Advances in Science and Technology Research Journal

Advances in Science and Technology Research Journal, 2026, 20(3), 318–335
https://doi.org/10.12913/22998624/214373
ISSN 2299-8624, License CC-BY 4.0

Received: 2025.09.25
Accepted: 2025.11.17
Published: 2026.02.01

3D visualization of code smells: A scalable multi-level metaphoric approach for software developers

Chathuranga Hasantha[1], Thenuri Sandara Hettiarachchi[2*], Chaman Wijesiriwardana[1]

[1] Faculty of Information Technology, University of Moratuwa, Sri Lanka
[2] School of Computing, University of Staffordshire, London, United Kingdom
* Corresponding author's e-mail: thenurih@apiit.lk

ABSTRACT

Code smells are indicators of poor software design or implementation choices that hinder software maintainability, performance, and overall software quality. Manual identification of code smells is time-consuming and resource intensive. Detection tools have their own drawbacks, including false positives, scalability issues, a lack of context, inadequate coverage, and poor usability. Furthermore, existing code smell visualization models are limited by abstraction levels, scalability issues, reliance on manual processes, and lack of empirical studies, which limit developers' ability to make informed decisions efficiently. To fill this gap, this paper introduces a novel 3D visualization model that combines static analysis with `island` and `city` metaphors to represent classes, methods, and their relationships, along with associated code smells. The model was applied to a software project, and the usability of the model was evaluated in a pilot study. The findings from this study demonstrate that the proposed model provides an intuitive visualization of code smells than traditional tools, thereby supporting developers' decision-making and improving program comprehension.

Keywords: code smell, code smell detection, detection approaches, current trends.

INTRODUCTION

Code smells are symptoms of poor design and implementation decisions that have a direct impact on the overall quality of the software [1,2]. These bad decisions lead to issues such as maintainability, performance, and overall software quality [3], which can have significant long-term consequences [4,5]. Code smells, such as complex classes and methods [6,7], compromise both the readability and efficiency of the code, which ultimately reduces sustainability and increases the maintenance burden [8]. The presence of code smells is almost unavoidable in software development due to several factors, such as time constraints, lack of experience, and the growing complexity of modern software systems. Developers often face tight deadlines, leaving little room to address these underlying issues during the development process. Despite the development of various code smell detection tools, developers still face challenges in understanding, maintaining, and evolving complex software systems, which has led to a growing interest in visualization-based approaches [9]. Although several 3D visualization models have been proposed, they still fall short of meeting developer needs. In particular, current models do not provide the ability to customize and visualize code smells across multiple abstraction levels, including class, method, and inside-method views. Moreover, these tools often lack interactivity, limiting developers' ability to navigate, zoom, rotate, and explore the visualized structures, thereby reducing their effectiveness in supporting flexible and intuitive code comprehension [10,11].

Motivation scenario

Rob, a programmer working on an accounting system, has been assigned tasks to add new features, including generating a customer details

summary and formatting phone numbers. Before implementing these features, Rob needs to go through the existing code, read comments, and consult software documentation. If the issues are unresolved, he must seek help from a senior programmer to resolve the code-related difficulties. When adding the new features, Rob can follow Fowler's 22 code smells [12]. As a guide to identify issues during coding, but this can lead to errors. Even though Current tools like JDeodorant [9,10], iPlasma, InFusion [9,11], have been introduced to identify and refactor code smells. These tools are limited in their ability to detect only a few smells and do not support visualization, which enables to visualize the smells in a flexible manner. This often leads to false positives and a lack of real-time feedback [9]. To address these challenges, a 3D visualization model can offer a more intuitive, real-time way to detect issues across different abstraction levels.

As a solution, this paper presents a novel interactive 3D visualization model designed to improve code smell visualization at different abstraction levels. The proposed model integrates island and city metaphors to visualize code smells at different levels of abstraction, aiming to enhance program comprehension and assist developers in identifying problematic code efficiently. Unlike existing tools, this model also provides evolutionary context, helping developers understand the historical development of code smells, which supports better refactoring decisions. The approach addresses limitations of current tools, such as visualizing smells in different abstraction levels, false positives, and a lack of interactivity [13].

RELATED WORK

This literature review offers a comprehensive overview of code-smell visualizations, exploring their applications in software development, the methodologies used to implement them, and the potential limitations that may affect their effectiveness in real-world scenarios. By synthesizing insights from prior work, this section clarifies where current approaches fall short and positions the need for richer, multi-level visualizations such as the one proposed in this study.

Code smells visualization approaches

Today's software systems are increasingly large and complex, and many people collaborate in their development and maintenance [14]. This makes it more and more difficult to program, understand, and modify the software tasks, especially when working on the code of other people. Therefore, tools for supporting these tasks have become essential [15].

An approach for the automatic detection presented and visualization of code smells, and discuss how this approach can be used in the design of a software inspection tool [16]. There is an illustration of the feasibility of their approach with the development of jCOSMO, a prototype code smell browser that detects and visualizes code smells in JAVA source code. While this tool contributes a valuable idea, it typically supports limited structural levels (methods, classes, packages) and lacks richer abstractions for architectural or system-level analysis.

Murphy [17] presents the value of versatility, expressivity, and context-sensitivity when showcasing smells, and proposes a mock-up model of a detection tool using the above properties. Various tools have been developed to help developers inspect the quality of source code. A code smell detector that uses an interactive ambient visualization to make programmers aware of smells and make informed, confident refactoring judgments. This paper suggests that such tools have a place in the software developer's toolkit. Although code visualization tools are increasingly applied to support code smell detection, they have limited module structures, such as methods, classes, and packages.

Researches used multiple visualizations to detect bad smells [18]. They present four views with concern properties: package-class method structure, inheritance structure, dependency graph, and dependencies-weighted graph, and study to assess the extent to which visual views support code smell detection. These work showcases the importance of multi-perspective views when visualizing bad smells, but still fall short of offering unified, intuitive metaphors for smell visualization across abstraction levels.

Developed a semi-automatic detection approach that combines automatic pre-processing and visual representation and analysis of data [19]. It is complementary to automatic approaches for anomalies whose detection requires

knowledge that cannot be easily extracted from the code directly. The detection is seen as an inspection activity supported by a visualization tool that displays large programs (thousands of classes) and allows the analyst to navigate at different levels of the code. In this approach, they specifically target anomalies that are difficult to detect automatically.

Visualization tools

The manual code smell detection has several drawbacks, such as being time-consuming, non-repeatable, and does not scale and the approach that code smells are detected by humans has not been thoroughly explored yet [20]. And also, detection in large systems is a very time and resource-consuming, and error-prone activity [4] because smells cut across classes and methods, and their descriptions leave much room for interpretation. Therefore, the software industry needs effective and practical tools to scaffold the process of maintaining quality software.

The relationship between the class error probability and bad smells based on three versions of the Eclipse project, and the result showed that classes that are infected with the code smell Shotgun Surgery, God Class, or God Methods have a higher class error probability than non-infected classes [21].

The manual detection of design flaws showed more familiar with a software system, their ability to objectively evaluate it and spot design flaws decreases [22].

Dhambri et al [23] developed a 3D software visualization model to detect design abnormalities. The model uses both quantitative data based on metrics and structural data based on connections among modules. There is an improvement needed in performance variability to reduce this variability. This approach is a manual process and needs to be combined with an automatic one in another direction in their investigations.

An interactive ambient visualization novel smell detector implemented for programmers. This model is used to help programmers identify code smells and refactoring judgments [24]. This is discussed, has been tested with limited code smells.

