How Do Java Methods Grow?

Daniela Steidl, Florian Deissenboeck
CQSE GmbH
Garching b. München, Germany
{steidl, deissenboeck}@cqse.eu

***Abstract*—Overly long methods hamper the maintainability of software—they are hard to understand and to change, but also difficult to test, reuse, and profile. While technically there are many opportunities to refactor long methods, little is known about their origin and their evolution. It is unclear how much effort should be spent to refactor them and when this effort is spent best. To obtain a maintenance strategy, we need a better understanding of how software systems and their methods evolve. This paper presents an empirical case study on method growth in Java with nine open source and one industry system. We show that most methods do not increase their length significantly; in fact, about half of them remain unchanged after the initial commit. Instead, software systems grow by adding new methods rather than by modifying existing methods.**

I. Introduction

Overly long methods hamper the maintainability of software—they are hard to understand and to change, but also to test, reuse, and profile. Eick et al. have already determined module size to be a *risk factor* for software quality and "cause for concern" [1]. To make the code modular and easy to understand, methods should be kept small [2]. Small methods also enable more precise profiling information and are less error-prone as Nagappan et al. have shown that method length correlates with post-release defects [3]. Additionally, the method length is a key source code property to assess maintainability in practice [4], [5].

Technically, we know *how* to refactor long methods—for example with the *extract method* refactoring which is well supported in many development environments. However, we do not know *when* to actually perform the refactorings as we barely have knowledge about: How are long methods created? How do they evolve?

If methods are born long but subsequently refactored frequently, their hampering impact on maintainability will diminish during evolution. In contrast, if overly long methods are subject to a broken window effect[1], more long methods might be created and they might continue to grow [7]. Both scenarios imply different maintenance strategies: In the first case, no additional refactoring effort has to made as developers clean up on the fly. In the second case, refactoring effort should be made to prevent the broken window effect, *e. g.*, by refactoring a long method immediately after the first commit.

To better know when to refactor overly long methods, we need a deeper understanding of how methods evolve. In a case study on the history of ten Java systems, we study how single methods and the overall systems grow. We track the length of methods during evolution and classify them as *short*, *long*, and *very long* methods based on thresholds. The conducted case study targets the following five research questions regarding the evolution of a single method (RQ1, RQ2, RQ3) and the entire system (RQ4, RQ5):

RQ1 *Are long methods already born long?* We investigate whether methods are already long or very long at their first commit to the version control system or whether they gradually grow into long/ very long methods.

RQ2 *How often is a method modified in its history?* We examine which percentage of methods is frequently changed and which percentage remains unchanged since initial commit.

RQ3 *How likely does a method grow and become a long/very long method?* We check if methods, that are modified frequently, are likelier to grow over time, or, in contrast, likely to be shortened eventually.

RQ4 *How does a system grow—by adding new methods or by modifying existing methods?* We build upon RQ2 to see how many changes affect existing methods and how many changes result in new methods.

RQ5 *How does the distribution of code in short, long, and very long methods evolve?* We investigate if more and more code accumulates in long and very long methods over time.

The contribution of this paper is two-fold: First, we give insights into how systems grow and, second, we provide a theory for when to refactor. As the case study reveals that about half the methods remain unchanged and software systems grow by adding new methods rather through growth in existing methods, we suggest to focus refactoring effort on newly added methods and methods that are frequently changed.

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant Q-Effekt, 01IS15003A". The responsibility for this article lies with the authors.

[1]The broken window theory is a criminological theory that can also be applied to refactoring: a dirty code base encourages developers to try to get away with 'quick and dirty' code changes, while they might be less inclined to do so in a clean code base [6], [7].

II. Related Work

A. Method Evolution

Robles et al. analyze the software growth on function level combined with human factors [8]. However, the authors do not use any origin analysis to track methods during renames or moves. Instead, they exclude renamed/moved functions. The result of our RQ2 is a replication of their finding that most functions never change and when they do, the number of modifications is correlated to the size—but our result is based on a thorough origin analysis. Whereas Robles et al. further study the social impact of developers in the project, we gain a deeper understanding about how systems and their methods grow to better know when methods should be refactored.

More generally, in [9], Girba and Ducasse provide a meta model for software evolution analysis. Based on their classification, we provide a *history-centered* approach to understand how methods evolve and how systems grow. However, the authors state that most history-centered approaches lack fine-grained semantical information. We address this limitation by enriching our case study with concrete examples from the source code. With the examples, we add semantic explanations for our observed method evolution patterns.

B. System Size and File Evolution

Godfrey and Tu analyze the evolution of the Linux kernel in terms of system and subsystem size and determine a super linear growth [10]. This work is one of the first steps in analyzing software evolution which we expand with an analysis at method level. In [11], the authors discover a set of source files which are changed unusually often (active files) and that these files constitute only between 2-8% of the total system size, but contribute 20-40% of system file changes. The authors similarly mine code history and code changes. However, we gain a deeper understanding about the evolution of long methods rather than the impact of active files.

C. Origin Analysis

In [12], we showed that up to 38% of files in open source systems have at least one move or rename in its history which is not recorded in the repository. Hence, tracking code entities (files or methods) is important for the analysis of code evolution. We refer to this tracking as origin analysis.

In [13], [14], the authors present the tool Beagle which performs an *origin analysis* to track methods during evolution. This relates to our work, as we also conduct a detailed origin analysis. However, instead of using their origin analysis, we reused our file-based approach from [12] which has been evaluated thoroughly and was easily transferable to methods.

To analyze method evolution, the authors of [15] propose a sophisticated and thoroughly evaluated approach to detect renamings. Our origin analysis can detect renamings as well, however, at a different level: Whereas for us, detecting a renaming as an origin change suffices, the work of [15] goes further and can also classifies the renaming and with which purpose it was performed. As we do not require these details, our work relies on a more simplistic approach.

