International Journal of Mathematical, Engineering and Management Sciences
Vol. 8, No. 2, 213-229, 2023
https://doi.org/10.33889/IJMEMS.2023.8.2.013

Resource Allocation Modeling Framework to Refactor Software Design Smells

Priyanka Gupta
Department of Operational Research,
University of Delhi, Delhi-110007, India.
E-mail: pgupta1@or.du.ac.in

Adarsh Anand
Department of Operational Research,
University of Delhi, Delhi-110007, India.
Corresponding author: adarsh.anand86@gmail.com

Mohamed Arezki Mellal
LMSS, Faculty of Technology,
M'Hamed Bougara University, Boumerdes, 35000, Algeria.
E-mail: mellal.mohamed@univ-boumerdes.dz

(Received on August 02, 2022; Accepted on October 24, 2022)

Abstract
The domain to study design flaws in the software environment has created enough opportunity for the researchers. These design flaws i.e., code smells, were seen hindering the quality aspects of the software in many ways. Once detected, the segment of the software which was found to be infected with such a flaw has to be passed through some refactoring steps in order to remove it. To know about their working phenomenon in a better way, authors have innovatively talked about the smell detection mechanism using the NHPP modeling framework. Further the authors have also chosen to investigate about the amount of resources/efforts which should be allotted to various code smell categories. The authors have developed an optimization problem for the said purpose which is being validated on the real-life smell data set belonging to an open-source software system. The obtained results are in acceptable range and are justifying the applicability of the model.

Keywords- Code smells, NHPP modeling framework, Refactoring process, Resource allocation optimization problem.

1. Introduction

Coined by Kent Beck, the term “SOFTWARE CODE SMELL” has gained (Fowler, 2018) worldwide fame for its involvement in the perceived deficiencies related to design flaws existing in the software system. These smells were found affecting the software’s development process, artifacts or even the involved individuals who are using the software. There can be a case when the existence of these design flaws deliberately deteriorates the software’s quality. This relation between the smells and the quality attributes of the software makes it more important for the researchers to study these concepts and explore more about their existing cause and effect relationship.

Before going into the details about this concept, it’s necessary to understand the route by which they are introduced in the software system. It could be because of bad implementation and design choices for the system, absence of skilled code-developers, frequent software evolution, bad human resource planning and many more as described by Martini et al. (2014). Apart from these factors, priority given to enhancement in the feature set of the software instead of maintaining the quality standards has been found to be one of the major reasons behind the occurrence of these smells (Lavallee and Robillard, 2015).

Their existence in the system was seen hindering the process to maintain and evolve the software around its functional capabilities. They are the representatives of the deeper design issue lying undetected in the software (da Silva Sousa, 2016). It may be said that they are an indication for the existence of suboptimal or a poor solution (Khan and El-Attar, 2016) or even the violation of the recommended code writing practices (Suryanarayana et al., 2014). All these above mentioned attributes corresponds to the characteristics which helps in defining code smells. These described characteristics of code smells play a helpful hand in detecting smells in the software even when they are not tagged as smells. This comprehensive and evolving understanding about the classification and characteristics of smell strengthens the basic understanding about the presented concept.

These smells generally can have adverse effects on the working behaviour of the software. The relationship between software quality and the existing smells is also not hidden from anyone. The existence of smell has a multi-fold impact on the various sub-attributes of software quality namely maintainability (Bavota et al., 2012), reliability, testability (Saban´e et al., 2013), effort or cost applied, performance, productivity (Hecht et al., 2016) and even on reliability (Jaafar et al., 2013), which is the most influential attribute of software quality (Gupta et al., 2021; Verma et al., 2018). Moreover, these smells were seen to be studied under the several existing dimensions. One of them is on the basis of their effects on the software development (Mantyla et al., 2003) and other include the violation of code-development fundamentals (Ganesh et al., 2013). Apart from these, code smells can also be classified on the basis of granularity (Karwin, 2010) andartificat based approach (Moha et al., 2009). Further, Zhang et al. (2011) have emphasized on the fact that the repercussion of smells is not studied well.

Since the inception of this idea of code smells, they have been repeatedly studied my many researchers around the globe. All the research professionals along with the software organizations have tried their best to identify the catalog of these smells. For the said purpose, they have enthusiastically discovered an extensive taxonomy of smell detecting techniques. Some of them are software metrics based tools (Marinescu, 2005), heuristics based proposals (Moha et al., 2009), history based models (Palomba et al., 2014) or even the most widely used machine learning based algorithms (Maiga et al., 2012). Apart from them, literature is flooded with the proposals who have talked about various smell detecting tools. Fernandes et al. (2016) have reviewed about 84 such algorithms which play an important role in smell detection process. Researchers have also presented an approach which focused on the challenges faced by researchers while developing a smell detection technique (Rasool and Arshad, 2015).

Further, all these smell detection algorithms are nothing but the application of a tool under the concerned area. According to the literature survey, this area lacks models to justify with the detection phenomenon mathematically. Keeping eye on this, the authors have tried to address this research gap. They have drawn an analogy from fault detection phenomenon of software reliability (Singh et al., 2017) and have assumed that the mean value function corresponding to the count by which smells are being refactored in the software follows a Non- Homogeneous Poisson Process (NHPP). More justification and the utility of this concept are provided in the later parts of the article.