A multiple views approach based on concern-driven software visualization resources to effectively spot code smells. The approach relies on interactive visual abstractions of source code to support a concern-sensitive analysis based on different views [25]. Evaluation of this approach is made for small criteria, and it is not sufficient for large projects. The visual analysis could be a more detailed and effective analysis of the proposed approach, depending on the appropriate assignment of concerns to source code. Despite these advancements, these tools rarely integrate multiple abstraction levels into a single coherent metaphor.

Silva et al. [26] developed an interactive visualization to help developers identify Code Smells called VISMELLS. The survey's data concluded that VISMELLS can facilitate the discovery of Code Smells. Visualization of values from software metrics used to detect Code Smells is also enabled in VISMELLS.

Mumtaz [18] demonstrated an approach to analyze multivariate object-oriented software metrics to detect outliers, which could be connected to bad smells in the context of software quality. As a result, this approach helps visually identify the data elements as bad smells, which are also perceived as outliers in the linked visualizations. The automatic detection of bad smells is dependent on the published detection rules, and this approach is not yet tested by users other than the authors.

Software visualization is used to visualize the code smells in the program. Hammad, et al [16] proposed a visualization approach and shows classes as buildings and bad smells as letter avatars based on the initials of the names of bad smells. These avatars are shown as warning signs on the buildings. A framework is proposed to automatically analyze code to identify bad smells and to generate the proposed visualizations. The evaluation of the proposed visualizations showed that they reduce the comprehension time needed to understand bad smells.

Island metaphor visualization technique is used to emphasize the modular aspects of OSGi, and an interaction technique is implemented to preserve user comfort while inspecting large software systems. This approach is used for exploring OSGi-based software systems in virtual reality. This approach needs to be applied for practicability in aiding software comprehension tasks [16].

Evaluation of the impact of the medium on the effectiveness of 3D software visualizations is the most important fact. They experimented on the 3D city visualization technique that has proven effective for software comprehension tasks. Currently, they have applied this to a standard

computer screen (SCS), an immersive 3D environment (I3D), and a physical 3D printed model (P3D). Further, they want to investigate the impact of media used for collaborative visualization, such as wall displays, multi-touch tables, etc.

While current tools show the utility of visualization, there are no integrated, scalable, and multi-level visualizations that allow developers to study code smells in their entirety rather than fragmented perspectives.

Limitations

Although significant progress has been made in developing visualization approaches for code smell detection, several key limitations remain. First, many tools restrict their analysis to limited abstraction levels, such as methods, classes, or packages, thereby overlooking higher-level architectural smells that impact overall system design [8,25]. This narrow focus reduces the applicability of these tools in large-scale, real-world projects where multi-level analysis is critical. Second, scalability remains a recurring obstacle. While some approaches demonstrate promising results on small or medium-sized systems, few have been evaluated on large industrial projects [23,26]. The computational and cognitive load of analyzing thousands of classes or modules often overwhelms the visualization techniques, limiting their utility for practitioners. Third, most studies rely heavily on manual or semi-automatic processes. Even when automated metrics support detection, the interpretation of results is often left to developers, reintroducing the same subjectivity and inconsistency that visualization was intended to mitigate [20]. This lack of full automation means that many tools are not yet mature enough for integration into standard development pipelines. Finally, empirical evaluations of visualization approaches remain limited. Many studies are restricted to small case studies, controlled experiments, or academic prototypes, which undermines generalizability [18,27]. Taken together, these gaps indicate a clear need for more comprehensive visualization techniques that combine automation with multi-level, scalable, and interactive representations. Addressing these challenges will not only enhance code comprehension but also improve the reliability of refactoring decisions, making visualization a critical research direction in software engineering.

Proposed approach

This section proposes a novel visualization model designed to detect and analyze code smells using two intuitive metaphors: the Island Metaphor [28,29] and the City Metaphor [30,31]. These metaphors provide a structured way to explore large software systems [32], depicting code structures and potential smells at both the class and method levels. The model itself does not directly detect code smells but integrates SonarQube's output, which identifies code smells like Long Method, Feature Envy, God Class, Large Class, Duplicate code, Long Parameter List [33]. By using SonarQube's capabilities, this model provides an interactive 3D representation of these code smells. This approach enables developers to gain deeper insights into the code base through visually augmented models.

Class level view

A novel approach to visualize code smells at the class level is introduced through the Island Metaphor. This metaphor represents decoupled entities [34] in a software system, enabling a simplified way to visualize the complex code structures [28,29]. In this metaphor, each software system is represented as an ocean, while individual classes within the packages are represented as islands.

Method level view

At the method level, the model employs the city metaphor, which provides a deeper insight into the internal structure of each island [31,32]. Which means this visualization technique is used to describe the methods and the attributes. In the city metaphor, methods are represented as buildings, variables are depicted as people, and input parameters are shown as small squares (windows) on the block. The number and height of the buildings reflect the content of the methods, such as the number of lines or the complexity of the code. In this view, the code smells are also highlighted in red.

Figure 1 depicts a detailed visualization of the city metaphor, offering a better understanding of the content described above. This metaphor is interactive and can be navigated using the keyboard, allowing developers to extract a large number of methods inside a single class. By isolating these elements as building blocks, the city metaphor helps highlight areas that may require

more attention. As a result, the proposed model offers a visual view that depicts the overall evolutionary characteristics of packages, classes, and methods, including the identified code smells (Figure 2).

Identification and analysis of code smells

Beyond the basic visualization, the model provides a detailed analysis to help developers understand the nature of code smells. If a developer clicks on a selected class or method object at the abstraction level, a summary message window appears, displaying key metrics such as the class name, number of attributes, number of lines of code, number of methods, and affected methods.

Whiteboard view and visual feedback

In addition to the massage window, the model provides an inside view of a building, which includes an interactive whiteboard and analytical charts for a selected method. The whiteboard displays a summary of each detected smell type, such as long parameter lists, feature envy, and duplicate methods. Furthermore, this includes attributes, analytical data of the code smells through charts, and a summary of the suspicious code snippets. The analytical data that are represented in these charts are based on the information collected throughout the visualization process, offering a complete view of the code quality.

Figure 4 illustrates the whiteboard view, where identified code smells are listed. This gives the developers an idea of the code base that needs to be improved.

Final code snippets and refactoring suggestions

Once the code smells are identified, developers can view the actual code snippets that are affected by code smells. The final step bridges the gap between the visual representation and actionable code refactoring.

Data extraction and object mapping

To generate these visualizations, numerous metrics need to be abstracted from the source code. These metrics help to determine the size of the visual element. Table 1 summarizes the object mapping of the corresponding visualization techniques: island view and city view. Showing how each software component is displayed within the visualization model.

