The Ozone Hole Metric

Optimize software components by shrinking their ozone holes

Dirk Engel, info@engel-internet.de, Mar 4, 2021

Why?

A while ago I was looking for a metric that helps me to monitor and evaluate development and maintenance of a set of software components. During my search I found several quite specific code metrics but most of those cover only one technical aspect (examples can be found in Vogel, 2020). The results are often ambiguous and difficult to interpret, at least without looking into the code again. So it's hard to say if a value is too high or too low, good or bad, and in all cases it doesn't tell you much about the overall component state. Actually, that is not what I was searching for. I want something crisp, a metric that can be used to compare components – different ones as well as the evolution of a single one – in an easy way. Something that can be intuitively used like height in real life, where you can say right away without thinking: A is taller then B.

How?

Considering what I like to have represented by this metric – quality and system resource consumption – it becomes clear that not a single metric will be able to meet my needs, it needs to be a composite metric. A combination of the following parameters:

- Type classifier that differentiates between project specific (one-off), product line specific, and generic components
- Quality dimension
  - Risk which is usually the product of probability and impact. However, it's no easy to deal with probability in software. Depending on who you ask, the software is written *bug free* and running on *ideal* hardware. Consequently, the probability to run in a fault should equal 0% - unfortunately, it never is. For that reason I propose to replace probability with complexity. Complexity in turn can be seen as product of internal complexity (estimated by looking at factors like number of threads, use of sophisticated algorithms, etc.) and external complexity (depending on number of provided and used interfaces).
  - Branch coverage
  - Number of static code analysis findings and/or number of defect tickets
- Resource consumption
  - Binary size, e. g. the code section which not only refers to a system resource but also serves as an objective indicator for component size
  - CPU load
  - RAM usage

How can all these parameters be easily combined? My choice fell on a radar chart with quality parameters on the left-hand side and resource consumption to the right. The new metric is defined as the area surrounded by the parameter variables (Fig. 1).

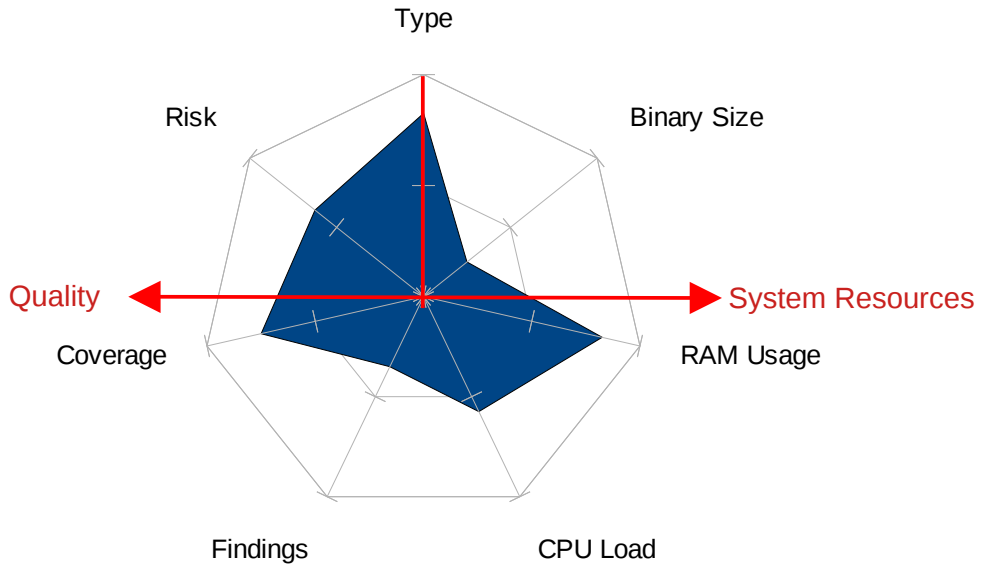

Fig. 1: Radar chart with the quality dimension on the left-hand side and the resource consumption on the right-hand side.

Up to here nothing is said about the actual values and range that should be used for the different parameters. However, as the title says, we like to "shrink" ozone holes, i. e. we strive for minimal areas in the radar chart and therefore for smaller parameter values. That is already given for the quality parameters as well as for risk and findings, smaller numbers mean better numbers, but what units and scale should be used? And what about type and branch coverage? Absolute or relative values?