In [16], the authors propose a change distilling algorithm to identify changes, including method origin changes. Their algorithm is based on abstract syntax trees whereas our work is based on naming and cloning information. As we do not need specific information about changes within a method body, we can rely on more simplistic heuristics to detect origin changes.

Van Rysselberghe and Demeyer [17] present an approach to detect method moves during evolution using clone detection. However, in contrast to our work, as the authors use an exact line matching technique for clone detection, their approach cannot handle method moves with identifier renaming.

D. Code Smell Evolution

Three groups of researchers examined the evolution of code smells and all of them reveal that code smells are often introduced when their enclosing method is initially added and code smells are barely removed [18]–[20]. Our paper explains these findings partially: As most methods remain unchanged, it is not surprising that their code smells are not removed. For the specific code smell of long methods, our paper confirms that the smell is added with the first commit (many long methods are already born long) and barely removed (many long methods are not shortened).

III. Terms and Definition

Commit and Revision. A commit is the developer's action to upload his changes to the version control system. A commit results in a new revision within the version control system.

Initial and Head Revision. We refer to the initial revision of a case study object as the first analyzed revision of the history and to the head revision as the last analyzed revision respectively. Hence, the initial revision must not necessarily be the initial revision of the software development, and the head revision is likely not the actual head revision of the system.

Number of Statements. To measure the length of a method, we use the number of its statements. We normalize the formatting of the source code by putting every statement onto a single line. Then, we count the lines excluding empty lines and comments. The normalization eliminates the impact of different coding styles among different systems such as putting multiple statements onto a single line to make a method appear shorter.

Green, yellow, and red methods. Based on two thresholds (see Subsection IV-C), we group methods into short, long, and very long entities, and color code these groups as green (short), yellow (long), and red (very long) methods both in language as in visualization. A green (yellow/red) method is a method that is green (yellow/red) in the head revision.

Touches and Changes. We refer to the touches per method as the creation (the initial commit) plus the edits (modification) of a method, to the changes as the edits excluding the creation of the method—hence: $\#\text{touches} = \#\text{changes} + 1$.

TABLE I: Case Study Objects

Name	History [Years]	System Size [SLoC]			Revision Range	Commits	Methods		
		Initial	Head	Growth			Analyzed	Yellow	Red
ArgoUML	15	9k	177k		2-19,910	11,721	12,449	5%	1%
af3	3	11k	205k		18-7,142	4,351	13,447	5%	1%
ConQAT	3	85k	208k		31,999-45,456	9,308	14,885	2%	0%
jabRef	8	3.8k	156k		10-3,681	1,546	8,198	10%	4%
jEdit	7	98k	118k		6,524-23,106	2,481	6,429	9%	2%
jHotDraw7	7	29k	82k		270-783	435	5,983	8%	1%
jMol	7	13k	52k		2-4,789	2,587	3,095	7%	2%
Subclipse	5	1k	96k		4-5,735	2,317	6,112	9%	2%
Vuze	10	7.5k	550k		43-28,702	20,676	29,403	8%	3%
Anony.	7	118k	259k		60-37,337	5,995	14,651	5%	1%
Sum	**72**	**375.3k**	**1,912k**			**61,417**	**114,652**		

IV. Approach

We analyze a method evolution by measuring its length after any modification in history. We keep track of the lengths in an *length vector*. Based on the length vector, we investigate if the length keeps growing as the system size grows or, in contrast, if methods are frequently refactored.

A. Framework

We implemented our approach within the incremental analysis framework Teamscale [21]. Teamscale analyzes every single commit in the history of a software system and, hence, provides a fine-grained method evolution analysis. To obtain the complete history of a method, we use the fully qualified method name as key to keep track of a method during history: the fully qualified method name consists of the uniform path to the file, the method name, and all parameter types of the method. In cases the fully qualified method name changes due to a refactoring, we conduct an *origin analysis*.

B. Origin Analysis

To obtain an accurate length measurement over time, tracking the method during refactorings and moves is important—which we refer to as *origin analysis*. If the origin analysis is inaccurate, the history of an entity will be lost after a method or file refactoring and distort our results. Our method-based origin analysis is adopted from the file-based origin analysis in [12] which analyzes each commit incrementally. In principle, this origin analysis detects origin changes by extracting possible origin candidates from the previous revision and using a clone-based content comparison to detect the most similar candidate based on thresholds. We evaluated it with manual random inspection by the authors on the case study data.

We use a total number of 4 heuristics which are executed for each commit on the added methods—an added method is a method with a fully qualified method name that did not exist in the previous revision. For each added method, the heuristics aim to determine if the method is *truly* new to the system or if the method existed in the previous revision with a different fully qualified method name. A method that is *copied* is not considered to be new. Instead, it will inherit the history of its origin. The following heuristics are executed sequentially:

Parameter-change-heuristic *detects origin changes due to a changed parameter type.* It compares an added method to all methods of the previous revision in the same file and with the same name. The comparison is done with a type-II-clone detection on the content of the method body as in [12]. If the content is similar enough based on thresholds we detect this method as origin for the added method.[2]

Method-name-change-heuristic *detects origin changes due to renaming.* It compares an added method to all methods of the previous revision that were in the same file and had the same parameter types with the same content comparison.

Parameter-and-method-name-change-heuristic *detects origin changes due to renaming and changed parameter type* It compares an added method to all methods of the previous revision that were in the same file. As there are more potential origin candidates than for the previous two heuristics, the content comparison of this heuristic is stricter, *i. e.*, it requires more similarity to avoid false positives.[3]

File-name-change-heuristic *detects origin changes due to method moves between files.* It compares an added method to all methods of the previous revision that have the same method name and the same parameter types (but are located in different files).[4]

To summarize, our origin tracking is able to detect the common refactorings (as in [22]) method rename, parameter change, method move, file rename, or file move. It is also able to detect a method rename combined with a parameter change. Only a method rename and/or parameter change combined with a file rename/move is not detected. However, based on our experience from [12], we consider this case to be unlikely

[2]For replication purposes, the thresholds used were $\theta_1 = \theta_2 = 0.5$. For the meaning of the thresholds and how to choose them, please refer to [12]

[3]Similarity thresholds $\theta_1 = 0.7$ and $\theta_2 = 0.6$

[4]We also use the higher similarity thresholds to $\theta_1 = 0.7$ and $\theta_2 = 0.6$.

as we showed that file renames rarely occur: many file moves stem from repository restructurings without content change.