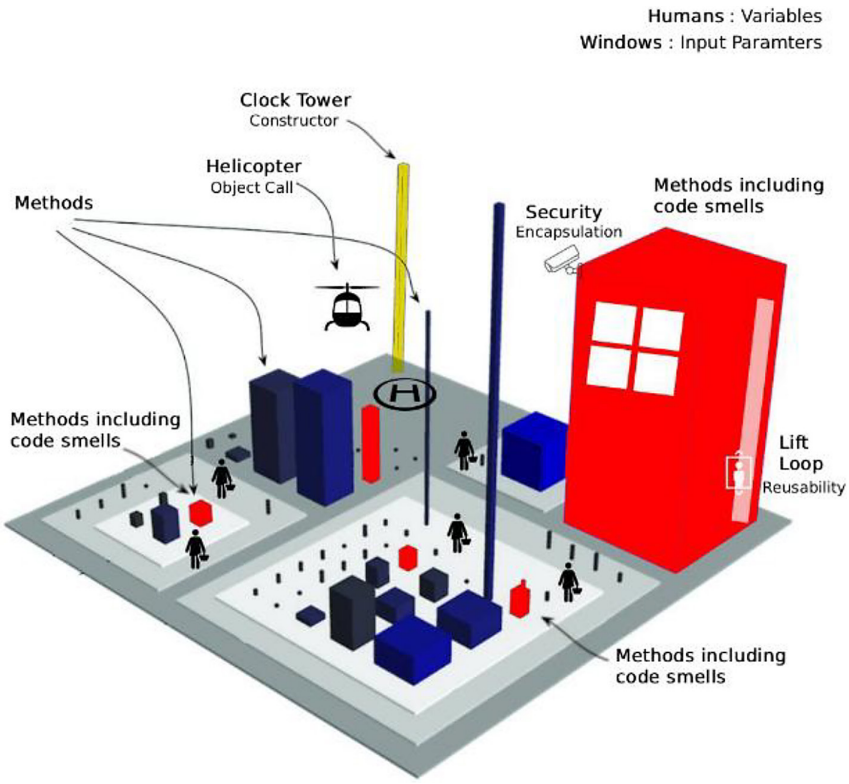

Figure 1. Visual representation of city metaphor

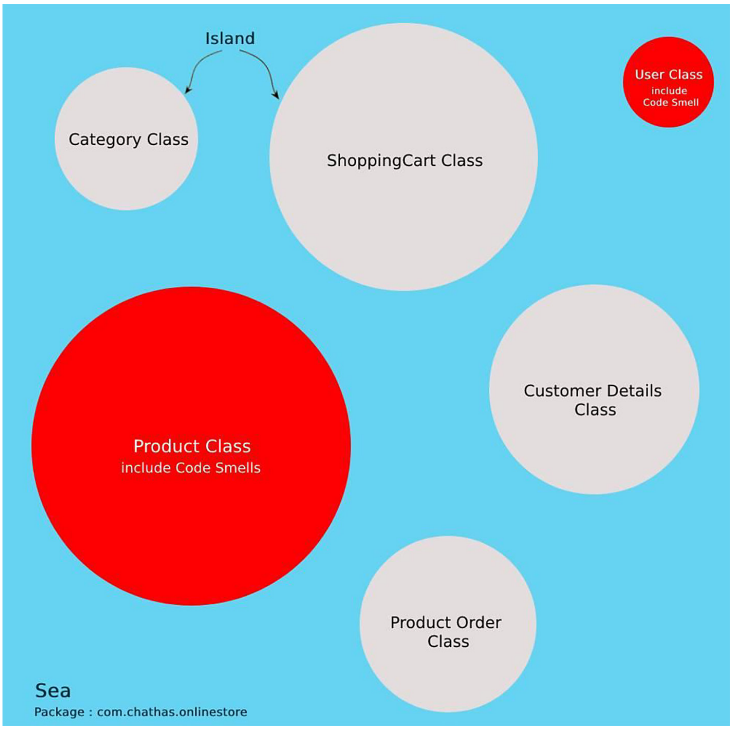

Figure 2. Visual representation of Island metaphor

PROOF-OF-CONCEPT IMPLEMENTATION

This section showcases the proof-of-concept implementation of the 3D visualization model. The goal here is to transform the abstract code smell information into dynamic, multi-tiered visualizations that support developers in identifying and analyzing design issues. Before the implementation process, a detailed model overview is provided to help readers understand the main components of the model and how they interact. The implementation stage is divided into three sections: (i) extracting a dataset from a selected software project, (ii) visualizing code smells across three abstraction levels (class, method, and inside-method), and (iii) applying an algorithm to avoid object overlapping for a clear and meaningful layout (Figure 3).

System design and model overview

The tool is designed based on established visualization techniques and code smell detection approaches. The key purpose of this model is to automatically visualize the source code, highlighting areas affected by code smells, and visualize it in a 3D environment. This approach has been achieved by importing semantically structured JSON data generated from the static analysis of the chosen software project.

The proposed tool's function is based on a set of rules; these rules handle specific code constructs. The tool checks if these rules are being violated during the analysis process, and any violations are flagged as code smells. This approach is beneficial for developers when identifying potential design issues that might have been missed or ignored. Using this tool, they can receive early warnings, even for smells introduced intentionally.

The tool is implemented using PHP (CodeIgniter) for server-side logic, the Babylon.js JavaScript library for interactive 3D rendering, and MySQL for database management. This makes the platform independent and deployable on any web hosting platform without any third-party dependencies.

- PHP (CodeIgniter) was selected for its lightweight, secure, and scalable server-side framework, providing an efficient backend for handling large datasets generated during static code analysis.
- Babylon.js enables the creation of interactive 3D environments that support real-time navigation and manipulation of visualized code smells, which aligns to provide an intuitive and dynamic visualization experience for developers.
- MySQL was chosen for database management due to its reliability and efficient querying

Table 1. Object mapping 3D visualization model

Attribute	Mapping object
Classes	Island
Lines of code in class	Perimeter of a cylinder
Packages	Sea
Pie chart view in the island metaphor	Numerical proportion of code smell considering the source code
Left sidebar navigation	Classes and methods included in the generated model
Right sidebar	Details of the number of issues, severity, time, and effort needed to fix the issue
Methods / functions	Building block
Attributes / variables	Left whiteboard inside building
Input parameters	Blocks on the floor, classroom
Lines of code	Building Height
Inside of method	Classroom
Pie chart	Illustrates the numerical proportion of code smells by considering the source code
Bar chart	Types of code smells found in the source code
Whiteboard	Suggestions to fix code smells
View graph button	Illustrate the numerical proportion of code smell by considering the source code in the method

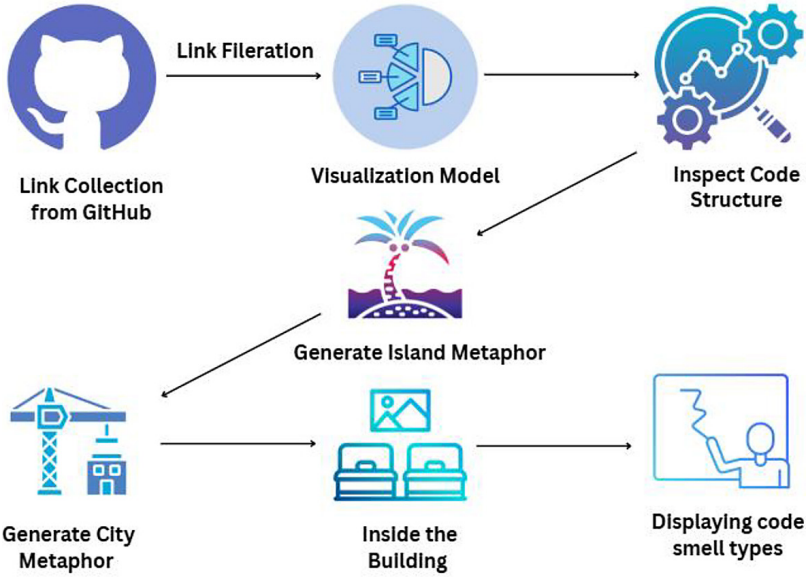

Figure 3. Implementation cycle

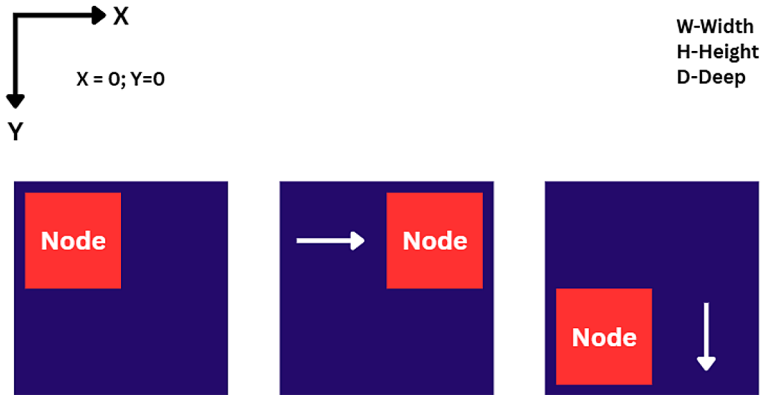

Figure 4. Object moving to avoid overlapping

capabilities, allowing for smooth handling of large-scale software project data and ensuring scalability for future applications.