Moreover, it should be noted that, the distinct detection algorithms described above have helped the management and the researchers to locate several existing smells from the software system. These various smell dimensions differ from each other by the way they got introduced in the software system; their defining characteristics; in the manner they will impact the software system and many more (Wake, 2003). While going on with the literature survey, it was identified that there exist several domains and their corresponding focus area, to have some insights about the smells and study their basic characteristics. Further, smell's detection phenomenon has successfully related themselves with the

identified quality issue in the software code. From them, the authors have identified 14 such types/ categories which were sufficient enough to talk about smells in all aspects (Sharma and Spinellis, 2018). Some of them are smells related to architecture, design, usability, implementation, performance of the software system and many more. Since the motive of the proposal is to have more insights about the functional behaviour of these smells. While going into the details of 14 smell categories and relating them to the different existing smells in the software system, the authors have identified that some of the categories were related to each other and were impacting each other i.e., the categories reuse and usability are highly correlated with each other. So, to have a more focused study, the authors have clubbed them into 7 categories on the basis of the existing correlation between them, their popularity and the number of times they have been studied together. The synthesized and consolidated form of 7 smell categories is presented in the following Table 1.

Table 1. Categories related to smell.

Category	Description
I	Configuration (Sharma et al., 2016), Services (Kral and Zemlicka, 2007)
II	Test (Greiler et al., 2013), Web (Nguyen et al., 2012), Models (El-Attar and Miller, 2009), Energy (Vetr et al., 2013)
III	Reuse (Long, 2001), Usability (Almeida et al., 2015)
IV	Performance (Smith and Williams, 2000)
V	Implementation (Arnaoudovaet al., 2013)
VI	Database (Karwin, 2010)
VII	Architecture (Garcia et al., 2009), Design (Suryanarayanaet al., 2014), Aspect-Oriented (Alves et al., 2014)

Table 1 summarizes the accumulation of 14 categories identified by (Sharma and Spinellis, 2018) into the 7 consolidated ones. Considering the space constraints of the article, the description about the chosen categories can be understood using respective references. Further, Table 1 explicitly provides the reference article from which the description about any desired smell types can be studied.

Therefore, the detection phenomenon of any of the existing smell in the literature can be studied in the terms of these 7 accumulated categories. As talked earlier, there exist several detection algorithms which help the researcher with the smell detection process. On the basis of knowledge and information acquired by the researcher or the management, they are free to imply any of the existing tools. Once the detection process of smell is done, refactoration of smells comes into the picture.

Refactoration is the process via which any modulation in the software programme is done to eliminate the design related flaws from its working environment. It is said that, as soon a smell is detected, its refactoration steps are identified by the testers who then works efficiently for the smell removal. Further, to minimize the impact of smells on the facet of software's development process, its refactoration process is introduced in the software system. Many researchers have worked proficiently in this domain to refactor smells. Researchers have conducted a survey which highlights the process to refactor the existing smells (Mens and Tourwe, 2004). Al Dallalet (2015) have presented anamalgamation of several refactoring techniques that can be applied by the management to eradicate smells from the software environment. A hand book written by Fowler (2018) contains all the information one seek to know about the refactorization process of code-smells. Further, no matter how many smells are detected in the system, they need refactoring, which is the only way to remove/ avoid the possibility of any kind of miss-happening in the software system.

But the organizations must understand that "NOTHING COMES FREE" in the software environment apart from the unwanted bugs or smells. If they want to perform refactoring of the identified smells, they must talk about the amount of resources that are involved in this process. The applied resources can be in

the form of cost, human efforts or even CPU hours to name a few. It is said that the management has a specified amount of resources fixed for the refactoring process. They must have then bifurcated this allotted budget into some fractions, corresponding to their impact, occurrence, classification or anything. After this random allocation of refactoring resources, they would be able to reach to their specified goal. But, the existence of this randomness is not beneficial for any individual involved in the process. They may end up spending more resources at the place where they are not required or even land in the situation to undervalue the resources where they are needed the most. Moreover, this random allocation of resources is not going to help the developers, quality assurance personnel or even the managers to enhance the productivity of the software. This urgent need to standardize the way to allocate these maintenance efforts is addressed by the authors in this article.

For the said purpose, the authors have developed an optimization problem which deals with the allocation of refactoring resources to the 7 identified smell categories. Keeping in mind that, certain kind of smells might need more amount of resources/ efforts to refactor them, an optimization problem is presented in the article. The considered formulation maximizes the total count of smell refactored from the software in such a way that the resources allotted to them should satisfy the budget constraints. More details about the same are provide in Section 3 of this article.

Further this art of formulating an optimization problem has been widely studied in the literature in diverse fields. From the discrete resource allocation problem in the stochastic environment (Shi, 2000), to using genetic algorithm for testing – resource allocation (Dai et al., 2003) or from the modular software system (Huang and Lo, 2006) to the agile framework for software development (Anand et al., 2021), or from distributed 5G virtualized network (Halbian, 2019) to manage IOT applications (Verma et al., 2020), from urban transport management (Tao and Dui, 2022) to software patch management (Anand and Gokhale, 2020b), the applicability of this resource allocation problems can be found in every single stream existing in the research area.

To sum up this introduction section, the crux and the motive of the whole article is presented in the following manner.

Goal: Firstly, the motive of this research article is to develop the detection protocol for smells in the mathematical form following the NHPP process. Then the authors have talked about the amount of optimal refactoring resources which should be allotted to various code-smell categories such that the management can acquire the maximum utility from them.

Premise: To showcase the validity of the modeling framework, the real- life smell data set belonging to one of the releases of Azureus software system have been acquired and is worked upon. The results obtained are quite satisfactory and easily implacable by the management.

Organisation: The mathematical model for code smell refactoration process is presented in section 2 with the study design and formulated optimization programming problem in section 3. Section 4 provides a detailed description of the data visualization and result discussion in section 5. Section 6 talks about research contribution. Implications for practice is showcased in section 7 followed by conclusion and references.

