Refactoring and Software Complexity Variability

By Alex Yakyma

The inherent complexity of software design is one of the key bottlenecks affecting speed of development. The time required to implement a new feature, fix defects, or improve system qualities like performance or scalability dramatically depends on how complex the system design is. In this paper we will build a probabilistic model for design complexity and analyze its fundamental properties. In particular we will show the asymmetry of design complexity which implies its high variability. We will explain why this variability is important, can and has to be efficiently exploited by refactoring techniques to considerably reduce design complexity.

Intro

There are different views on refactoring in the industry. Because refactoring is relatively neutral in respect to software development methodology, teams that practice scrum or even waterfall may apply refactoring techniques to their code. There are opinions on refactoring as a form of necessary waste (some authors elaborate the concept of pure waste vs. necessary waste (see [Elsamm]) attributing refactoring to necessary waste). This analogy often becomes extremely ironical since many executives and software managers think that refactoring is in fact a pure waste and thus should not be undertaken by teams. At the same time there is the very valid standpoint regarding refactoring as a method of reducing the technical debt (see [Cunning]). Some consider refactoring as a way of entropy reduction (see [Hohmann], p. 14). The importance of this team skill on the corporate scale is explained in [Leff], ch 20.

In all cases it is obvious that refactoring has to deal with something we call software design complexity – an overall measure of how difficult it is to comprehend and work (add new functionality, maintain, fix defects etc.) with a given software system.

Let us start analyzing complexity deeper in order to understand how to cope with it.

Modeling the Complexity

We base our model of software design complexity on its multiplicative nature. Let’s consider a list of factors that influence the complexity. It is not at all a full list and not necessarily the list in order of importance (applicable to OOP-based technology stack):

1. Meaningful class and member names
2. Compact method bodies
3. Usage of ‘typed’ collections
4. Usage of interfaces
5. Use of framework capabilities vs own implementations
6. Following single responsibility principle
7. Code duplication
8. ...

(1)
Let’s see what happens if we have a combination of any two factors from list (1). E.g. if we do not follow the single responsibility principle [WikiSRP] the code is hard to understand, debug, or maintain because objects of the same class can play considerably divergent roles in different contexts. At the same time not giving meaningful names to classes and their members makes code very hard to comprehend. A combination of these two has a multiplicative effect. To articulate this better let’s use an example:

Assume we have class A with a vague class name and member names. Then the person that debugs the code and encounters objects of this class will have to cope with some complexity C of the class caused by naming problem of this class and its members (for simplicity sake think of C as the effort required to understand what A means in the context of our debugging episode). Let’s now also assume that A fulfills 3 different responsibilities depending on the context. Then in order to understand the behavior of A in this specific context of debugging you need to analyze what each of the names (class, field or method) would mean in each of the three possible contexts spawned by roles for class A. In other words, the complexity is \( C \times 3 \).

Another example. If you invent a wheel every time instead of using existing language or framework capabilities you spawn a lot of unnecessary code (classes and their methods). If you also do not control the length of method bodies, then in each ‘extra’ method you have a lot of complications; so instead of just 20 extra methods you get 20 extra methods with each representing approximately 50% extra complexity because of its size resulting in \( 20 \times 1.5 = 30 \) ‘units’ of overall complexity.

So we may consider overall complexity C as a product of a big number of individual factors:

\[
C = f_1 \cdot f_2 \cdot ... = \prod_i f_i \quad (2)
\]

Another important characteristic of the factors above is that they are all independent. Taking into consideration the random nature of these factors and assuming their fairly typical properties (more detail in explanation section below) we conclude that:

**Software design complexity follows lognormal distribution.**

The following grey section contains the proof and is optional for a reader who wants to skip to immediate takeaways from this assertion.

More precisely the assertion above can be formulated as following:

**(Assertion 1)** Software design complexity is approximately a lognormaly distributed random variable.

Let’s prove that the assertion is true.

**Proof of Assertion 1.**

Let’s make two assumptions:

1) Random variables \( \{\ln (f_1), \ln (f_2), ...\} \) have finite means and variances. This assumption

---

1 It is important to note that our model describes random behavior of complexity at any arbitrary (but fixed) moment of time. In other words our model answers the following type of questions: what could be the design complexity of a product after the team works on it for, say, 2 months.