C. Method Length and Threshold-based Classification

To measure method length, we use its number of statements (see Section III). We include only changes to the *method body* in its length vector: As we analyze the growth evolution of methods during modification, we do not count a move of a method or its entire file as a content modification. Excluding white spaces, empty and comment lines when counting statements, we do not count comment changes either.

Based on their length, we classify methods into three categories: Based on previous work [23] and general coding best practices [2], we consider methods with less than 30 statements to be *short*, between 30 and 75 statements to be *long*, and with more than 75 statements to be *very long*—under the assumption that a method should fit entirely onto a screen to be easy to understand. We further discuss this threshold-based approach in Section VIII.

V. Case Study Set Up

The case study targets the research questions in Section I.

Study Objects. We use ten Java systems which cover different domains, and use Subversion as version control, see Table I. Nine of them are open source, the anonymous system is industrial from an insurance company. We chose study objects that had either a minimum of five years of history or revealed a strongly growing and—in the end—large system size (*e. g.*, ConQAT and af3). The systems provide long histories ranging from 3 to 15 years and all show a growing system size trend. The number of analyzed commits in the table is usually smaller than the difference between head and initial revision because, in big repositories with multiple projects, not all commits affect the specific project under evaluation. For all systems, we analyzed the main trunk of the subversion, excluding branches.

Excluded Methods. For each study object, we excluded generated code because, very often, this code is only generated once and usually neither read nor changed manually. Hence, we assume generated entities evolve differently than manually maintained ones (or do not evolve at all if never re-generated).

We also excluded test code because we assume test code might not be kept up-to-date with the source code and, hence, might not be modified in the same way. In [24], the authors showed that test and production code evolved synchronously only in one case study object. The other two case study objects revealed phased testing (stepwise growth of testing artifacts) or testing backlogs (very little testing in early stages of development). For this paper, we did not want to risk test code distorting the results, but future work is required to examine the generalizat if our results on production code can or cannot be transfered to test code.

We excluded methods that were deleted during history and only work on methods that still exist in the head revision. In some study systems, for example, code was only temporarily added: In jabRef, *e. g.*, , antlr library code was added and removed two months later as developers decided to rather include it as a .jar file. We believe these deleted methods should not be analyzed. We also want to exclude debugging code, as this code is only used temporarily and hence, likely to be maintained differently than production code. Excluding deleted methods potentially introduces a bias because, for example, also experimental code that can be seen as part of regular development, is excluded. However, we believe that including deleted methods introduces a larger bias than excluding them.

We excluded simple getter and setter methods as we expect that these methods are not modified during history. Table I shows for each system the total number of remaining, analyzed methods as well as the percentage of yellow and red methods. Yet, the table's growth plots show the size of the code base *before* excluding deleted, getter, and setter methods. Hence, for jabref, the plot still shows the temporary small pike when the antlr library was added and deleted shortly after.

VI. Research Question Evaluation

For each research question, we describe the evaluation technique, provide quantitative results, supplement them with qualitative examples when necessary, and discuss each result.

A. Are long methods already born long? (RQ1)

We investigate whether the yellow and red methods are already yellow or red at the first commit of the method.

1) Evaluation: For each project, we count the number of methods that were already yellow or red upon their first commit relative to the total number of yellow or red methods in the head revision. Over all projects, we also calculate the average $\varnothing$ and the standard deviation σ. As the projects differ greatly in the overall number of yellow/red methods, we calculate the weighted average and the weighted standard deviation and use the number of methods per project as weight.

2) Quantitative Results: Table II shows the results: between 52% and 82% of yellow methods were already born yellow—on average about 63%. Among all red methods, between 38% and 88% with an average of 51% were born already red.

3) Discussion: More than half of the red methods are already born red and two third of the yellow methods are already born yellow. This average reveals a small standard deviation among the ten case study objects. Hence, these methods do not grow to become long but are already long from the beginning. We also investigated that green methods are already born green: in all case study objects, between 98–99% of the green methods are already born green. As our origin analysis tracks method refactorings and also code deletions within a method, we conclude that most short methods are initially committed as a short method rather than emerging from a longer method.

4) Conclusion: Most yellow/red methods are already born yellow/red rather than gradually growing into it.

B. How often is a method modified in its history? (RQ2)

We examine how many methods are frequently changed and how many remain unchanged since their initial commit.

TABLE II: Percentage of yellow/red methods that were born already yellow/red with average ∅ and standard deviation σ

ArgoUML	af3	ConQAT	jabRef	jEdit	jHotDraw	jMol	Subclipse	Vuze	Anony.	∅	σ
62%	72%	72%	64%	82%	72%	60%	67%	52%	63%	63%	10%
46%	57%	88%	58%	73%	68%	38%	48%	45%	49%	51%	9%

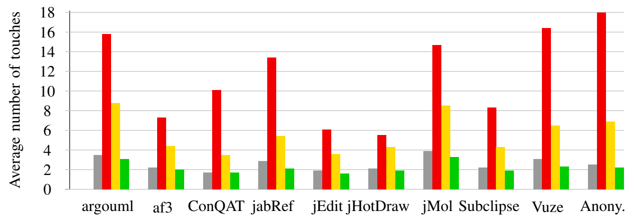


Fig. 1: Average number of touches overall (gray), to short (green), long (yellow), and very long (red) methods

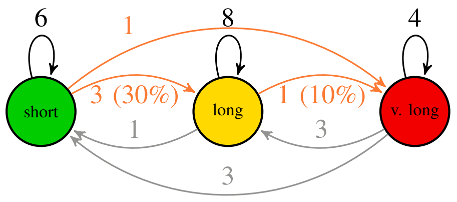


Fig. 2: Method growth (orange) and method shrinking transitions (gray) in a Markov chain

1) Evaluation: First, we evaluate the average number of touches to a method over its life time. We analyze this average for all methods and for all green, yellow, and red methods individually. Second, we compute the percentage of methods that remain unchanged.