2. Model Development

At the first stage, it is viable to understand the pattern via which the existing smells in the system are to be refactored by the testing team. To cater this phenomenon, the authors have drawn an analogy from the

fault removal process as explained in the field of software reliability (Anand et al., 2020a; Kapur et al., 2011). It can be said that the count of smells refactored at any time point can be studied with the help of counting process which symbolizes for the occurrence of an event at any time point.

Let $N(t)$ symbolizes for the count of occurrences of an event during the interval $[0,t]$. Let the mean-value function of $N(t)$ is represented by $cs(t)$,which represents the number of code-smells refactored till the time t.The function $N(t)$ is regarded as the Non- Homogeneous Poisson Process (NHPP), as it follows the following described properties:

a. At the beginning of the process, no smells are being refactored from the system i.e., at $t=0, N(t)=0$.
b. $\{N(t), t>0\}$ have independent increments.
c. $P\{N(t+\Delta t)-N(\Delta t)\geq 2\}=o(\Delta t)$, i.e., the chance of refactoring two or more smells simultaneously is close to 0.
d. $P\{N(t+\Delta t)-N(\Delta t)=1\}=r(t)+o(\Delta t)$, which symbolizes the probability of detecting a single smell.

It should be noted that here, $o(\Delta t)$ represents a very small quantity which approximates to 0 for smaller t and $r(t)$ is the rate by which detection of smells is performed in the software system.
Since, $cs(t)$ symbolizes for the mean value function of $N(t)$,then, it can be seen that:

$$\Pr(N(t)=k)=\frac{(cs(t))^{k}\, e^{-cs(t)}}{k!}, \qquad k=0,1,2.... \tag{1}$$

which means that $N(t)$ follows the Poisson Distribution with mean-value function $cs(t)$.

Furthermore, the presented article is based on the situation that the time scale is very large when compared to the existing code smells in the system waiting to get detected. Along with this, one of the assumptions being utilized in the article is the basic assumptions of NHPP modeling framework that the rate by which existing smells are refactored in the software is directly proportional to the remaining number of smells waiting to be refactored (Anand et al., 2018; Bhatt et al., 2017).

Tracing the same, the collective number of smells detected from the software at time point *t* can be represented as:

$$\frac{dcs(t)}{dt}=r(t)[c-cs(t)] \tag{2}$$

where c represents the total number of code smells existing in the software which are to be detected.

On solving the above presented equation along with the initial conditions $cs(t)=0$ *at* $t=0$, the eventually formed equation can be represented as:

$$cs(t)=c*H(t) \tag{3}$$

where, $H(t)$ denotes the cumulative distribution function /rate via which the smells are to be detected.

Furthermore, the authors have tried to develop a preliminary model to represent the count of smells. For this, they have assumed the nature of cumulative distribution function to be exponential in nature (Anand et al., 2019). Tracing the said assumption, the eventual equation representing the count of smells being detected from the software system is taking the following form.

$$cs(t) = c * \left(1 - e^{-r*t}\right) \quad (4)$$

where, r represents the rate by which the smells are detected.

It should be noted that the term "$cs(t)$" in equation (4) will corresponds to the total number of code smells refactored from the system at time point t .The above presented generalized equation can be converted to deal with the distinct types of code smells refactored when limited resources (z_i) are applied. The formed equation for this case will be:

$$cs_i(z_i) = c_i * \left(1 - e^{-r_i * z_i}\right) \quad (5)$$

where, $cs_i(z_i)$ be the i[th] smell type refactored from the software when z_i resources are applied.

Utility of Equation (5): The formulation presented is going to provide the management with an estimated count of i[th] type of smells which are to be refactored from the software system with the application of z_i resources. This equation holds true when the actual amount of refactored smells waiting to get refactored is c_i and the rate by which this refactoring is going to take place is r_i . Further it should be noted that, this equation considers the categorization of each type of smell due to which different amount of resources/ efforts are applied for their refactoration process.

It must be notified that no organization has the capability to invest unlimited amount of resources for the removal of smells from the system. They always want to get the maximum output with their limited in-hand resources. To cater this problem, the mathematical modeling to optimally allocate refactoring resources to different smells is presented in the next section.

3. Study Definition and Design

Continuing the above specified concepts, the fundamentals of allocation of resources are discussed here.

The goal is to optimize the available resources/ efforts applied for the refactoration of smells from the software system, which will enhance its overall efficiency and quality over its productive life cycle.

In order to reach towards the above specified target and to create better understanding and readability for the readers, the authors have divided this section into several sub-parts. These sub-parts/ sub-sections discuss the various steps involved in the resources-allocation process in details.

3.1 Variable Selection

The first step towards the optimal allocation of refactoring resources to the code smell categories is to identify the variables whose impact must be considered. A researcher or the management is free to have as many as variables in the undertaken study, but to present the preliminary model in this direction; the authors have chosen a handful of the variables from them. More details about the chosen variables and their nature are provides as below:

- **Predictor Variable:** They refer to the count of smells corresponding to every single category considered in the study along with the rate by which they are assumed to be refactored i.e., c_i and r_i respectively.
- **Response Variable:** The amount of resources/efforts which should be optimally applied for their refactoration i.e. z_i.
- **Constraint Selection:** While dealing with the problem to allocate resources; the total budget of the organization must be taken into account tries of smells should be less than the budget of the organization. It is considered that the total amount of resources which are to be bifurcated among different category.
- **Objective:** The motive is to refactor maximum number of smells of each category existing in the software system after the allotment of z_i refactoring resources i.e., Maximize $cs_i(z_i)$.