The output is an interactive, browser-based 3D visualization model, allowing developers to inspect code smells at multiple abstraction levels. The systems interactive nature allows developers to navigate through the visualized code structures, facilitating a deep understanding of the code base and supporting better decision-making.

Dataset extraction from selected software project

The process of preparing the input dataset consists of two steps: The first step is selecting or importing a project found through search engines, online communities, and software repositories. The second step is formatting the raw analysis data into a predefined JSON format that serves as input for the visualization engines. The extracted, formatted data are stored as a JSON meta-model. The meta-model consists of class names, line counts, methods, parameters, code smell types, and other metadata. This is useful for meaningful visualizations.

Visualization models for three abstraction levels

The 3D visualization model is developed using open-source technologies, offering interactive zooming, rotation, and navigation capabilities [34]. The model uses a one-to-one mapping approach and employs unit visualizations at each level of abstraction.

The main visual representation of the proposed approach consists of islands, buildings, and interior spaces. Islands have emerged from the sea and are represented by gray-colored surfaces, while a blue-colored background represents a package. Building blocks represent methods, and the gray-colored ground surface represents the base class. Parameters are mapped to small boxes on the floor, and the whiteboard represents code smell types, snippets, suggestions, and local variables inside the method.

The perimeter of the cylinder and the height of a building block correspond to the number of lines of code. The colors of each object – dark red, light red, yellow, and green – represent the severity of the code smells. The small green boxes on the floor inside the building represent the parameters declared in the method, while the whiteboard on the wall inside the building represents the code snippet, code smell types, local variables, and suggestions.

Algorithms to avoid overlapping objects

Object placement on the canvas is crucial for generating the visualizations. This process aims to provide a more convenient solution for placing objects on the canvas without overlapping. An algorithm was applied to avoid the overlapping of visual objects at all abstraction levels. The algorithm works as follows: First, retrieve the class array from the JSON request, pick the first class, and place it on the canvas. After detecting the first x and y coordinates and the size of the class, calculate the next coordinates using the first-class object and a constant value. Iterate this process while considering the canvas width. After the first row on the canvas ends, the visual object's coordinates need to move to the next row.

According to this process, take the first method/class block and check whether there is enough space to place it on the root. If there is enough space, the object is placed in the upper-left corner. Then, the space in the root will be divided into two parts: the right node and the left node. When the second block is processed, it starts from the root to check for available spaces. Since the upper-left corner is utilized, check the spaces to its right and below. If that block is placed, the space is split into two. This process continues until every block is placed inside the canvas.

The root canvas has to be grown based on the method/class size. First, we equalize the root size to the size of the largest method. Then, the algorithm starts as before. The first block will be placed without any issue since the root is equal to its size. From the second block onward, the root has to be resized.

Code smells visualization using a Novel 3D model

The final stage of the tool generates 3D visualizations for buildings, islands, and inside-building views according to this dataset. Zooming, localization, and browsing are essential features that are under consideration. Developers will have the ability to search for and locate a specific building that corresponds to a specific class. They will also be able to zoom in or out of the buildings. The tips, navigation, and summary graphs

feature helps developers navigate through buildings in the 3D environment and easily understand large-scale systems with many classes.

In the proposed approach, individual island groups can form archipelagos, providing the first abstraction level and the user interface of the island-view prototype. Figure 4 illustrates the automatically generated class-level model created using the imported JSON request. The perimeter of the cylinder island view varies based on the number of code lines included in the class. Perimeters highlighted using dark red are meant to include code smells in the classes or any other lower-level member that contains code smells.

Figure 5 shows the message box appearing after clicking on each class. It includes details such as the class name, the number of code lines in the class, the code smells present in the class or lower member levels, and a link to the next abstraction level.

Figure 6 illustrates the next abstraction level using the city metaphor. The method level shows the building block visual objects related to the methods inside the class. The height of the building blocks varies based on dynamic values updated in the database. This height is calculated based on the number of code lines in the method. Each block is shown in a different dark color on the ground to represent the class if it has a bad smell related to the method. The severity of the bad smell determines the color, with darker colors indicating higher severity. The severity levels are critical, major, minor, and informational (info).

Figure 7 shows the message box that appears after clicking on each method. It includes details such as the type of code smell, a code snippet, a detailed severity level, and a link to the next abstraction level. The building shown in Figure 8 represents the model. It consists of three walls, each with more details. The number of boxes on the floor represents the number of parameters for each method. The type of code smell and error code is shown on the whiteboard on the main wall, while the right-side wall provides suggestions and tips to solve these code smell issues.

We adapted the pie chart visual paradigm, using colors and portions to represent code smells and clean code that are affected by a specific concern. Figure 9 illustrates how concerns are represented in the pie chart. The portion colored in dark red corresponds to the percentage of code smells in methods that are affected by a specific concern.

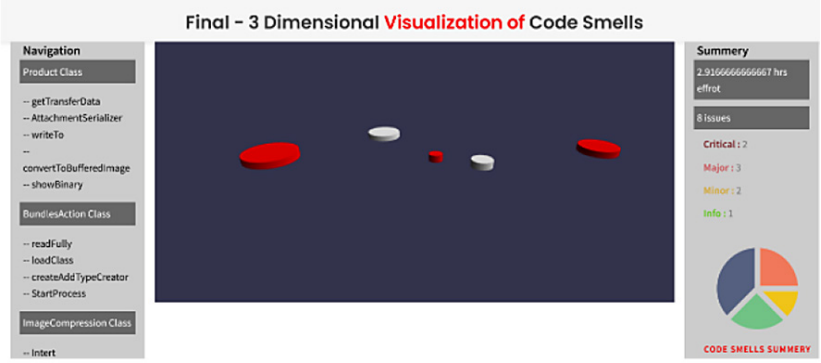

Figure 5. 3D model for island metaphor

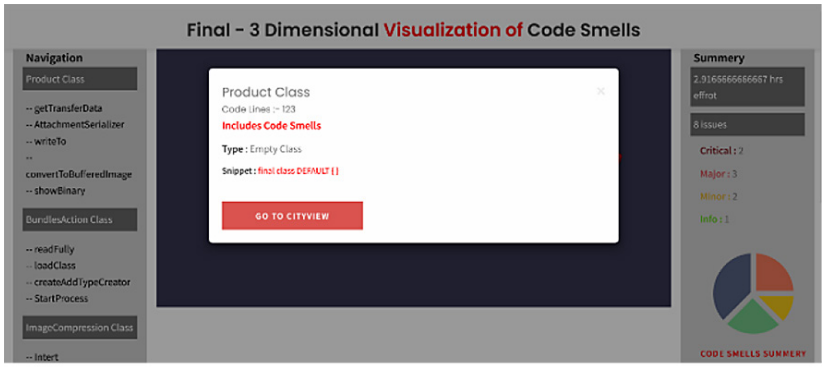

Figure 6. Message box with details of the class in the island metaphor

In this figure, we can see supportive links (red color links), issue status (critical, major, minor, or info), the method you selected, and its parent class (left sidebar in Figure 9).