2) Quantitative Results: Figures 1 shows the average number of touches. The results are very similar for all case study systems. On average, any method is touched 2–4 times, a green method between 2 and 3 times. This average increases on yellow and even more on red methods: Red methods are changed more often than yellow methods than green methods.

Table III shows how many methods are changed or remain unchanged after their initial commit. For each system, the table denotes the percentage of changed and unchanged methods, which sum up to 100%. The horizontal bar chart visualizes the relative distribution between changed (gray) and unchanged (black) entities. Only between 31% and 63% of the methods are touched again after initial commit. On average, 54% of all methods remain unchanged. This is the weighted average over all percentages of unchanged methods, with the number of analyzed methods from Table I as weights.

3) Examples: In ArgoUML, frequently touched red methods comprise the `main` method (167 touches) and `run` (19 touches) method in the classes `Main.java` and `Designer.java`, for example. Other methods that are frequently touched are the parsing methods for UML notations such as `parseAttribute` in `AttributeNotationUml.java` or `parseTrigger` in `TransitionNotationUml.java`. Parsing seems to be core functionality that changes frequently.

4) Discussion: When estimating where changes occur, one can make different assumptions: First, one could assume that changes are uniformly distributed over all lines of source code. Second, one could assume that they are uniformly distributed over the methods. Third, changes are maybe not uniformly distributed at all. Our results support the first assumption—under which longer methods have a higher likelihood to get changed because they contain more lines of code. We also evaluated if yellow and red methods have a longer history in our case study than green methods. This would be a very simple reason why they are changed more often. We analyzed the history length of each method, counting the days from its initial commit to the head revision. However, the average analyzed history of a yellow or red method is about the same as for a green one. Even when analyzing the average number of changes per day, the result remains the same: red methods are modified more often than yellow ones and yellow methods are more often modified than green methods.

As about half of the methods remain unchanged, this raises two questions: First, have certain components become unused such that they could be deleted? To mitigate this threat of analyzing out-dated code components, we additionally analyzed where changes occur on file-level. In contrast to the methods, we found that 91% of all files[5] are touched at least twice, 83% at least three times, still 67% at least five times. While the file analysis does not eliminate the threat completely, we yet believe it reduces its risk. Second, this also raises the question where the changes to a system actually occur. If many existing methods remain unchanged, changes either focus on only half the methods or must result in new methods. RQ4 examines this in more detail.

5) Conclusion: On average, a method gets touched between two and fours times. About half the methods remain unchanged.

C. How likely do modified methods grow and become yellow or red? (RQ3)

The results of RQ1 and RQ2 showed that many methods are already born yellow and red and many methods are not even changed. This research question examines if methods that are frequently modified grow over time or, in contrast, if their likelihood to be refactored increases when modified often.

1) Evaluation: We examine how likely a green method becomes yellow and a yellow method becomes red—analyzing the transitions between green, yellow, and red methods—and, vice versa, how likely it is refactored.

[5]This is the average over all systems with min. 80% and max. 100%

TABLE III: Percentage of changed (gray) / unchanged (black) methods

ArgoUML	af3	ConQAT	jabRef	jEdit	jHotDraw	jMol	Subclipse	Vuze	Anony.
░█	░█	░█	░█	░█	░█	░█	░█	░█	░█
62%/38%	46%/54%	31%/69%	49%/51%	35%/65%	46%/54%	63%/37%	39%/61%	48%/52%	48%/52%

Transition likelihoods in Markov model. In particular, we model the method evolution as a Markov diagram with states *green*, *yellow*, and *red*, see Figure 2. To investigate the *method growth transitions*, we examine the green $\rightarrow$ yellow, yellow $\rightarrow$ red, green $\rightarrow$ red transitions. We further examine the red $\rightarrow$ yellow, yellow $\rightarrow$ green, red $\rightarrow$ green transitions and refer to them as *method shrinking transitions*. However, strictly speaking, these transitions can not only result from refactorings, but also from code deletions within a method. Hence, the resulting *shrinking* probabilities denote only an upper bound for the real refactoring probability.

We model a transition based only on the method's state in the initial and head revision, *e. g.*, a method that was born green, grows yellow, and is refactored to green again, will lead to a transition from green to green. To estimate the likelihood of a transition, we calculate the relative frequency of the observed transitions in the histories of the case study objects: if a method was, *e. g.*, born green and ended up yellow in the head revision, then we count this method evolution as one transition from state green to yellow. Figure 2 gives a fictitious example with 30 methods in total, 10 methods each being born green, yellow, and red respectively. Among all yellow methods, 8 remained yellow till the head, one became green, one became red. Among all green methods, 6 remained green, three evolved into yellow, one into red. In this example, the relative frequency for a method to switch from green to yellow would be 30% (=$\frac{3}{10}$), to switch from yellow to red would be 10% (=$\frac{1}{10}$).

Modification-dependent Markov model. We analyze the transition likelihoods in the Markov model under four different constraints depending on the number n of touches per method.

First, we calculate the Markov model for all methods ($n \geq 1$). Then, we compute the model on the subset of methods being touched at least twice ($n \geq 2$), at least five ($n \geq 5$) and at least ten times ($n \geq 10$). We investigate whether methods that are modified often ($n \geq 10$) have a higher probability to grow than less modified ones. In contrast, one could also assume that a higher number of modifications increases the likelihood of a refactoring, *e. g.*, when developers get annoyed by the same long method so that they decide to refactor it if they have to touch it frequently. We choose these thresholds for n because five is approximately the average number of touches over all yellow methods in all study objects and ten is the smallest upper bound for the average touches to a yellow method for all systems, see Figure 1. A larger threshold than 10 would reduce the size of the method subset too much for some case study objects—making the results less representative.