3.2 Optimization Model

From the above presented information, the basic nature of the undertaken study is made clear. Further, the mathematical equations for the above developed framework can be represented in the following manner:

$$\left.\begin{aligned} Max \sum_{i=1}^{n} c_i \left(1 - e^{-r_i z_i}\right) \\ s.t. \quad \sum_{i=1}^{n} z_i \le c_b \\ z_i \ge 0 \end{aligned}\right\} \qquad \text{(OPP-I)}$$

The outcome of the above presented Optimization Programming Problem (OPP) is going to indicate the amount of resources (z_i) which should be allocated to the i^{th} category of code smells. Instead of going with the random allocation, if the software organizations will go with these optimal allocations, they will be refactoring maximum possible smells from the system.

After these allocations, the management will have an idea about the number of smells that can be refactored from the software under the given budget and the ones which are still leftover in the system. From this number of leftover smells, the management can then decide that whether he wants to make the software more productive by allocating more resources for their refactorization or not. This will help them to make decisions about the optimal number of refactoring phases that can be done in the concerned software.

Further, more explanation about the governed optimization problem and the obtained results are explained with the help of model validation on the real-life data set, which is provided as below.

4. Data Validation

To present the working behaviour and validation of the developed optimization problem, real-life data sets have been extracted from the source (Khomh et al., 2009). More details about the data set are provided as below.

Data Set: The software considered in this study is Azureus (also known as "Vuze") which specifically allows sharing files over the internet. This is an open-source BitTorrent client written in Java language. This software has been used by over 12+ million users in the world.

When the change history of the software was analyzed it was found to be infected with different varieties of code smells. The authors have fetched the code smell data corresponding to one of the release versions of Azureus software from the (Khomh et al., 2009). It comprises of about 19082 smells belonging to 33 different types.

This data went through a manual bifurcation of smell categories on the basis of their types/ categories as provided by Sharma and Spinelllis (2018), whose description is mentioned in the introduction section. The existing 33 smells of the Azureus software was then allotted to one of the 7 identified categories on the basis of their description, characteristic and type. This has enabled the authors to study the similar types of smell in a clubbed manner, thereby increasing their interpretability and reducing the dimensionality of the considered problem.

To carry on with the analysis, the working behaviour of the developed modeling framework is now analyzed on the data set corresponding to 7 smell categories. More details about the obtained results are provided in the next section.

4.1 Study Results

First Phase of Refactoring: The authors have now reported the implications of the undertaken study on the real-life data set. The Table 2 explicitly represents the number of code smells corresponding to 7 identified categories and their refactoration rate (c_i) for one of the release versions of Azureus open-source software. Considering the total budget value for the available resources as 35000 units, the modeling framework presented in OPP-1 along with equation (5) is then executed with the help of Lingo Software (Lindo System, 1995). whose usage guidelines are provided by Bhatt et al. (2019). The obtained values for the optimal resource allocation to the identified categories are presented in Table 2.

It can be analyzed from Table 2 that, during the first phase of refactoring with 35000 units of in-hand resources, the maximum amount of 9495.535 units, was allocated to Vth category **(Implementation)** of smells. Further, it can be seen that during this refactoring phase, the highest percentage of smells remaining in system i.e., 89.26 % was corresponding to the Ist category (**Configuration and Services**) of smells for which the minimum amount of refactoring resources i.e., of 934.715 units was assigned. The lowest percentage of remaining smells i.e., 2.80 % was found corresponding to IIIrd category **(Reuse and Usability)** of code-smell.

It can be interpreted from Table 2 that the smells corresponding to IIIrd category (Reuse and Usability) were considered to be most crucial ones therefore they are refactored with the highest rate. Further, they were allocated the second highest refactoring resources i.e., of 8930.84 units, amongst the available 7 categories.

Moreover, all along the first refactoring phase, a total of 84.16% of design related flaws can be refactored whereas, around 15.83 % smells still existed in the software system.

Table 2. First phase of refactoring.

Category	Existing Number of Code Smells	r_i	z_i	No of Code Smells Refactored	No of Code Smells Refactored (Rounded off)	% of Code Smells Refactored	% of Code Smells Remaining
I	792	0.00012	**934.7153**	84.03436	85	10.73232	**89.26768**
II	570	0.00024	1984.95	216.0172	217	38.07018	61.92982
III	7561	0.0004	8930.84	7348.61	7349	97.19614	**2.803862**
IV	2504	0.00034	6778.538	2254.13	2255	90.05591	9.944089
V	2426	0.00016	**9495.535**	1895.026	1896	78.15334	21.84666
VI	1018	0.0001	1808.781	168.4412	169	16.60118	83.39882
VII	4211	0.0007	5066.641	4089.634	4090	97.12657	2.873427
Total	19082	--	35000	16055.89	16061	84.16833	15.83167

Now, it's up to the management and the software organization that whether they are satisfied with the 15.83 % of remaining smells or they still want to refactor them, to make their software more productive, error-free and highly efficient.

Second Phase of Refactoring: Here the authors have considered a scenario that the management is still interested to work on the refactoration of existing smells i.e., 15.83 % of the existing 19082 smells which is 3021 smells. They have been allocated the budget of 25000 units of refactoring resources/ efforts for the said purpose. When the optimization problem (OPP1) was executed along with equation (5) for the remaining 3021 smells, the amount of resources allocated to different categories of smell were obtained. The results are showcased in Table 3.

As done for Table 2, the information can also be analyzed and synthesized from Table 3. It was observed that with 25000 units of refactoration resources, the management can successfully refactor 1526 smells out of existing 3021 smells i.e., 50.35 % of the existing number. The highest amount of refactoring resources i.e., 7028.894 units, were allotted to category VI (Database) and the lowest resources of 1000 units to category VII (Architecture, design and aspect-orientation).