The Figures 4–9 illustrate the navigation of the entire process of the proposed model's implementation.

Visualizing evolutionary aspects of code smells

The above images (Figures 11 and 12) illustrate how the model visualizes the evolution of code smells across abstraction levels. In Version 1 (Figure 11), multiple methods exhibit severe smells, represented by tall red blocks, while in Version 2 (Figure 12), after refactoring, these smells are visibly reduced in severity and complexity. Each 3D element dynamically mirrors these changes over time, making it easy for developers to track growing or diminishing code smells. By showcasing these trends visually, the model helps developers prioritize refactoring based on historical patterns rather than relying on textual descriptions or static snapshots. This leads to more informed and proactive maintenance decisions.

Limitations of the proposed approach

Despite the strong performance of the proposed model, several limitations exist. The approach focuses mainly on class, method, and inside-method levels, without addressing higher-level architectural relationships such as dependencies between packages or modules. The model's industrial-scale systems have yet to be empirically validated due to the high number of classes. Table 2 provides evidential support for this statement. The table illustrates the scalability of the proposed model across various sizes of software projects: small, medium, and large-scale. The performance of the model shows successful visualizations for the small and medium-sized projects, but unsuccessful visualizations for the large-scale project with over 1.6 million lines of code. Moreover, the visualization relies on SonarQube outputs for code smell detection, making it dependent on external tools rather than performing intrinsic analysis. Finally, the evaluation was limited to a pilot study with eleven participants, indicating the need for broader empirical validation using larger datasets and diverse practitioner groups to enhance generalizability.

EVALUATION

Comprehensive evaluation of the proposed model

The primary objective of this evaluation is to assess the usability and effectiveness of the proposed 3D visualisation model by determining whether it facilitates developers in identifying and understanding code smells in software projects. The evaluation was conducted through a pilot experiment, which involved researchers and experts in the software field (software engineers, senior software engineers, tech leads, QA, and database administrators). The group of participants consisted of thirteen practitioners with up to 4 years of experience, four with up to 7 years of experience, and four with more than 8 years of professional experience. The primary objective is to assess how effectively the visualisation model can identify and facilitate understanding of bad smells in code.

For this evaluation, the model was tested using a sample software project selected from SonalCloud. The projects were pre-analyzed to

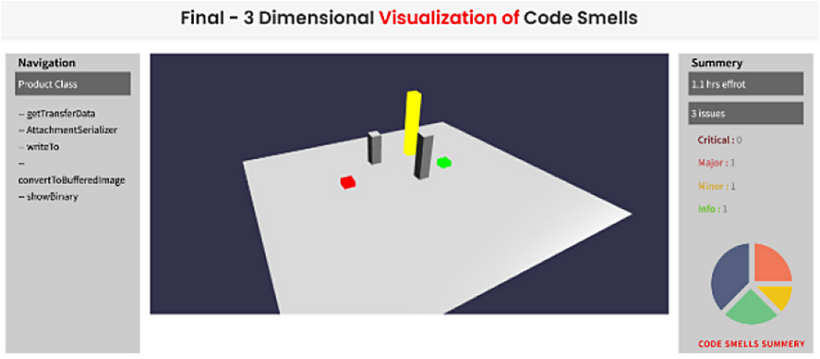

Figure 7. 3D model for city metaphor

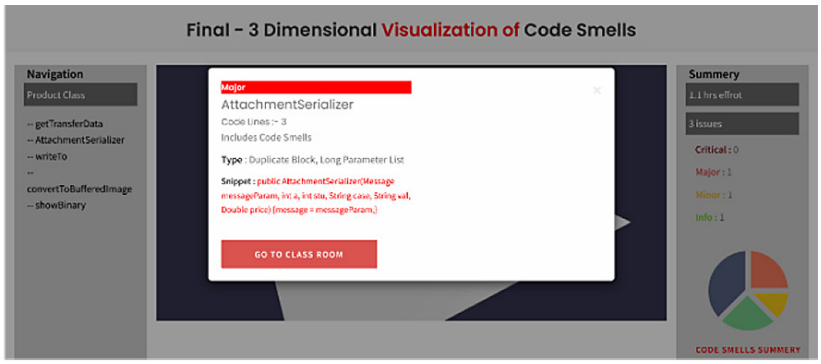

Figure 8. Message box with details of the method in the city metaphor

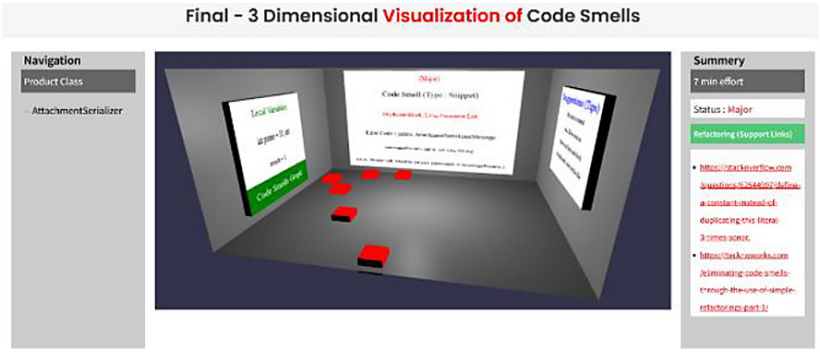

Figure 9. 3D model for inside building

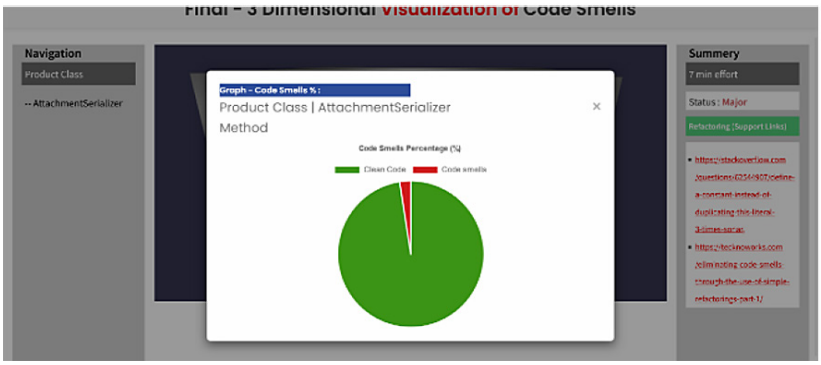

Figure 10. Code smells percentage included in the method

identify code smells. The selected GitHub project results were formatted into a JSON input request and uploaded to the model. The model generated 3D visualizations for the relevant project, including different abstraction levels: the island metaphor (for class-level view), city metaphor (for method-level view), and detailed internal views (such as classroom and whiteboard views).