2 “Approximately” means that if for a second we assume that there is not just “big” but infinite number of factors in (1) then expression (2) can be ‘reduced’ to \( C_n \) – a product of the first \( n \) factors. Our assertion basically states that the distribution of \( C_n \) is infinitely close to lognormal with \( n \to \infty \).
absolutely makes sense from practical standpoint (e.g. usage of typed vs untyped collections in the source code may vary but have its average ratio (of say 40%) with finite standard deviation (let’s say, 20%); another example: method bodies would have an average length with deviation from the average approximately limited by its standard deviation etc.). Due to the fact that these factors have finite and relatively small amount of values, we may conclude that their logarithms also have finite means and deviations.

2) Lindeberg’s condition (see [L-Cond]) which only looks scary but actually means the fairly simple fact that the ‘outliers’ (that sit outside the ‘circle’ with radius composed of all variables’ standard deviations) represent a minor set.

With this all said, we may apply Central Limit Theorem (in its version by Lindeberg and Feller, see [CLT-L-F]) to the sequence of random variables \( \ln(f_1), \ln(f_2), \ldots \). This gives us:

\[
\frac{\sum_{i=1}^{n} \ln(f_i) - \sum_{i=1}^{n} \ln(\mu_i) }{\sqrt{\sum_{i=1}^{n} \ln^2(\sigma_i)}} \sim N(0,1) \tag{3}
\]

meaning that the expression on the left of (3) converges to a normally distributed random variable \( \text{rv} \) in distribution (see [CNVRG] for more detail). Here \( \ln(\mu_i) = E[\ln(f_i)] \), and \( \ln^2(\sigma_i) = E[(\ln(f_i) - [\ln(\mu_i)])^2] \) - mean and variance respectively.

But this means that the expression on the left in (3) is extremely close to normal distribution for big \( n \). Let’s fix some big integer \( n \). Then remembering that for any positive real numbers \( a \) and \( b \) \( \ln(a) + \ln(b) = \ln(ab) \) we have:

\[
\ln\left(\prod_{i=1}^{n} f_i\right) \approx^d \alpha N(0,1) + \beta \tag{4}
\]

where \( \alpha \) and \( \beta \) are constants (their meaning can be easily derived from (3)) and thus on the right side of (4) we have also a normally distributed variable \( \approx^d \) means that distribution functions of \( \text{rv’s} \) are approximately equal, not the \( \text{rv’s} \) themselves). This by definition means that \( C \) is approximately lognormal for big \( n \).

The analytical expression for the probability density function of a lognormally distributed \( \text{rv} \) can be found in [Lognorm] and is not of our current interest. Instead we will be more interested in its generic behavior.

**Analyzing the Model**

For a given moment of time \( t_1 \) the graph of design complexity pdf looks like this\(^3\):

As follows from figure 1, lognormal distribution is 'skewed'. Unlike normal distribution where \( \text{mean} = \text{mode} \) and pdf would be symmetrical w.r.t. mean value (representing well-known bell curve), lognormal pdf is skewed with its peak further to the right. This is more thorough expression of the fact that the complexity grows over time. Though note that the dynamic analysis of design complexity is beyond the scope of this paper.

---

\(^3\)To better understand how pdf changes with time: if \( t_1 < t_2 \) then obviously the complexity at the moment \( t_2 \) is also lognormal but its graph is more “stretched” lengthwise horizontal axis so that in particular its “peak” is further to the right. This is more thorough expression of the fact that the complexity grows over time. Though note that the dynamic analysis of design complexity is beyond the scope of this paper.
curve), in case of lognormal rv “smaller” complexities are “compressed” on the left of mode value while “higher” complexities scattered wide on the right of mode. In other words:

(Asymmetry of design complexity) It is unlikely to have design complexity less than the mode and likely to have complexity much higher than the mode.

This fact sounds like a pretty sinister beginning of our journey, but the following two facts mitigate the impact:

1. High variability of design complexity basically affects those teams that do not purposefully reduce complexity, and...
2. There is a reliable method of reducing complexity.