Further, it should also be highlighted that after the first phase of refactorization using the proposed modeling framework, the data (remaining number of smells waiting to be refactored) has taken the uniform shape. This has resulted in the scenario that during the second phase, the available resources were bifurcated among smell categories in such a manner that an equal proportion of smells of each category were removed from the software. This fact can be validated from Table 3 that, a uniform proportion i.e., approx. 50.44 %smells are refactored from each category of identified smells.

Table 3. Second phase of refactoring.

Category	Existing Number of Code Smells	r_i	z_i	No of Code Smells Refactored	No of Code Smells Refactored (Rounded off)	% of Code Smells Refactored in this step	% of Code Smells Remaining
I	707	0.00012	5851.52	356.6796	357	50.44973	49.55027
II	353	0.00024	2919.862	177.8398	178	50.37954	49.62046
III	212	0.0004	1754.277	106.9039	107	50.42635	49.57365
IV	249	0.00034	2058.996	125.3575	126	50.34438	49.65562
V	530	0.00016	4385.693	267.2597	268	50.42636	49.57364
VI	849	0.0001	**7028.894**	428.6155	429	50.48475	49.51525
VII	121	0.0007	**1000.758**	60.94505	61	50.36781	49.63219
Total	3021	--	25000	1523.601	1526	50.35033	49.64967

From the above presented Table 2 and 3, it can be concluded that, after the first and second round/phase of refactoring, around 1495 smells (Actual- removed in first phase of refactoring – removed in second phase of refactoring = 19082- 16061- 1526= 1495 [approximated after including the decimal places]) still exist in the system. Although, this figure of 1495 smells represents only 7.83% of the total existing smells, but still, it's the decision of the management that whether they want to continue refactoring or wants to apply those additional resources in enhancing the quality of the software by providing an upgrade full of new and up-to-date functionalities.

Further, it was analyzed that per unit cost/ resources employed for refactoration has significantly increased from the first phase to the second phase. During the first phase, 35000 units were spent in the refactoration of 16061 smells whereas along the second phase, only 1523 smells were refactored using 25000 units of resources. This simply means that per unit cost of refactoration is increasing exponentially. So, employment of resources for the third phase of refactoring may act as dead investment for the software organizations.

Third Phase of Refactoring: In order to prove this fact, the authors have executed the optimization problem (OPP-1) and equation (5) on the data set corresponding to remaining number of smells waiting in the software to get refactored after the second phase. The budget allocated for refactoring of 1495 smells is considered to be 20000 units. The obtained results are presented in the following Table 4.

It can be analyzed from Table 4 that after the third phase of refactoring, only 646 smells were refactored with an amount of 20000 units. A big percentage of 56.7893% of 1495 smells i.e., 849 smells, are still lying in the software system. Considering the enhancement in per unit refactoring cost, which can be easily seen in this phase, it is advised that the refactoring of this release version of Azureus software should be done for at max 2 phases. Allocation of resources for the removal of smells in the third phase should be completely avoided. But still, if the software organization has ample of resources with them or they consider the refactoration of smell as a very important step in building the prestige of the software, the third phase of refactoring or even its after phases will always be an open option for them. They are free to go with as many as phases as they want with their in-hand resources which is obviously going to enhance the overall software quality.

Table 4. Third phase of refactoring.

Category	Existing Number of Code Smells	r_i	z_i	No of Code Smells Refactored	No of Code Smells Refactored (Rounded off)	% of Code Smells Refactored in this step	% of Code Smells Remaining
I	350	0.00012	4684.796	150.5113	151	43.14286	56.85714
II	175	0.00024	2342.398	75.25565	76	43.42857	56.57143
III	105	0.0004	1405.439	45.15339	46	43.80952	56.19048
IV	123	0.00034	1640.825	52.59221	53	43.08943	56.91057
V	262	0.00016	3501.681	112.3835	113	43.12977	56.87023
VI	420	0.0001	**5621.755**	180.6135	181	43.09524	56.90476
VII	60	0.0007	**803.1078**	25.80193	26	43.33333	56.66667
Total	1495	--	20000	642.3115	646	43.2107	56.7893

5. Result Discussion

While going on with the analysis of the real-life data set, few questions were raised in the mind of authors, explanation of which will enhance the overall understanding about the presented concepts and will also add into the quality of the article.

Detailed explanation of the raised questions and their corresponding clarification are provided as below.

***Research Question* 1:** What is the benefit of studying the allocation of resources to existing code smells in the form of categories?

To answer this query, the developed optimization problem was executed on the data set of smells belonging to 33 different types i.e., instead of clubbing smells together in the 7 identified categories, the raw structure in which they are present in Khomh et al. (2009) was utilized and studied thoroughly. When this structure was studied, it was observed that even the resources/ efforts of 100000 units were not sufficient to refactor the 80% of the existing smells from the software system. As compared to the case when the clubbing of smells was done in 7 categories, only 35000 units of efforts were able to refactor about 84% of the existing smells (Refer Table 2).

So, grouping them together in the form of categories have not only added a new dimension in this study but have also provided an optimal way to the management via which they can increase the utility of their available resources instead of spending them extravagantly. This grouping has also guided the management to work on the refactoration process in a better way where the smells having similar attitude or behaviour are involved.

***Research Question* 2:** What is the need of going through with the refactoration process in phases, when instead of this, the management can spend the accumulated resources in one go?

The question has been raised on the utility of the presented article. If the management if free to allocate 60000 units of applied resources (35000 and 25000 for the first and second phase respectively) in one go, then is it necessary to bifurcate the resources amongst different phases. The answer to this query lies in the fact that apart from handling existing code smells in the software system, the management has several other tasks in his to do list, which may include the process of fault removal, vulnerability removal, releasing frequent patches, advertising their software or even in the development of the newer software-version and many more. With the presented approach, the management can firstly allocate a fraction of resources to refactor smells; can have a look at the working behaviour of software, then in the later hours can make decisions about the next plan of action which includes that whether they want to invest in refactoring smells or making the software more advanced as per the dynamic needs of the market.