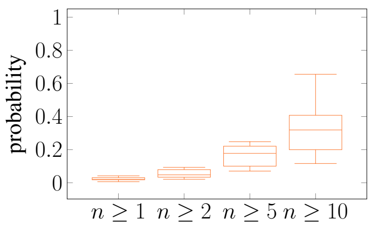
Aggregation. For each observed transition combined with each constraint on n, we aggregate the results over all case study objects by calculating the median, the 25th and 75th percentile as well as the minimum and maximum.

2) Quantitative Results: Figure 3 shows the results for method growth transitions with box-and-whisker-plots. Taking all methods into account without modification constraint ($n \geq 1$), all three growth transition probabilities are rather small. The highest probability is for the yellow $\rightarrow$ red transition with a mean of 7%. Hence, we can not observe a general increase in method length which is because many methods are not changed. However, the mean of every growth transition probability increases from $n \geq 1$ to $n \geq 10$: the more a method is modified, the more likely it grows. For the yellow $\rightarrow$ red transition, for example, the mean increases from 7% to 32%. This means that every third yellow method that is touched at least ten times will grow into a red method. For the short methods, 21% of the methods being touched at least five times grow into long methods. If short methods are touched at least ten times, they grow into a long method with 27% probability. The transition green $\rightarrow$ red occurs less often.

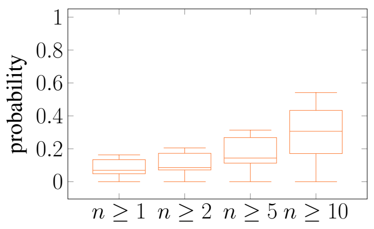
Figure 4 shows the results for the method shrinking transitions (stemming from refactorings or code deletions). For all three transitions and independent from the number of modifications, the mean is between 10% and 25%. For the red $\rightarrow$ yellow transition, this probability is between 10% and 15%, for yellow $\rightarrow$ green between 15% and 22%.

3) Examples: The method `setTarget` in `TabStyle.java` (ArgoUML) serves as a method, that grows from yellow to red while being frequently touched: It only contained 34 statements in revision 146 (year 1999). After 19 touches and four moves, it contained 82 statements in revision 16902 (year 2009). The method gradually accumulated more code for various reasons: The type of its parameter `t` got generalized from `Fig` to `Object`, requiring several additional `instanceof Fig` checks. Additionally, the method now supports an additional `targetPanel` on top of the existing `stylePanel` and it also now repaints after validating.

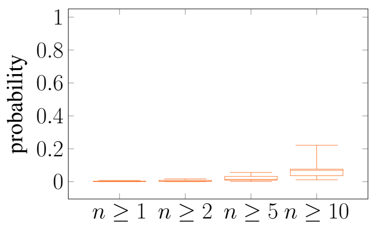
4) Discussion: This research question shows that there is a strong growth for frequently modified methods: Every third yellow method that is touched at least ten times grows into a red method and every fifth green method that is touched at least five times grows into a yellow method. However, the overall probability ($n \geq 1$) that a method grows and passes the next length threshold is very small for all three growth transitions (with a the largest mean of 7% for yellow $\rightarrow$ red). As we know that all case study systems are growing in system size (see Table I), this raises the question where the new code is added. In VI-B, we concluded that changes either affect only half of the code or result mainly in new methods. As many methods are unchanged and we do not observe a general



(a) *green* $\rightarrow$ *yellow*

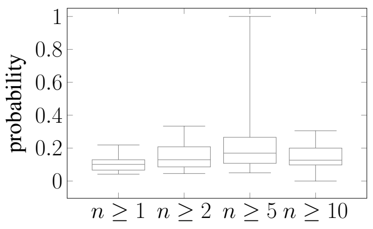


(b) *yellow* $\rightarrow$ *red*

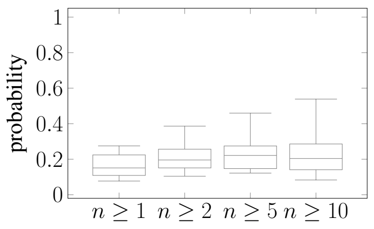


(c) *green* $\rightarrow$ *red*

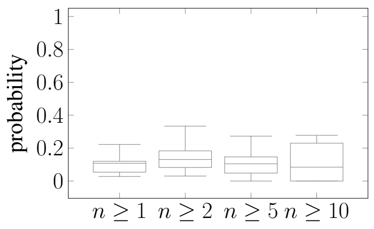
Fig. 3: Probabilities for *growth* transitions in the Markov model for methods



(a) *red* $\rightarrow$ *yellow*



(b) *yellow* $\rightarrow$ *green*



(c) *red* $\rightarrow$ *green*

Fig. 4: Probabilities for *refactorings* transitions in the Markov model for methods

growth in method length, we suspect that most changes lead to new methods. The next research question will address this.

In terms of refactorings, we cannot observe a strong correlation between the number of modifications and the mean shrinking probability. We conclude that the upper bound for the probability of method to be refactored is independent from the number of modifications it receives. In general, the probability that a method shrinks is rather low—only between every tenth and every fifth method cross the thresholds.

5) Conclusion: The more an entity is modified, the likelier it grows. In contrast to previous research questions that found no size increase for methods in general, we reveal a strong method growth for methods that are modified. However, based on our markov modeling, the number of modifications does not have an impact on the method shrinking probability. Our observed likelihood that a method is refactored is at most 25%.

D. How does a system grow—by adding new methods or by modifying existing methods? (RQ4)

As RQ2 revealed that a large percentage of methods remain unchanged after the initial commit, we aim to determine how a system grows—by primarily adding new methods or by modifying existing ones?