Using percentages, i. e. setting the value of one parameter in relation to the overall sum of a defined set, instead of using absolute values has two advantages: The different factors are "automatically" normalized to the range 0..100. The second effect is that the percentages behave like a penalty function. If one component does not improve over time but others do, the relative percentage values increase. Stagnation is punished. Applied to aforementioned parameters we have:

- **T**ype:
  - generic implementation (one fits it all) → 0
  - product line specific implementation (one for many) → 50
  - product/project specific (one-off) → 100
- **R**isk: 0..100 as the product of complexity ↓0..10↑ and impact ↓0..10↑
- Branch **C**overage: 100 minus the percentage of the component's source code branch coverage
- **F**indings: 0..100, number of findings in relation to total number in given set

- Binary **S**ize: 0..100, code section size in relation to total size of given set
- CPU **L**oad: 0..100, DMIPS in relation to total CPU load of given component set
- RAM **U**sage: 0..100, heap and stack usage in relation to the total usage of given set

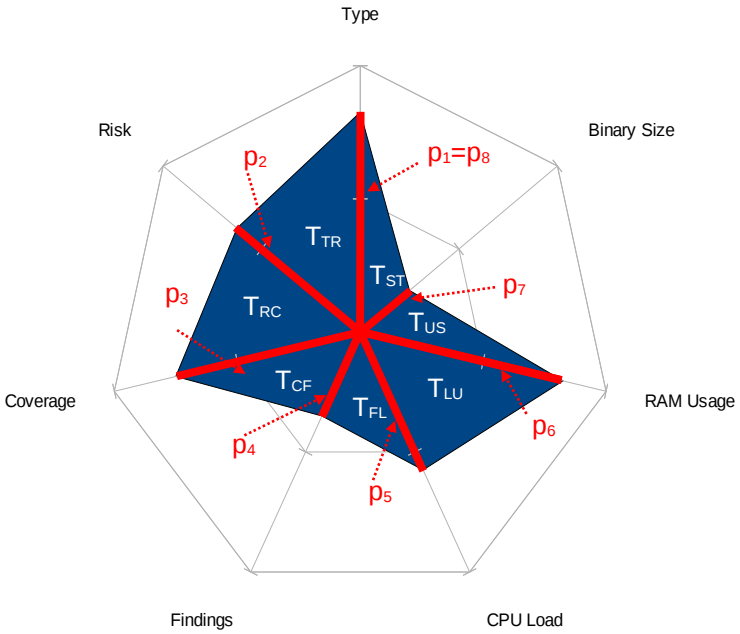

Fig. 2: The different components / triangles of the ozone hole radar chart.

The upper middle axis *Type* spans a triangle T_{TR} with *Risk* and a triangle T_{ST} with *Binary Size* (Fig. 2). When facing a large T_{TR}, a discussion should follow whether it would be a better approach to realize such a central and risky component as reusable (configurable) platform component instead of having one-offs. A large T_{ST} is an indicator for redesign or decomposition, better make this one a generic component and, if possible/reasonable, then decompose in order to get smaller reusable elements. The triangle T_{RC} defined by *Risk* and *Coverage* shows that you are flying blindly, you can't know what surprises are hidden in the uncovered source code. A large T_{CF} clearly reveals quality weaknesses of the given component and when come across a large T_{UL} you better think twice. High system resource consumption is not bad per se, of course, but when one component is frontrunner for CPU and RAM, you know at least where to concentrate your optimization efforts.

OK, so finally we have a nice diagram fulfilling the previously defined needs about quality and resource consumption indication – but is it really a metric?

Software quality metric: A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality (IEEE, 1998).

Well, if you like, you can turn the radar chart into a single number by calculating the overall area of all triangles:

$$OH = \frac{1}{2}\sin\left(\frac{360}{n}\right)\sum_{1}^{n} p_i p_{i+1}$$

However, don't simplify too much, you'll lose important information by doing this. And it is much harder (or even impossible) to deal with oversimplified models. In my opinion the use of FTE in planning serves sometimes as a bad example (Engel, 2018). Unfortunately, it's often too late when you recognize that you may neglected some important aspects (also known as Dunning Kruger effect, Kruger, 1999).

What?