The management can also have a look at the utility of the applied resources (per unit cost of refactoring) and then can decide about the next course of action. As explained with the data analysis part of Azureus software in the undertaken study that, the utility of applied resources has significantly decreased over a period of time. Therefore, the organizations may want to skip this process of further patch release for this refactorization. If the management have allotted the whole refactoring budget at this time, this interpretation would not have been acquired and they would be responsible for all the half-heartedly made decisions.

***Research Question* 3:** What if the allocated budget for refactoring is varied? Whether it will impact the working nature of the formulated problem or not? Whether deviations in results will be observed with the changes in allotted budget?

An interested reader may find it slightly inappropriate to fix the resource allocation budget to be 35000 units in the first phase and 25000 units in the second phase. In order to answer the asked query about the deviations in obtained results when this allocation is changed, sensitivity analysis corresponding to it is

presented. Variation in the allotted budget is made during the first and second phase of refactoring. The obtained results are presented in Table 5.

Table 5. Different phases of refactoring with variable resource constraint.

First Refactoring			Second Refactoring			Cumulative percentage of smells Refactored from the system in phase I and II
Allocated Budget	% of Smells Refactored	% of Smells Remaining	Allocated Budget	% of Smells Refactored	% of Smells Remaining	
25000	78.81773	21.18227	15000	34.31176	65.68824	86.08574
30000	81.76292	18.23708	20000	42.89296	57.10704	89.58534
35000	**84.16833**	**15.83167**	**25000**	**50.35033**	**49.64967**	**92.13963**
40000	86.2331	13.7669	30000	56.83104	43.16896	94.05697
45000	88.04109	11.95891	35000	62.46302	37.53698	95.51099

Looking at this Table 5, the management can interpret that the presented modeling framework is providing suitable results with the variation in the refactoring budget. It can be clearly inferred from the Table that, as the amount of refactoring resources are increased, the similar enhancement was followed by the percentage of smell that are refactored from the system. Seeing this table, the software management can also make decisions about the optimal number of refactoring phases that should be involved in the software depending on the number of existing design flaws in it.

***Research Question* 4:** The presented methodology also suggests that it is viable to leave some of the detected smells in the software i.e., there is no need to refactor all of the identified design-flaws. Don't you think it's harmful for the software to do so?

An enthusiastic researcher or the management is fully aware of the fact that every detected fault or vulnerability (security-flaw) is not removed from the software. Some of them always lie dormant in the software product. Management always applies its efforts in the direction to remove as much as they can to enhance the reliability of the software. But due to some unavoidable circumstances, they failed every single time to do so. Despite the presence of such faults/ vulnerability / flaws, the software systems are well accepted in the market.

Establishing an analogy in this context, some of the detected smells can also be left without refactoring in the system. So, if the management decides that after refactoring a certain number of code-smells, the want to stop this process for any particular release version, they are free to do that. Further, it's up to the management that how many resources they want to apply for the refactoration of smells and how many they want to left behind without refactoration.

To end this section, these provided questions and their explanations are expected to create a better understanding about the presented concept, the data analysis part and the corresponding implications for the betterment of the software organizations. They will guide the management in such a way which will enhance the quality of the software products.

6. Research Contribution

In the presented article, the authors have innovatively governed the process of refactoration of smell with the help of NHPP modeling framework. They have estimated the count of smells refactored from the software system at any time point t.

The authors have talked about the optimal quantity of resources/ efforts that should be applied to different code smell categories. These allocations will be done in such a manner that maximum refactoration of existing design flaws from the system is done. For the said purpose, they have backed up on the formulated optimization problem developed under the budget constraints.

Apart from this, they have explained the utility of their proposed model with the help of real-life data set belonging to Azureus open- source system. The criteria to decide the optimal number of phases for which the management should pursue with this refactoration process is also discussed in this article.

7. Implications for Practice

The article models the count of smells that can be refactored from the software system under two scenarios i.e., one at any given time point and the other corresponding to the limited resources that are applied for refactoration. Further, looking at the results analysis and related discussion presented in the article, the manual guide for the software organizations and the management is provided to help them in the optimal allocation of their in-hand resources. The authors have suggested that instead of spending extravagantly, the management should allot refactoring resources to different code smell classes/ categories using the developed optimization problem. The presented approach to allocate the resources "AS PER THE NEED", instead of the random allocation is going to be beneficial for the management is so many ways, whose description needs no introduction. Since the scarcity of resources will always be an issue for the organizations, the authors have claimed that there is no need to refactor all the existing/ detected smells from the software system. The process should be continued until the utility of the applied resources are under acceptable boundaries.

This presented algorithm for resource allocation will help the software firms which are facing resources crunch and who are not able to decide if allocating that funds to refactor the smells is going to beneficial for them or not. Using the presented methodology, they can have a check and can evaluate the one-to-one relationship between the assigned amount of resources and the fraction of smells refactored from the software system.

Along with this, the innovative resource allocation methodology for smell refactoring mechanism will guide the authors about the maximum number of refactoring phases that should be incorporated in the software. This number of phases is decided on the basis of the utility of the applied resources.

8. Future Scope and Limitations of the Study

The authors have considered that the existing smells in the vicinity of the software are refactored with the exponential rate. This assumption can be further relaxed by incorporating various types of refactoring functions in the study. Further, the authors have presented one of the ways through which resources are to be allocated to different code smells categories, in the future, different programming algorithms can be looked upon to perform the said task.

