metrics for quality control, where tools produce reports on violations based on a configurable range of admissible metric values. A high essence is indicative of a monolithic design that lacks the structure necessary for good understandability and maintainability. A low essence is indicative of an "overdesigned" system with excessive indirections, or of a system that consists mostly of glue connecting other systems together.

Problem localization. Besides a mere metric to flag potential problems, our approach also provides a representation (the loop call graph) that helps in locating the sources of the problem. In our own first uses of loop call graphs of student programs, we have found that for small programs (with tens of methods) a node-link diagram visualization, where essential nodes are highlighted, helps to quickly spot clusters of nodes with particularly high or low essence. Given that essence can be computed on any subgraph of the whole-program loop call graph, future approaches could automatically identify connected subgraphs with particularly high or low relative essence, and present these subgraphs to the developer as regions with bad smells. This would avoid the problem of visualizing loop call graphs of realistic programs, with their tens or hundreds of thousands of nodes.

Refactoring recommendation. We can group refactorings into three classes according to their impact on relative essence: refactorings that increase, do not affect, or decrease relative essence. Thus, the relative essence of a given subsystem could guide recommendations on which refactorings to perform. This would help to better modularize those subsystems with abnormally high relative essence, and to eliminate unnecessary indirections in subsystems with abnormally low relative essence.

Quality and process attribute prediction. Is it possible to predict process attributes, such as error rates or times to fix an error, based on our metric? Our hypothesis is that the distance of a system's essence from the essence of a reference system might be a predictor for various external product quality and process attributes. We would like to study this connection in future work.

8.2 Limitations

A main limitation of our approach is the mismatch between Brooks' *intensional* definition of essence, and our *operational* definition based on loop call graph nodes: (1) In a loop call graph, not all (inessential) non-recursive method nodes represent purely accidental complexity. Clearly, some methods do provide a meaningful algorithmic contribution even if they are not recursion headers and do not contain loops. (2) Not all (essential) recursion headers and loops are necessarily required for solving the problem the program needs to solve. A program may contain unnecessary algorithmic computations, or it may contain overly complex algorithms for a given problem. In general, we believe that *any* operational (and thus automatically measurable) definition of essence will be unable to determine with absolute certainty that a given piece of code is essential according to Brooks' intensional definition. However, we believe that our approach, while imperfect, is the first to try to quantify this fundamental property, and that it comes close to the intensional definition. By focusing on repetition we focus on the backbone of any algorithm, and we provide (as our results in Section 6 show), a metric that directly relates to design patterns, code smells, and refactorings.

9 Conclusions

While design metrics like coupling and cohesion provide hints for which parts of the system to *move* where, our new metric, essence, provides hints on which parts of the system to *remove*, and where to *add* extra indirections. Essence is a measure of the absence of indirection, layering, encapsulation, or delegation in a system. All of these aspects can positively affect software qualities such as modularity and understandability, however, too much indirection (and thus too little essence) can be an indication of over-engineered systems or even cruft.

Our metric, essence, is an internal product metric. To measure it, we only require the source or binary code of a software system. Essence is simple, precise, and can be measured automatically. It is based on the counts of well known and intuitive concepts: methods, loops, and the headers of recursive cycles in the call graph. Moreover, it can be efficiently computed. It took roughly 3 hours to calculate essence across the entire set of analyzed applications. Essence is a principled metric. It is based on the principle that loops and recursions are the only constructs that can increase the computational complexity of an algorithm. For this reason, any implementation of an algorithm will contain a "backbone" consisting of loops and recursions. By identifying loops and recursions, we can thus identify algorithmically essential parts of a program.

We have studied essence in the largest open corpus of software systems we are aware of, we have found that essence is not correlated to any existing design metrics, and we have found that essence is tightly related to design pattern density, to code smells, and to refactoring.

References

1. D. F. Bacon and P. F. Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, New York, NY, USA, 1996. ACM.

2. S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press.
3. F. P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
4. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.
5. M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, 1995.
7. P. Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Prog. Lang. Syst.*, 19(4):557–567, 1997.
8. M. S. Hecht and J. D. Ullman. Flow graph reducibility. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 238–250, New York, NY, USA, 1972. ACM.
9. R. Lincke, J. Lundberg, and W. Löwe. Comparing software metrics tools. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 131–142, New York, NY, USA, 2008. ACM.
10. T. McCabe. A complexity measure. *IEEE Transactions on Softw. Eng.*, SE-2(4):308 – 320, 1976.
11. N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to java runtime bloat. *Software, IEEE*, 27(1):56 –63, jan. 2010.
12. ObjectWeb. ASM. Web pages at `http://asm.objectweb.org/`.
13. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972.
14. G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Trans. Program. Lang. Syst.*, 24(5):455–490, 2002.
15. D. Riehle. Design pattern density defined. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 469–480, New York, NY, USA, 2009. ACM.
16. S. S. Skiena. Compiler optimization by detecting recursive subprograms. In *ACM '85: Proceedings of the 1985 ACM annual conference on The range of computing : mid-80's perspective*, pages 403–411, New York, NY, USA, 1985. ACM.
17. SPEC. SPECjvm2008 (Java Virtual Machine Benchmark). http://www.spec.org/jvm2008/.
18. V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Identifying loops using dj graphs. *ACM Trans. Program. Lang. Syst.*, 18(6):649–658, 1996.
19. B. Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Microsoft Research, October 1993.
20. E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010.
21. D. Zaparanuks and M. Hauswirth. Characterizing the design and performance of interactive java applications. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 23 –32, 28-30 2010.