The Ozone Hole Metric (OH) gives you a means to find hot spots in your set of software components and helps to monitor the progress over time. Identified hot spots of one decomposition level can be tracked down to the next lower level of the system (Fig. 3). To do so, the parameters of a component in level n are set in relation to the total numbers of its level n subsystem. A level n subsystem in turn forms a level n+1 composite component and the resulting parameters are set into level n+1 context. That way, the focus is automatically reset in each level to the most striking areas with the highest percentages.

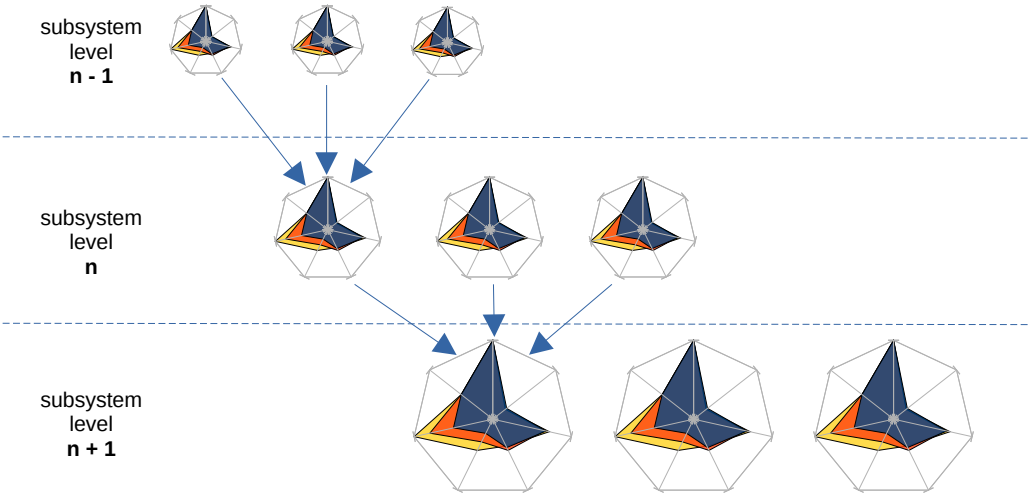

Fig. 3: The Ozone Metric follows your system decomposition hierarchy.

The radar charts can be used to present the component specific progress during the PI planning event. It facilitates the retrospective and when analyzed during the IP sprint, it gives you guidance for the next enablers.

You still wonder what all this has to do with the ozone hole? It's simply that I associated the first charts I looked at with the illustrations of the ozone hole. Imagine the silhouette of Antarctica in the background and you may have the same association. Furthermore, the goal is quite the same, the smaller the better is true for both, the real and the software ozone holes. Finally, it is a good narrative (Story Bias, Dobelli, 2013), who is not keen to help shrink the ozone hole? And what do you think sounds better when asked what you've been doing all day: "I fixed one or two weird findings of our static code analysis tool" or "I successfully helped my team to reduce ozone holes" (Framing, Dobelli, 2013)? Even the use of the metric abbreviation is straightforward, a smaller OH is better than a higher OH, MY GOODNESS.

References

Dobelli, R., 2013. *The Art of Thinking Clearly: Better Thinking, Better Decisions*. Hodder & Stoughton Ltd.

Engel, D, 2018, *Why FTE based planning is not useful for managing SW engineering teams.* LinkedIn. https://www.linkedin.com/pulse/why-fte-based-planning-useful-managing-sw-engineering-dirk-engel/ (last access 2021/03/06).

Institute of Electrical and Electronics Engineers (pub.), 1998: *IEEE Std 1061-1998. IEEE Standard for a Software Quality Metrics Methodology*. IEEE, New York 1998, ISBN 1-55937-529-9, Chapter 2. Definitions, S. 2.

Kruger, J., Dunning, D., 1999. *Unskilled and unaware of it. How difficulties in recognizing one's own incompetence lead to inflated self-assessments*. In: *Journal of Personality and Social Psychology*. Band 77, Nr. 6, 1999, S. 1121–1134. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.64.2655&rep=rep1&type=pdf (last access 2021/03/06).

Vogel, M., Knapik, P., Cohrs, M., Szyperrek, P., Pueschel, W., Etzel, H., Fiebig, D., Rausch, A., Kuhrmann, M., 2020. *Metrics in Automotive Software Development: A Systematic Literature Review*. In: Journal of Software: Evolution and Process. 10.1002/smr.2296. https://onlinelibrary.wiley.com/doi/10.1002/smr.2296 (last access 2021/03/06).