1) Evaluation: We use the following metric to calculate the quotient of code growth that occurs in new methods:

$$\text{growth\_quotient} = \frac{\text{code\_growth}_{\text{new}}}{\text{code\_growth}_{\text{existing}} + \text{code\_growth}_{new}}$$

With code growth, we refer to the number of added minus the number of deleted statements. Instead of using code churn (added *plus* deleted statements), we use the number of added *minus* deleted statements because we are interested in where the software grows. In case of positive code growth in new and in existing methods (more added statements than deleted statements), the growth quotient takes a value between 0 and 1 and, hence, denotes the percentage of code growth that occurs in new methods. If the growth quotient is negative, more code is deleted in existing methods than added in new methods (hence, the system got reduced in size). If the growth quotient is larger than 1, code was deleted in existing methods, but more statements were added in new methods.

To differentiate between code growth in new and in existing methods, we need to define the concept of a new method: we need to define a *time interval* in which a method is considered to be *new*. All methods that were added *after* the interval start date are considered *new methods* for this interval. For the calculation of the subsequent interval, these methods will be counted as existing methods. Whenever our origin analysis detects a method extraction, we consider the extracted method still as *existing* code as this code was not newly added to the system—it rather denotes a refactoring than a development of new code contributing to the system's grow.

We decided to use the interval lengths of one week, 6 weeks and three months. We also could have used different time intervals to define the concept of a new method and analyzed the history *e. g.*, commit-based. Hence, code growth could only occur in a new method in the commit with which the method was added to the system. An implementation of a new feature that spreads out over several commits would then result in new code growth only in the first commit. All other commits would produce code growth in existing methods. This contradicts the intention of the research question to investigate how the system grows over the time span of many years because—with this intention—the entire implementation of a new feature should result in code churn in new methods.

TABLE IV: Average growth quotient ∅ and its deviation σ on 3-months, 6-weeks, and weekly basis

		ArgoUML	af3	ConQAT	jabRef	jEdit	jHotDraw	jMol	Subclipse	Vuze	Anony.
3m	∅	85%	96%	96%	70%	72%	62%	74%	71%	76%	93%
	σ	56%	4%	3%	28%	24%	33%	37%	24%	12%	10%
6w	∅	83%	92%	95%	69%	67%	53%	75%	66%	95%	92%
	σ	100%	21%	4%	32%	38%	39%	37%	30%	209%	13%
1w	∅	91%	84%	83%	43%	45%	52%	95%	50%	59%	85%
	σ	750%	88%	142%	171%	62%	52%	231%	102%	41%	103%

2) Quantitative Results: Table IV shows the growth quotient as the average ∅ over all intervals and its standard variance σ. On a three month basis the code growth is between 62% and 96% in new methods and on the 6-weeks basis between 53% and 95%. Hence, for both interval spans, at least half of the code growth occurs in new methods rather than in existing methods. For the weekly basis, the growth quotient drops to 43% up to 95%.

3) Examples: The history of jEdit provides illustrative examples: Commit 17604 adds "the ability to choose a prefered DataFlavor when pasting text to textArea" and increases the system size by 98 SLoC—`JEditDataFlavor.java` is added (7 SLoC), `RichTextTransferable.java` is modified slightly within an existing method, and the rest of the system growth stems from the modification of `Registers.java` which grows by adding two new `paste` methods that both have a new, additional `DataFlavor` parameter. Commit 8845 adds a new layout manager by adding few lines to `VariableGridLayout` within existing methods and adding two new files, increasing the overall system size by 872 SLoC.

4) Discussion: Unsurprisingly, for the weekly time interval, the growth quotient drops for many systems as it depends on the commit and development behavior (systems in which the new feature implementations spread over weeks rather than being aggregated in one commit reveal a lower growth quotient for a shorter interval). However, based on the 6-weeks and three-months basis, we observe that—in the long run and independent from the commit behavior—software grows through new methods. These results match the outcome of the previous research questions: Most systems grow as new methods are added rather than through growth in existing methods. In particular, many methods remain unchanged after the initial commit (RQ2). Our results further match the result of [25] which showed that earlier releases of the system are no longer evolved and that most change requests to the software in their case study were perfective change requests.

5) Conclusion: Systems grow in size by adding new methods rather than by modifying existing ones.

E. How does the distribution of code in short, long, and very long methods evolve over time? (RQ5)

RQ1-RQ3 investigated the growth of a single method and RQ4 investigated the system growth. This research question examines how the overall distribution of the code over green, yellow, and red methods (*e. g.*, 50% code in green methods, 30% in yellow methods, 20% in red methods) evolves. This distribution corresponds to the probability distribution that a developer who changes a single line of code has to change a green, yellow, or red method. If the probability to touch a line in a yellow or red method increases over time, we consider this an increasing risk for maintainability—and an indicator for a required maintenance strategy to control the amount code in yellow or red methods.

1) Evaluation: After each commit, we calculate the distribution of the code over green, yellow, and red methods: we sum up the statements in all green, yellow, and red methods separately and count it relative to the overall number of statements in methods. Thus, we obtain a code distribution metric and—over the entire history—the distribution trend.

2) Quantitative Results: Figures 5 shows the results for the evolution of the code distribution in green, yellow, and red methods. On the x-axis, we denote the years of analyzed history, the y-axis denotes the code distribution. The red plot indicates the relative amount of code in red methods, the yellow plot indicates the relative amount of code in both red and yellow methods. The green lines indicate the amount of code in green, yellow, and red methods, and hence, always sums up to 100%. An increasing red or yellow plot-area indicates as a *negative trend* or an increasing risk for maintainability.

Figure 5 shows that there is no general negative trend. Only four systems (af3, jHotdraw, Subclipse, and Vuze) reveal a slight increase in the percentage of code in yellow and red methods. For ConQAT, jabref, and jEdit, the trend remains almost constant, for ArgoUML, jMol, and the industry system it even improves (partly).