The method is exactly refactoring. It is easy to see that using our (or similar) factorization (see (1) above), it is obvious what exactly\(^4\) to refactor to counter the effect of multiplicativity of design complexity and thus keeping complexity under control. In fact refactoring is no less important than the creation of code in the first place. As Martin Fowler points out (see [Fowler], p. 56-57): "Programming is in many ways a conversation with a computer… When I’m studying code I find refactoring leads me to higher levels of understanding that I would otherwise miss.”

Note that while we are aiming at reducing the complexity we still accept the fact that there is no way to avoid the variability of ‘higher values’ for it is an objective statistical law for this type of rv. In other words there is no way the team could turn lognormal distribution into symmetrical one, even though they are best of the best developers.

Another important consequence of the design asymmetry for the economy of software engineering is that (because \(\text{mean} \neq \text{mode}\)) \(\text{in the long run there is considerable hidden extra effort in maintaining the product.}\) Indeed the most probable outcome for complexity after one episode of development is by definition equal to the \(\text{mode}\). But after being repeated multiple times it gravitates to \(\text{mean}\) and as we remember \(\text{mean} > \text{mode}\) in case of lognormal rv. Thus \(N\) such episodes yield the additional (and much worse – hidden) maintenance cost proportional to \(N \times (\text{mean} – \text{mode})\). This hidden extra effort can never be totally eliminated but can be reduced.

A team that purposefully refactors, either partially or totally reduces the impact of certain individual complexity factors. The high variability exactly means that you can dramatically succeed with refactoring in reducing design complexity.

Refactoring means changes in design. These changes (sometimes dramatically) modify the information flows within the system, re-organizing and re-distributing information in different ways which leads to uncertainty and

\(^4\) Note that in the factorization (1) we required that factors were independent. Although it was absolutely necessary for analysis purposes, it is not at all required for your own strategy of refactoring. We may securely use ‘overlapping’ refactoring approaches if the team finds that convenient. The example of such dependent factors (and respectively the refactoring techniques) can be: 1) Complex flag-based conditions in loops – the factor, ‘Remove Control Flag’ – the refactoring method (see [Fowler], p.245) and 2) Unnecessary nested conditional blocks – the factor, ‘Replace Nested Conditional with Guard Clauses’ as a refactoring approach (ib., p.250). Obviously when you reasonably replace nested conditionals with ‘guards’, it will also affect some flag-based loops replacing their complex conditions with return statements where it is appropriate. So 1) and 2) affect each other to a certain extent but it is ok to use both as part of your strategy.
introduces variability to the outcome. Reinertsen (see [Rein]) points out the exceptional importance of variability in the economics of product development. In our case the outcome of refactoring is also quantifiable – it is a team’s velocity in delivering user value. While refactoring utilizes the variability, unit testing keeps refactoring within the limits. Unit tests bring considerable certainty to the scene: when you change few lines of code and then make sure your tests still run – this means that system functionality is not or almost not broken at all and changes in design did not lead to a priory wrong design.

Unit testing and refactoring used in conjunction sustain the balance of variability and help utilize this variability for implementing effective design.

It is important to know that because of high variability of complexity and ability of refactoring to dramatically reduce one, refactoring becomes extremely important competitive advantage of software teams.

Summary

Software design is usually more complex than we may think and factors like long methods or ambiguous names are just a few examples of a long list of forces that dramatically increase the complexity. Although the asymmetry of design complexity means that high complexity is more probable, it also gives teams a clue of how to exploit this asymmetry to reduce it. Continuous purposeful refactoring reduces the complexity at a same dramatic ‘rate’ and is necessary to sustain software maintainability in the long haul.

References


[CLT-L-F] Central Limit Theorem.
http://mathworld.wolfram.com/CentralLimitTheorem.html

[L-Cond] Lindeberg's condition.
http://en.wikipedia.org/wiki/Lindeberg%27s_condition

[Lognorm] Lognormal distribution
http://en.wikipedia.org/wiki/Log-normal_distribution

[CNVRG] Convergence of random variables.