The evaluation was designed with the following objectives:

- Ability to identify and visualize the software project in proper, understandable, and four abstraction levels, i.e., Island metaphor view, city metaphor view, classroom view, and whiteboard illustration with code smells.
- Ability to identify, categorize, and visualize the code smells in the software project

To orient practitioners with code smells, detection approaches, and evolutionary characteristics, the following steps will be carried out. First, all participants will be introduced to code smells and visualizations of code smells. Second, a detailed description of the proposed visualization model

Table 2. Model's scalability for small, medium, and large-scale projects

Project	Small	Medium	Large
Name	Calculator app created with Java Swing	Java SE Inventory Management System	A scalable, large-scale eCommerce framework
GitHub Link	https://github.com/HouariZegai/Calculator	https://github.com/sajxraj/InventoryManagementSystem/tree/master	https://github.com/ilscipio/scipio-erp/tree/master
LOC	735	6568	1.6 million lines
No of classes	6	25	9,500
No of methods	25	205	60,000
Visualization using the proposed model	Successful	Successful	Unsuccessful

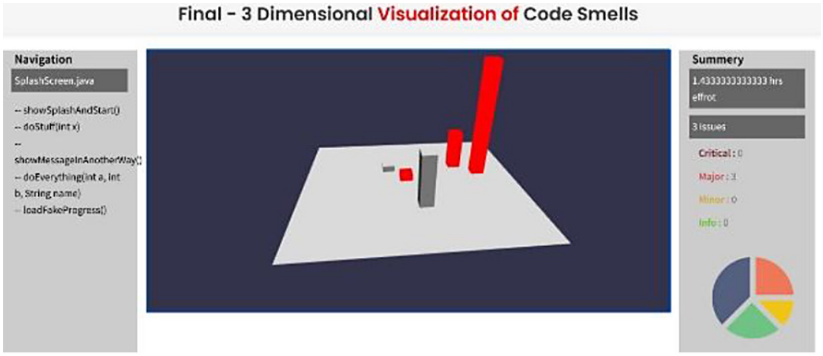

Figure 11. Version 1 of the SplashScreen.java class

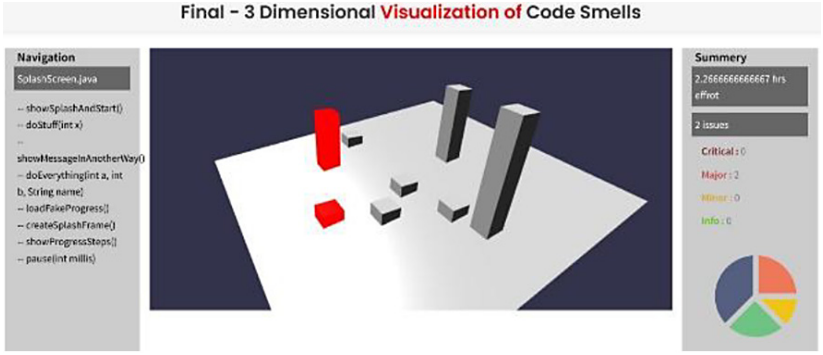

Figure 12. Version 2 of the SplashScreen.java class

will be given to the participants. The participants will be given time to go through the proposed visualization tool provided. Next, all participants will be asked to raise any questions to clarify any ambiguity before proceeding with the actual evaluation. After ensuring that all participants are familiar with the concept of code smells and their visualizations, the actual experiment phase will begin.

The experiment was organized as follows: As the first step, Google Forms were created with several questions for all levels, covering the process of code smell visualization. As the second step, the target code smells (duplicate blocks, long parameter lists, replace all(), extract T/Catch block) were extracted from the projects analyzed in SonarQube. This process had already been completed as part of the case study. The third step involves asking the participants to identify the objects using software solutions and answer the questions, along with the time consumed for each level. In the fourth step, feedback was collected from the participants to analyze whether the visualization helped them in detecting code smells and whether they were comfortable using the visualization tool. Finally, the duration that each respondent required to answer the questions at each level and the whole process of the tour was screen-recorded.

Participants were asked the following questions to evaluate their ability to detect code smells at each level.

Class Level Questions (CQ):

- CQ1: What are the classes in this Java project?
- CQ2: What are the code smell classes in the island metaphor (first view)?
- CQ3: How is the maximum NOC (Number of Code Lines) class identified?
- CQ4: What is the NOM (Number of Methods) in each class?

Method Level Questions (MQ):

- MQ1: What are the classes and their method names that include code smells?
- MQ2: What are the methods given priority to fix?
- MQ3: How is the minimum NOC (Number of Code Lines) method identified?
- MQ4: What is the percentage of code smell (%) in your existing class?

Inside the Method (IMQ):

- IMQ1: What is the type of code smell found in this method?
- IMQ2: What are the local variables and the number of attributes (NOA) within this method?
- IMQ3: What is the clean code percentage (%) in this method?
- IMQ4: Does this method include long parameter code smell? How did you identify it?
- IMQ5: What are the parameters in this method?

General Questions (GQ):

- GQ1: How do you find the method that includes code smells (critical)?
- GQ2: What is the shape of the class? What is the shape of the method?
- GQ3: Where can you find the code snippet including the code smells?

Evaluation of precision

The precision of the proposed model was evaluated based on the number of correct answers provided by participants when identifying code smells at various levels. Figures 13, 14, 15 and 16 show the number of correct answers submitted by participants for different code smells across levels. The results indicate that participants were generally able to identify code smells accurately, with some individuals performing better than others. For example, the first participant, a QA specialist, correctly answered 14 out of 17 questions, while another participant provided 16 correct answers. This demonstrates that the visualization model is effective in helping developers understand and identify code smells at different abstraction levels using the visual tools.

Evaluation of recall

Recall, or the ability to remember and identify code smells after an initial interaction with the model, was also assessed. Figure 17 shows the time taken by the participants to complete the tasks at the common level. This was designed to test the user's memorability of the novel model. Overall, the participants took 4.49 minutes to complete the tasks at the fourth level, showcasing their ability to recall and identify code smells efficiently (Figure 18). This demonstrates that the model is not only easy to learn but also supports identifying code smells.

Comparison of method-level and class-level detection

To assess how well users identify code smells at different abstraction levels, different sets of questions were used. The evaluation included questions related to specific code smells visualized using the model.

Figure 19 illustrates the correct answers provided by the eleven participants to the class-level questions. The group that used code smells visualization for the class-level questions answered 57.1% of all questions correctly.

Figure 20 shows the correct answers provided by the eleven participants to the method-level code smells visualization questions. The participants who used visualization questions at the method level completed the task with 81% of all questions answered correctly.

The data demonstrates that the method-level (81% accuracy) code smell detection outperforms class-level (57.1% accuracy) detection in terms of accuracy. This difference can happen because of the granularity of abstraction at each level. The City Metaphor, which represents methods as buildings, provides more detailed information, allowing developers to focus on specific code smells within methods, such as long parameter lists or feature envy. These