3) Discussion: For systems without negative trend, we assume that the length distribution of the new methods is about the same as for the existing methods. As we have shown that there is no major length increase in existing methods, but systems grow by adding new methods instead, it is plausible that the overall code distribution remains the same.

We briefly investigated whether code reviews could impact the evolution of the overall code distribution. Based on the commit messages, ConQAT and af3 perform code reviews regularly. ConQAT shows a positive evolution trend, af3, in contrast, one of the stronger negative trends. Hence, we cannot derive any hypothesis here. For the other systems, the commit messages do not indicate a review process but future work and developer interrogation is necessary to verify this.

4) Conclusion: With respect to the evolution of the code distribution in green, yellow, and red methods, we were not able to show a general negative trend, *i. e.*, no increasing risk for maintenance. Instead, it depends on the case study object.

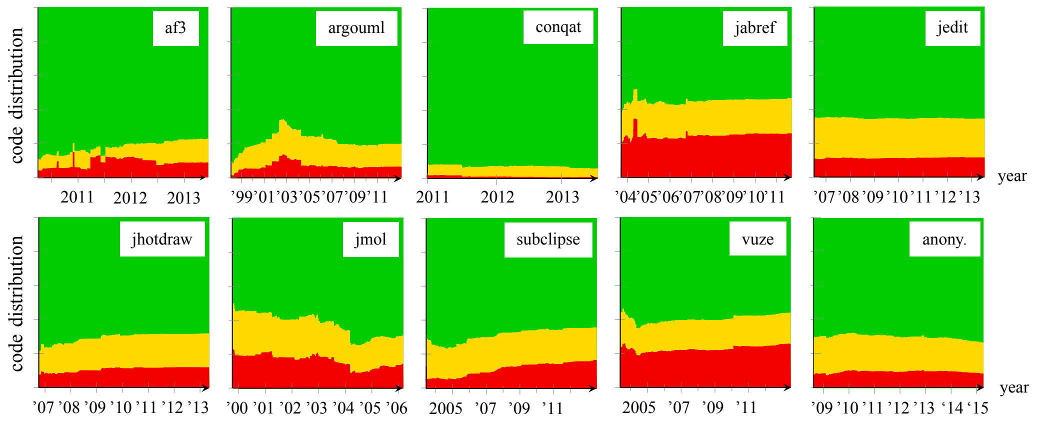


Fig. 5: Evolution of the code distribution over green, yellow, and red methods

VII. Summary and Discussion

The findings from our empirical study on software growth can be summarized as follows: Most yellow and red methods are already born yellow or red rather than gradually growing into yellow/red: Two out of three yellow methods were already yellow at their initial commit and more than half of the red method were born red. Furthermore, about half of the methods are not touched after their initial commit. On average, a method is only touched two to four times during history. This average holds for all study objects with low variance. Hence, for most methods the length classification within the thresholds is determined from the initial commit on. However, if a method is modified frequently, the length does grow: among all methods touched at least ten times, every third methods grows from a yellow to a red method. Nevertheless, over the entire system, the probability of a method to grow is low even though the system itself is growing in size. This can be explained by growth through newly added methods rather than by growth in existing methods. Finally, the distribution of the code in green, yellow, and red methods evolves differently for the case study systems. This distribution corresponds to the probability distribution that a developer has to understand and change a yellow or red method when performing a change on a random source code line. In our case study objects, we could not observe that the probability to change a yellow or red method generally increases. Generally, our case study did not show a significant difference between the industry and the open source system. However, we only analyzed one industry system. Hence, this claim cannot be generalized.

Implications. The implications of our work are based on two main results: First, most systems grow by adding new methods rather than through growth in existing methods. Second, existing methods grow significantly only if touched frequently. Hence, we derive the following recommendations—while differentiating between the benefits of short methods for readability, testability, reusability, and profile-ability on the one side, and changeability on the other side.

First, to obtain short methods that are easy to read, test, reuse and profile, developers should focus on an acceptable method length before the initial commit as we have shown that methods are unlikely to be shortened after. Even if these methods are not changed again, they will most likely still be read (when performing other changes), tested, and profiled (unless they become unmaintained code that could be deleted, of course). Given that the overall percentage of code in yellow and red methods reaches up to 50% in some case study objects (see Figure 5), the method size remains a risk factor that hampers maintainability even in the absence of further changes.

Despite knowing that many long methods are already born long, we yet do not know *why* they are born long. Do developers require pre-commit tool support to refactor already in their workspace, prior to commit? Or what other support do they need? Future work is required to obtain a better understanding.

Second, when refactoring methods to obtain a changeable code base, developers should focus on the frequently modified hot spots as we have shown that methods grow significantly when being touched frequently. When developers are modifying a frequently changed method, they have to understand and retest it anyway. Hence, refactoring it at the same time saves overhead. Furthermore, the method's change frequency over the past might be a good predictor for its change frequency in the future. However, future work is required to investigate this.

To summarize, developers should focus on frequently modified code hot spots as well as on newly added code when performing refactorings. This paper supports our hypothesis in [26], in which we showed that improving maintainability by prioritizing the maintenance defects in new code and in lately modified code is feasible in practice.

VIII. Threats to Validity and Future Work

Internal validity. Our work is based on a threshold-categorization of methods into green, yellow, and red methods. This obfuscates a growth of a method within its category, *e. g.*, a method length increase from 30 to 74 statements, affecting the results of RQ1, RQ3, RQ5. We introduced two thresholds to measure significant growth (such as a yellow-red transition) in contrast to minor length increases of few statements. Setting the thresholds remains a trade-off between obfuscating some growth versus including minor length increases. However, we used these thresholds as they have gained common acceptance in industry among hundreds of customers' systems of the software quality consulting company CQSE.