9. Conclusions

The existence of design flaws in the software environment has been creating issues for the quality aspects of the software since ages. Though there does not exist any significant relationship between these design flaws and the fault occurrence phenomenon, but still, they have always acted as a way to demolish the software quality in one or another way. The need to refactor smells as soon as they are detected was felt by the management. The amount of resources required for this action was the next question which is raised, which has not been talked about yet in the literature.

Keeping this research gap in mind, the authors have initially modeled the number of smells detected in the system waiting to get refactored with the application of NHPP modeling framework. Then they have developed the pioneer resource allocation optimization problem for refactoration of code smells from the software system. The validation of the study is done on the real-life data set of Azureus Open-source software and the obtained results are quite promising.

Conflict of Interest

The authors confirm that there is no conflict of interest for this publication.

Acknowledgements

The authors would like to thank you very much for the reviewer's comments/suggestions that improved the contents of the article.

References

Al Dallal, J. (2015). Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information and software Technology*, *58*, 231-249.

Almeida, D., Campos, J.C., Saraiva, J., & Silva, J.C. (2015, April). Towards a catalog of usability smells. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (pp. 175-181). Association for Computing Machinery, New York.

Alves, P., Figueiredo, E., Ferrari, F. (2014). Avoiding code pitfalls in aspect-oriented programming. In: Quintão Pereira, F.M. (ed) *Programming Languages*. SBLP 2014. Lecture Notes in Computer Science (Vol 8771). Springer, Cham. https://doi.org/10.1007/978-3-319-11863-5_3.

Anand, A., & Gokhale, A.A. (2020b). Impact of available resources on software patch management. In: Anand, A., Ram, M. (eds) *Systems Performance Modeling* (Vol. 4, pp. 1-12). Berlin, Boston.

Anand, A., Das, S., Singh, O., & Kumar, V. (2019, February). Resource allocation problem for multi versions of software system. In *2019 Amity International Conference on Artificial Intelligence (AICAI)* (pp. 571-576). IEEE. United Arab Emirates.

Anand, A., Gupta, P., Klochkov, Y., & Yadavalli, V.S.S. (2018). Modeling software fault removal and vulnerability detection and related patch release policy. In: Anand, A., Ram, M. (eds) *System Reliability Management* (pp. 19-34). CRC Press. Boca Raton.

Anand, A., Gupta, P., Tamura, Y., Ram, M. (2020a). Software multi up-gradation modeling based on different scenarios. In: Ram, M., Pham, H. (eds) *Advances in Reliability Analysis and its Applications* (pp. 293-305). Springer Series in Reliability Engineering. Springer, Cham. https://doi.org/10.1007/978-3-030-31375-3_8.

Anand, A., Kaur, J., Singh, O., & Ram, M. (2021). Optimal resource allocation for software development under agile framework reliability: Theory & applications, *SI 2* (64), 48-58.

Arnaoudova, V., Di Penta, M., Antoniol, G., & Guéhéneuc, Y.G. (2013, March). A new family of software anti-patterns: Linguistic anti-patterns. In *2013 17th European Conference on Software Maintenance and Reengineering* (pp. 187-196). IEEE. Genova, Italy.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., & Binkley, D. (2012, September). An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE International Conference on Software Maintenance* (ICSM) (pp. 56-65). IEEE. Trento, Italy.

Bhatt, N., Anand, A., & Aggrawal, D. (2019). Improving system reliability by optimal allocation of resources for discovering software vulnerabilities. *International Journal of Quality & Reliability Management*, *37*(6/7), 1113-1124.

Bhatt, N., Anand, A., Yadavalli, V.S.S., & Kumar, V. (2017). Modeling and characterizing software vulnerabilities. *International Journal of Mathematical, Engineering and Management Sciences*, *2*(4), 288-299.

da Silva Sousa, L. (2016, May). Spotting design problems with smell agglomerations. In *Proceedings of the 38th International Conference on Software Engineering Companion* (pp. 863-866). https://doi.org/10.1145/2889160.2889273.

Dai, Y.S., Xie, M., Poh, K.L., & Yang, B. (2003). Optimal testing-resource allocation with genetic algorithm for modular software systems. *Journal of Systems and Software*, *66*(1), 47-55.

El-Attar, M., & Miller, J. (2009). Improving the quality of use case models using antipatterns. *Software & Systems Modeling*, *9*(2), 141-160.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., & Figueiredo, E. (2016, June). A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering* (pp. 1-12). Association for Computing Machinery, New York.

Fowler, M. (2018). Refactoring: Improving the design of existing code. Addison-Wesley Professional.

Ganesh, S.G., Sharma, T., & Suryanarayana, G. (2013). Towards a principle-based classification of structural design smells. *Journal of Object Technology*, *12*(2), 1-1.

Garcia, J., Popescu, D., Edwards, G., & Medvidovic, N. (2009, March). Identifying architectural bad smells. In *2009 13th European Conference on Software Maintenance and Reengineering* (pp. 255-258). IEEE. Kaiserslautern, Germany.

Greiler, M., Van Deursen, A., & Storey, M.A. (2013, March). Automated detection of test fixture strategies and smells. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation* (pp. 322-331). IEEE. Luxembourg, Luxembourg.

Gupta, P., Anand, A., & Ram, M. (2021). Reliability as key software quality metric: a multi-criterion intuitionistic fuzzy-topsis-based analysis. *International Journal of Reliability, Quality and Safety Engineering*, *28*(06), 2140003.

Halabian, H. (2019). Distributed resource allocation optimization in 5G virtualized networks. *IEEE Journal on Selected Areas in Communications*, *37*(3), 627-642.

