Growing Executable Code-Quality-Knowledge Organically

Daniel Speicher, Tiansi Dong, Christian Bauckhage, Armin B. Cremers
{dsp, dongt, bauckhage, abc}@bit.uni-bonn.de
Bonn-Aachen International Center for Information Technology, Universität Bonn

Introduction Writing tests before the actual code gives a clear guiding perspective to the implementation and provides immediate and repeatable feedback once the test passes. If the developer chooses small enough steps, all aspects of the code are motivated and verified by a test and conversely all tests have relevance. All code is "pulled" by tests into existence.[1]

"Executable code-quality-knowledge" is our term for a system of rules providing a partial definition of how the code should be structured ("concepts") and what to avoid ("smells")[2]. In the following we elaborate a process to let these rules *grow organically* together with tests and code. "Organic" growth relies on what is already at hand: *code samples* of good or bad quality, or imperfections in the rules surfacing as *false positives*. All rules should be "pulled" by a real demand into existence. Our tool[3] is based on a model that facilitates such an incremental process.

When a team lets the executable code-quality-knowledge grow organically, they can restrain the rules to those concepts and smells they thoroughly understand and consider relevant enough to commit to them and to let them be checked automatically. Tests should be as complete as possible but there has to be some slack in quality criteria to let developers explore alternatives. There is often a moment of choice in how code is structured and how concepts are expressed. Still, consistency is helpful to keep the code understandable. A process to grow code-quality-knowledge is especially helpful when working with "new" domains[4] or in an educational setting.

Model The relation of the real system, its real environment, code, tests, rules and samples is illustrated in Fig. 1. Code defines the behavior of a system. Tests verify that the code leads to the expected behavior. Tests and code are a model of certain aspects of reality. Rules of the executable code-quality-knowledge are a model of certain aspects of the code. We use annotated code samples as "tests" for the rules [4].

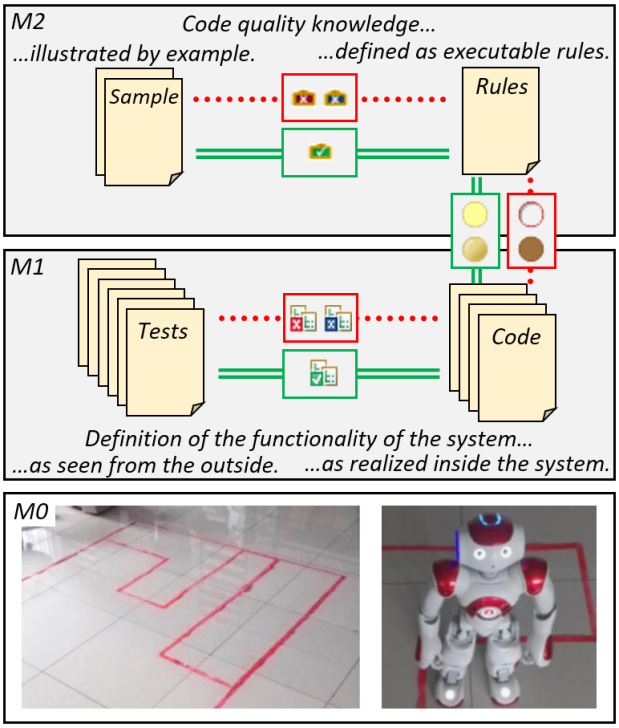

Figure 1: Meta-Levels. Tests and code on M1 define how the real system on M0 should behave in its real environment. The rules on M2 evaluates the internal quality of the the code. Samples are for rules like tests for code. Images taken from [5] where high level specifications (on level M1) are discussed. Robotics is a domain, where specific code quality criteria apply.

These rules define *analytical* and *constructive concepts* as well as *smells* and *shines*. Whether a program element falls under an analytical concept is defined by a single rule. We may define a concept INSTANTIATOR METHOD by a rule that is true for any method, that contains an instantiation. The IDE can show (🟡)[5] all elements that fall under this concept. There is no doubt whether a program element is an INSTANTIATOR METHOD or not. Either it has the defining property or not. This is different for constructive concepts like design patterns. We may expect a FACTORY METHOD to contain an instantiation, but in addition the method should return a new instance of a sub-

[1] Freeman and Pryce use the term "growing software" for this approach because it allows to "have something working at all times, making sure that the code is always as well-structured as possible and thoroughly tested." [1, p. xvii]

[2] "Concept" and "smell" are the keywords we use in our implementation. In [3] we used the terms "design idea" and "design flaw". When an "idea" gets defined it becomes a "concept".

[3] Cultivate based on JTransformer, Eclipse and SWI-Prolog.

[4] I.e. a domain the team has not yet worked in – may it have been in the recent years mobile computing, micro service architecture, robotics, or machine learning.

[5] Cultivate creates optionally an "editor annotation" that highlights the element (here: method declaration) in the editor and is shown as a yellow circle on a vertical ruler.

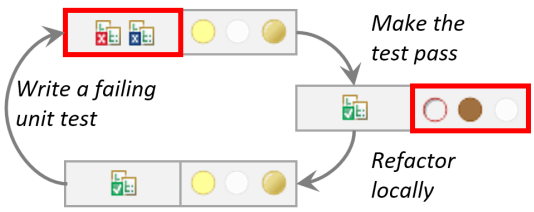

Figure 2: Test-first with code quality feedback.