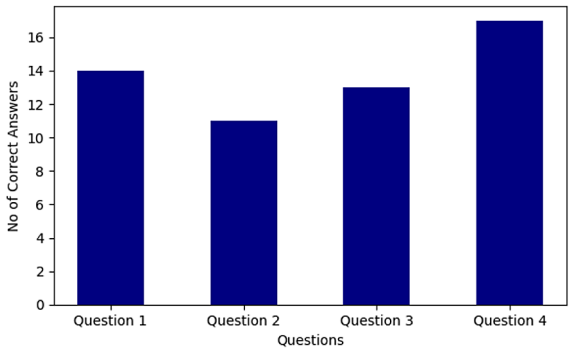

Figure 13. Number of correct answers submitted at class level

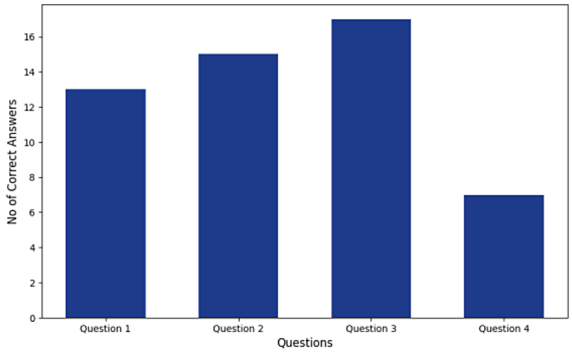

Figure 14. Number of correct answers submitted on the method level

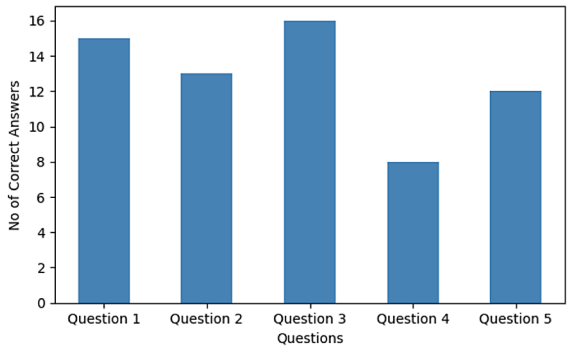

Figure 15. Number of correct answers submitted at the inside method level

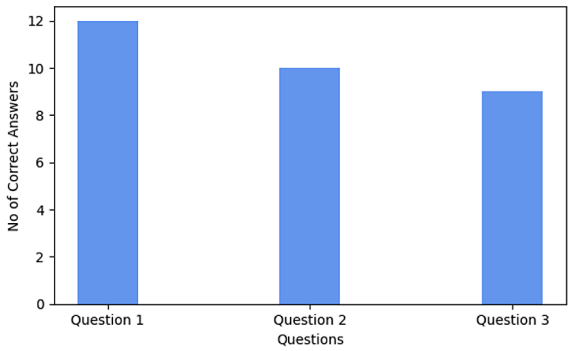

Figure 16. Number of correct answers submitted in the common level

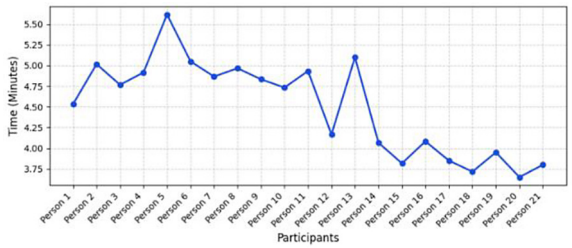

Figure 17. Total time taken to complete the common level

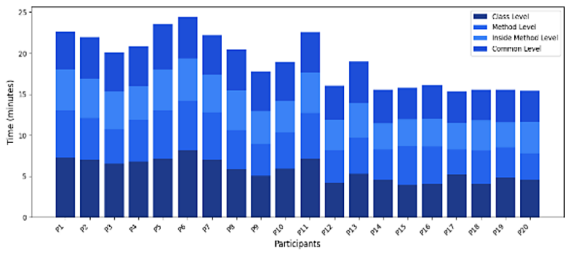

Figure 18. Time to complete each level

smells are more difficult to detect at the broader class level, where the island metaphor groups entire classes together. According to Cognitive Load Theory, visualizing more granular details at the method level reduces cognitive load by presenting developers with smaller chunks of information, making it easier to spot and address specific code smells [35].

Time analysis across levels

This study relies on three consecutive abstract levels of process that class level, method level, and inside method level. Participants answered with the prepared questions by identifying the object and the bad smells visualization in this tool. In this context, Figure 18 depicts the analysis of the time participants took to complete each level.

Hence, the analysis clearly provides sufficient information about the total time taken for each level and the time taken to complete all the levels. Therefore, the average time to complete the whole process is 18.87 minutes for each user.

Table 3 shows the average time taken by all participants to complete all levels and each level separately. The class level tool is the longest, as it involves a broad scope of code visualization. The

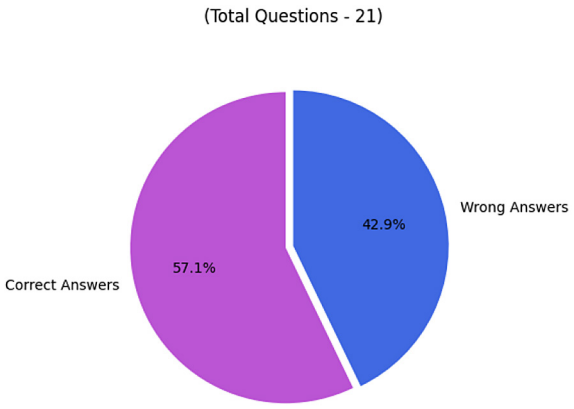

Figure 19. Class level code smell identification

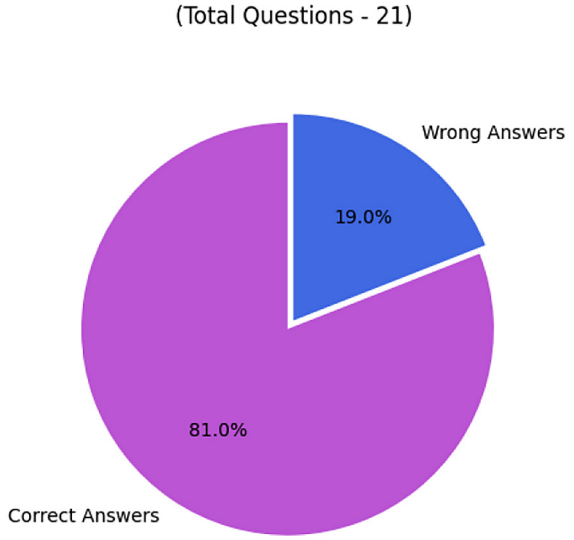

Figure 20. Method level code smell identification

island metaphor at this point required developers to process large, less specific chunks of information, which is more cognitively demanding. In contrast, the method level (4.55 minutes) focused on more granular details (methods), which were easier to navigate and interpret, aligning with Cognitive Load Theory that suggests smaller, more specific information chunks are easier to process.

The inside method level took 4.12 minutes, showing that once developers focused on individual methods, the task became quicker and more efficient. This suggests that method-level and inside-method visualizations help developers identify specific code smells more efficiently, reducing cognitive load.

In total, the average time of 18.87 minutes reflects the model's efficiency across each level, where granular abstraction levels (method and inside-method) led to faster detection of code smells.

Model comparison

Table 4 shows a detailed comparison of several existing smell detection tools and visualization tools alongside the proposed model, highlighting key features such as visual method, interactivity, scalability, abstraction levels, code smells detected, and refactoring suggestions. This set of attributes was chosen because together, they cover fundamental aspects to assess how effective tools are at visualizing code smells. This assessment encompasses visualization tools of both 2D and 3D, providing insight on how different dimensions affect developer perception, scalability, and usability.

When comparing the proposed model with the existing models, the proposed model demonstrates high interactivity, allowing users to explore code structures by zooming, localizing and browsing across multiple levels of details. Its scalability is categorized as medium, as visualizing 3D visualizations for large scale projects is hard because of its complexity. In terms of abstraction, the model not only supports class level, but also extends to method level. Moreover, it goes further to visualize code smells in Inside Methods, allowing developers to trace code smells form high-level structures down to behavioural interactions within methods.

Threats to validity

1. Internal validity

Internal validity relates to the correctness of the evaluation procedure and whether confounding factors may have influenced the outcome. One threat arises from the lack of a control group using conventional tools (e.g., SonarQube UI) for comparison. As a result, it is difficult to isolate whether improvements in performance stemmed from the 3D visualization model itself or from the novelty of the task. In addition, while all participants received the same training and instructions, variation in familiarity with code smell concepts or experience with 3D environments could have affected their performance. To reduce this risk, the evaluation included a standardized orientation session and task walkthroughs for all participants.