Hecht, G., Moha, N., & Rouvoy, R. (2016, May). An empirical study of the performance impacts of android code smells. In *Proceedings of the International Conference on Mobile Software Engineering and Systems* (pp. 59-69). ACM. https://doi.org/10.1145/2897073.2897100.

Huang, C.Y., & Lo, J.H. (2006). Optimal resource allocation for cost and reliability of modular software systems in the testing phase. *Journal of Systems and Software*, *79*(5), 653-664.

Jaafar, F., Guéhéneuc, Y.G., Hamel, S., & Khomh, F. (2013, October). Mining the relationship between anti-patterns dependencies and fault-proneness. In *2013 20th Working Conference on Reverse Engineering* (WCRE) (pp. 351-360). IEEE. Koblenz, Germany.

Kapur, P.K., Pham, H., Gupta, A., & Jha, P.C. (2011). *Software reliability assessment with OR applications*. Springer, London.

Karwin, B. (2010). *SQL antipatterns: Avoiding the pitfalls of database programming*. Pragmatic Bookshelf.

Khan, Y.A., & El-Attar, M. (2016). Using model transformation to refactor use case models based on antipatterns. *Information Systems Frontiers*, *18*(1), 171-204.

Khomh, F., Di Penta, M., & Gueheneuc, Y.G. (2009, October). An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering* (pp. 75-84). IEEE. Lille, France.

Kral, J., & Zemlicka, M. (2007, August). The most important service-oriented antipatterns. In *International Conference on Software Engineering Advances (ICSEA 2007)* (pp. 29-29). IEEE. Cap Esterel, France.

Lavallée, M., & Robillard, P.N. (2015, May). Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (Vol. 1, pp. 677-687). IEEE. Florence, Italy.

Lindo Systems. (1995). Lindo/386 5.3.

Long, J. (2001). Software reuse antipatterns. *ACM SIGSOFT Software Engineering Notes*, *26*(4), 68-76.

Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y.G., Antoniol, G., & Aïmeur, E. (2012, September). Support vector machines for anti-pattern detection. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (pp. 278-281). IEEE. Essen, Germany.

Mantyla, M., Vanhanen, J., & Lassenius, C. (2003, September). A taxonomy and an initial empirical study of bad smells in code. In Proceedings *of International Conference on Software Maintenance* (pp. 381-384). IEEE. Amsterdam, Netherlands.

Marinescu, R. (2005, September). Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)* (pp. 701-704). IEEE. Budapest, Hungary.

Martini, A., Bosch, J., & Chaudron, M. (2014, August). Architecture technical debt: Understanding causes and a qualitative model. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications* (pp. 85-92). IEEE. Verona, Italy.

Mens, T., & Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, *30*(2), 126-139.

Moha, N., Guéhéneuc, Y.G., Duchien, L., & Le Meur, A.F. (2009). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, *36*(1), 20-36.

Nguyen, H.V., Nguyen, H.A., Nguyen, T.T., Nguyen, A.T., & Nguyen, T.N. (2012, September). Detection of embedded code smells in dynamic web applications. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (pp. 282-285). IEEE. Essen, Germany.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., & De Lucia, A. (2014). Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, *41*(5), 462-489.

Rasool, G., & Arshad, Z. (2015). A review of code smell mining techniques. *Journal of Software: Evolution and Process*, *27*(11), 867-895.

Sabané, A., Di Penta, M., Antoniol, G., & Guéhéneuc, Y.G. (2013, March). A study on the relation between antipatterns and the cost of class unit testing. In *2013 17th European Conference on Software Maintenance and Reengineering* (pp. 167-176). IEEE. Genova, Italy.

Sharma, T., & Spinellis, D. (2018). A survey on software smells. *Journal of Systems and Software*, *138*, 158-173.

Sharma, T., Fragkoulis, M., & Spinellis, D. (2016, May). Does your configuration code smell?. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)* (pp. 189-200). IEEE. Austin, USA.

Shi, L. (2000). A new algorithm for stochastic discrete resource allocation optimization. *Discrete Event Dynamic Systems*, *10*(3), 271-294.

Singh, O., Anand, A., & Singh, J.N. (2017). Testing domain dependent software reliability growth models. *International Journal of Mathematical, Engineering and Management Sciences*, *2*(3), 140.

Smith, C.U., & Williams, L.G. (2000). Software performance antipatterns. In *Proceedings of the 2nd International Workshop on Software and Performance* (pp. 127-136). Santa Fe, United States.

Suryanarayana, G., Samarthyam, G., Sharma, T. (2014). *Refactoring for software design smells: Managing technical debt*. 1st Edition. Morgan Kaufmann.

Tao, Y., & Dui, H. (2022). Reliability and resource allocation and recovery of urban transportation system considering the virus transmission. *International Journal of Mathematical, Engineering and Management Sciences*, *7*(4), 476-490

Verma, R., Parihar, R.S., & Das, S. (2018). Modeling software multi up-gradations with error generation and fault severity. *International Journal of Mathematical, Engineering and Management Sciences*, *3*(4), 429-437.

Verma, S., Gupta, A., Kumar, S., Srivastava, V., & Tripathi, B.K. (2020). Resource allocation for efficient IOT application in fog computing. *International Journal of Mathematical, Engineering and Management Sciences*, *5*(6), 1312-1323.

Vetr, A., Ardito, L., Procaccianti, G., Morisio, M. (2013). Definition, implementation and validation of energy code smells: An exploratory study on an embedded system. *ThinkMin*d, 34-39

Wake, W.C. (2003). *Refactoring workbook*. 1st Edition, AddisonWesley Longman Publishing Co., Inc.

Zhang, M., Hall, T., & Baddoo, N. (2011). Code bad smells: A review of current knowledge. *Journal of Software Maintenance and Evolution: Research and Practice*, *23*(3), 179-202.