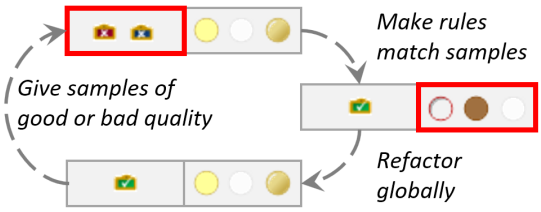

Figure 3: "Sample-first" global refactoring.

type of its return type, and it should be overridden or overriding. As it is not clear, whether these conditions are sufficient and because we want to be aware of imperfect FACTORY METHODS, constructive concepts are defined by (i) one rule that defines which program elements are meant to fall under the concept (*extension*) and (ii) a list of necessary conditions (*intension*). The intended extension is typically communicated by the developer through aspects of the source code that are not interpreted by the compiler like identifier names[6]. The IDE may again show elements that are meant to fall under this concept and either fulfill all necessary conditions (●) or not (○). Smells are defined by a rule indicating the problematic elements of a certain concept. Shines are defined analogously. The IDE again shows the smelly (●) or shiny (●) elements.

Process The process of growing software guided by tests (see [1]) with code-quality-knowledge in place is illustrated in Fig. 2. After the developer saw a new test fail and made it pass by evolving the code, her intuition where to refactor may be backed by the IDE showing smells or incomplete instances of concepts.

The process of "sample-first" global refactoring in Fig. 3 grows the executable code-quality-knowledge. The developer creates or copies sample code into a "museum" folder and annotates bad and good elements.[7] She then creates an executable smell definition that matches the bad but not those good elements. Then she refactors all the elements shown as smelly by the IDE.[8] Refactoring in Fig. 2 is about fixing smells of different kinds locally, while refactoring in Fig. 3 is about fixing one kind of smell globally.

The combined process in Fig. 4 adds a third option to account for false positives. If the developer recognizes a potential smell to be a false positive – which implies that the samples have not been representative enough – she revises the samples and consequently the rules. In [2] we had suggested that the established knowledge about design concepts and smells is not without contradiction and that adapting smells with respect to concepts may highly improve their precision. In [3] we reported about an experiment of applying the LAW OF DEMETER to JHotDraw 5.1 designed to generate a high number of false positives. These false positives lead us to a variety of interesting concepts to which we adapted the smell and reached almost perfect precision. In a later experiment we applied the adapted smell to AuctionSniper (from [1]). The many false positives were easily removed, mostly by defining the extension of already given concepts. In addition the process led us to the most interesting concept in this software: An INTERNAL DOMAIN-SPECIFIC LANGUAGE to specify test expectations.

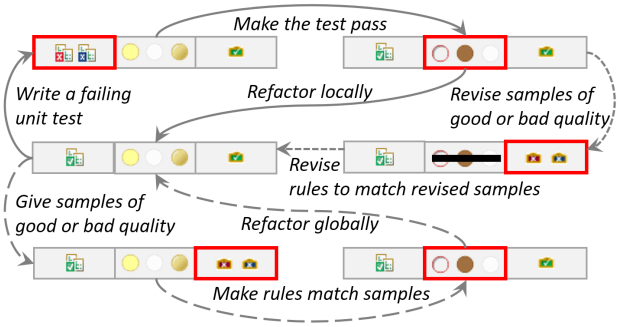

Figure 4: Combined process. Adding functionality test-first ⊕ Starting global refactorings "sample-first" ⊕ Revising the executable code-quality-knowledge based on false positives.

Future Work We plan to explore the given process for Robotics and Machine Learning. Furthermore our own software needs quality improvement, which asks for a controlled experiment to bootstrap code quality. Machine Learning approaches to code quality may benefit from the integration of the given model.

References

[1] S. Freeman and N. Pryce. *Growing Object-Oriented Software, Guided by Tests*. AW Professional, 1st edition, 2009.

[2] D. Speicher. Code Quality Cultivation. In *IC3K 2011, Revised Selected Papers*. Springer Berlin Heidelberg, 2013.

[3] D. Speicher. Polishing Design Flaw Definitions. *WSRE 2016, Softwaretechnik-Trends, Band 36, Heft 2*, 2016.

[4] D. Speicher, J. Nonnen, and A. Bremm. Code Museums as Functional Tests for Static Analyses. In *WSR 2012, Softwaretechnik-Trends, Band 32, Heft 2*, 2012.

[5] H. Sun, X. Ma, T. Dong, and A. B. Cremers. An Assertion Framework for Mobile Robotic Programming with Spatial Reasoning. In *42st Annual IEEE Computer Software and Applications Conference (COMPSAC'18)*, 2018.

[6] Package, class, field, method names. Prefixes or postfixes of them. Marker interfaces. Annotations. Placement within certain packages. References from configuration files.

[7] From now on this code sample is not meant to be executed but only to be analyzed. It moves conceptually from M1 to M2.

[8] The same process may as well start from a sample of good quality. The developer then defines a constructive concept and refactors the incomplete instances. Shines and analytical concepts may be defined similarly but do not require refactorings.