Construct validity. The validity of our results depends on the accuracy of our origin analysis. Based on a thoroughly evaluated origin analysis for files [12], the adapted origin analysis for methods was evaluated manually by the authors with a large random sample. By manual inspection, we were able to validate only the precision, *i. e.*, that the detected method refactorings were correct. We were not able to validate the recall on the case study objects. We did verify it on our artificial test data set. However, for our case study, a lower recall would mean that we do not record some method extractions but treat the extracted methods as new methods. This does not affect the results of RQ1 and RQ5. It would affect RQ2–RQ4 but previous work has shown that overall absolute number of method refactorings is not very high [22] so we assume that our results are valid. Further, we measured the method length in number of statements. One could argue that comments should be included. However, as the overall comment ratio and also the amount of commented out code varies greatly (see [27]), we believe including comments would add unnecessary noise to our data and make the results less comparable.

External validity. We conducted our empirical case study only on ten Java systems. To generalize our results, the case study needs to be extended to other OO systems as well as to test code. We have first indications that C/C++ and CSharp results are similar but they require future work to be published.

IX. Conclusion

Many software systems contain overly long methods which are hard to understand, change, test, and profile. While we technically know, *how* to refactor them, we do not know *when* we should refactor them as we had little knowledge about how long methods are created and how they evolve. In this paper, we provide empirical knowledge about how methods and the overall system grow: We show that there is no significant growth for existing methods; most overly long methods are already born so. Moreover, many methods remain unchanged during evolution: about half of them are not touched again after initial commit. Instead of growing within existing methods, most systems grow by adding new methods. However, if a method is changed frequently, it is likely to grow. Hence, when performing refactorings, developers should focus on frequently modified code hot spots as well as on newly added code.

X. Acknowledgements

We deeply thank Andy Zaidman for his excellent reviews.

References

[1] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Transaction on Software Engineering*, 2001.

[2] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 2008.

[3] N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," in *Int'l Conference on Software Engineering*, 2006.

[4] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized Code Quality Benchmarking for Improving Software Maintainability," *Software Quality Control*, vol. 20, no. 2, 2012.

[5] I. Heitlager, T. Kuipers, and J. Visser, "A Practical Model for Measuring Maintainability," in *Int'l Conference on the Quality of Information and Communications Technology*, 2007.

[6] M. Gladwell, *The Tipping Point: How Little Things Can Make a Big Difference*. Back Bay Books, 2002.

[7] E. Ammerlaan, W. Veninga, and A. Zaidman, "Old Habits Die Hard: Why Refactoring for Understandability Does Not Give Immediate Benefits," in *Int'l Conference on Software Analysis, Evolution and Reengineering*, 2015.

[8] G. Robles, I. Herraiz, D. German, and D. Izquierdo-Cortazar, "Modification and developer metrics at the function level: Metrics for the study of the evolution of a software project," in *Int'l Workshop on Emerging Trends in Software Metrics*, 2012.

[9] T. Gîrba and S. Ducasse, "Modeling history to analyze software evolution," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 3, 2006.

[10] M. W. Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study," in *Int'l Conference on Software Maintenance*, 2000.

[11] L. Schulte, H. Sajnani, and J. Czerwonka, "Active Files As a Measure of Software Maintainability," in *Int'l Conference on Software Engineering*, 2014.

[12] D. Steidl, B. Hummel, and E. Juergens, "Incremental Origin Analysis of Source Code Files," in *Working Conference on Mining Software Repositories*, 2014.

[13] M. Godfrey and Q. Tu, "Growth, Evolution, and Structural Change in Open Source Software," in *Int'l Workshop on Principles of Software Evolution*, 2001.

[14] ——, "Tracking Structural Evolution Using Origin Analysis," in *Int'l Workshop on Principles of Software Evolution*, 2002.

[15] V. Arnaoudova, L. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y.-G. Gueheneuc, "REPENT: Analyzing the Nature of Identifier Renamings," *IEEE Trans. on Software Engineering*, vol. 40, no. 5, 2014.

[16] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, 2007.

[17] F. Van Rysselberghe and S. Demeyer, "Reconstruction of Successful Software Evolution Using Clone Detection," in *Int'l Workshop on Principles of Software Evolution*, 2003.

[18] R. Peters and A. Zaidman, "Evaluating the Lifespan of Code Smells Using Software Repository Mining," in *European Conference on Software Maintenance and Reengineering*, 2012.

[19] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and Why Your Code Starts to Smell Bad," in *Int'l Conference on Software Engineering*, 2015.

[20] A. Chatzigeorgiou and A. Manakos, "Investigating the Evolution of Code Smells in Object-oriented Systems," *Innovations System Software Engineering*, vol. 10, no. 1, 2014.

[21] L. Heinemann, B. Hummel, and D. Steidl, "Teamscale: Software Quality Control in Real-Time," in *Int'l Conf. on Software Engineering*, 2014.

[22] Dig, Danny and Comertoglu, Can and Marinov, Darko and Johnson, Ralph, "Automated Detection of Refactorings in Evolving Components," in *European Conference on Object-Oriented Programming*, 2006.

[23] D. Steidl and S. Eder, "Prioritizing Maintainability Defects by Refactoring Recommendations," in *Int'l Conf. on Program Comprehension*, 2014.

[24] A. Zaidman, B. V. Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, 2011.

[25] P. Mohagheghi and R. Conradi, "An Empirical Study of Software Change: Origin, Acceptance Rate, and Functionality vs. Quality Attributes," in *Int'l Symposium on Empirical Software Engineering*, 2004.

[26] D. Steidl, F. Deissenboeck, M. Poehlmann, R. Heinke, and B. Uhink-Mergenthaler, "Continuous Software Quality Control in Practice," in *Int'l Conference on Software Maintenance and Evolution*, 2014.

[27] D. Steidl, B. Hummel, and E. Juergens, "Quality Analysis of Source Code Comments," in *Int'l Conference on Program Comprehension*, 2013.