2. External validity

External validity concerns whether the results of this experiment can be generalized to more general contexts. First, the evaluation was conducted using reasonably simple and moderately sized

Table 3. Average time to complete the whole process and each level

-	Class	Method	Inside	Common
Avg time (min)	5.68	4.55	4.12	4.48

Table 4. Model comparison between existing models and the proposed model

Tool	Visualization	Interactivity	Scalability	Abstraction levels
JDeodorant [10]	None	Limited	Limited	Class and method level
InFusion [9]	None	None	Medium	Class, method and metrics level
VISMELLS [26]	2D	Medium	Limited	Class and method level
CodeCity [36]	3D	Limited	Medium	Class and package levels
CodeCharta	3D	Medium	Medium	File levels
Proposed model	3D	High	Medium	Class level, method level, inside method

Note: CodeCharta [Internet]. CodeCharta. [cited 2025 Oct 10]. https://codecharta.com/

software projects, which may not fully represent the complexity and scale of large-scale software systems. Though the intention is to test usability in a controlled setting, the evaluation results may vary for real-world software projects. Second, the participant group, though diverse in professional roles (developers, QA engineers, tech leads), consisted of only 11 individuals, which may not reflect the full spectrum of developer experience or industry practices.

3. Conclusion validity

Conclusion validity refers to the strength of the inferences drawn from the data. The relatively small number of participants presents a threat to statistical reliability. Since no statistical hypothesis testing was applied (e.g., t-tests, MWU), conclusions are based on descriptive metrics only (e.g., accuracy percentages, task duration). This limits the ability to generalize the findings beyond this study. However, as proof-of-concept, the evaluation offers useful initial evidence of the model's practical usage and lays the foundation for more rigorous empirical studies in future work.

CONCLUSIONS

This paper introduces a novel 3D visualization model to effectively visualize code smells by integrating island and city metaphors to represent classes, methods, and their relationships. The code smells identified through SonarQube are mapped into visual objects, allowing developers to understand software systems across multiple abstraction levels. Consequently, this visualization enhances program comprehension, supports the refactoring of problematic code, and ultimately contributes to improving the overall quality of software projects. The evaluation results revealed its effectiveness in recognizing code smells and their underlying reasons in an efficient way. Based on the results, it is evident that the metaphor-based visualization supports developers' understanding by leveraging spatial memory and helping form intuitive mental models of code. Mapping abstract code structures to familiar real-world structures reduces cognitive load and aids in comprehension and decision-making.

Future work will focus on visualizing additional elements, such as class relationships, method interactions, data type dependencies, and attribute invocations. Furthermore, integration of this model with state-of-the-art tools used in the software industry would streamline the decision-making process by providing higher-level insights into software quality.

Acknowledgements

The authors of this paper gratefully acknowledge the financial support provided by the Senate Research Grant (Grant No – SCR/ST/2025/78) of the University of Moratuwa. Also, authors would like to thank the School of Computing at Asia Pacific Institute of Information Technology (APIIT).

REFERENCES

1. Cairo AS, Carneiro G de F, Monteiro MP. The Impact of Code Smells on Software Bugs: A Systematic Literature Review. Information [Internet]. 2018 Nov [cited 2025 Sept 23]; 9(11): 273. https://www.mdpi.com/2078-2489/9/11/273

2. Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. In: Proceedings of the 40th International Conference on Software Engineering [Internet]. New York, NY, USA: Association for Computing Machinery; 2018 [cited 2025 Sept 22]. 482. (ICSE '18). https://doi.org/10.1145/3180155.3182532
3. Sharma T, Spinellis D. A survey on software smells. J Syst Softw [Internet]. 2018 Apr 1 [cited 2025 Sept 23]; 138: 158–73. https://www.sciencedirect.com/science/article/pii/S0164121217303114
4. Hasantha C. A Systematic Review of Code Smell Detection Approaches. 2021 May 5 [cited 2025 Sept 23]; https://zenodo.org/record/4738772
5. Rao RS, Dewangan S, Mishra A. An Empirical Evaluation of Ensemble Models for Python Code Smell Detection. Appl Sci [Internet]. 2025 Jan [cited 2025 Sept 23]; 15(13): 7472. https://www.mdpi.com/2076-3417/15/13/7472
6. Palomba F, Di Nucci D, Panichella A, Zaidman A, De Lucia A. On the impact of code smells on the energy consumption of mobile applications. Inf Softw Technol [Internet]. 2019 Jan 1 [cited 2025 Sept 23]; 105: 43–55. https://www.sciencedirect.com/science/article/pii/S0950584918301678
7. Al-Shaaby A, Aljamaan H, Alshayeb M. Bad smell detection using machine learning techniques: A systematic literature review. Arab J Sci Eng [Internet]. 2020 Apr 1 [cited 2025 Sept 23]; 45(4): 2341–69. https://doi.org/10.1007/s13369-019-04311-w
8. Steinbeck M. An arc-based approach for visualization of code smells. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER) [Internet]. 2017 [cited 2025 Sept 23]. 397–401. https://ieeexplore.ieee.org/abstract/document/7884641
9. Paiva T, Damasceno A, Figueiredo E, Sant'Anna C. On the evaluation of code smells and detection tools. J Softw Eng Res Dev [Internet]. 2017 Oct 6 [cited 2025 Sept 23]; 5(1): 7. https://doi.org/10.1186/s40411-017-0041-1
10. Tsantalis N, Chaikalis T, Chatzigeorgiou A. JDeodorant: Identification and Removal of Type-Checking Bad Smells. In: 2008 12th European Conference on Software Maintenance and Reengineering [Internet]. 2008 [cited 2025 Sept 23]. 329–31. https://ieeexplore.ieee.org/abstract/document/4493342
11. Imran A, Kosar T, Zola J, Bulut MF. Predicting the Impact of Batch Refactoring Code Smells on Application Resource Consumption [Internet]. arXiv; 2023 [cited 2025 Sept 23]. http://arxiv.org/abs/2306.15763
12. Biederman I. Recognition-by-components: A theory of human image understanding. Psychol Rev. 1987; 94(2): 115–47.
13. Albuquerque D, Guimarães E, Braga A, Perkusich M, Almeida H, Perkusich A. Empirical Assessment on Interactive Detection of Code Smells. In: 2022 International Conference on Software, Telecommunications and Computer Networks (SoftCOM) [Internet]. 2022 [cited 2025 Oct 27]. 1–6. https://ieeexplore.ieee.org/document/9911317
14. Wijesiriwardana C, Wimalaratne P. Fostering Real-Time Software Analysis by Leveraging Heterogeneous and Autonomous Software Repositories. IEICE Trans Inf [Internet]. 2018 Nov 1 [cited 2025 Sept 23]; E101-D(11): 2730–43. https://globals.ieice.org/en_transactions/information/10.1587/transinf.2018EDP7094/#
15. Bassil S, Keller RK. Software visualization tools: survey and analysis. In: Proceedings 9th International Workshop on Program Comprehension IWPC 2001 [Internet]. 2001 [cited 2025 Sept 23]. 7–17. https://ieeexplore.ieee.org/abstract/document/921708
16. Katbi A, Hammad M, Elmedany W. Multi-view city-based approach for code-smell evolution visualisation. IET Softw [Internet]. 2020 [cited 2025 Sept 23]; 14(5): 506–16. https://onlinelibrary.wiley.com/doi/abs/10.1049/iet-sen.2020.0010
17. Murphy-Hill E. Scalable, expressive, and context-sensitive code smell display. In: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications [Internet]. New York, NY, USA: Association for Computing Machinery; 2008 [cited 2025 Sept 22]. 771–2. (OOPSLA Companion '08). https://doi.org/10.1145/1449814.1449854
18. Mumtaz H, Beck F, Weiskopf D. Detecting Bad Smells in Software Systems with Linked Multivariate Visualizations. In: 2018 IEEE Working Conference on Software Visualization (VISSOFT) [Internet]. 2018 [cited 2025 Sept 23]. 12–20. https://ieeexplore.ieee.org/abstract/document/8530127
19. Langelier G, Sahraoui H, Poulin P. Visualization-based analysis of quality for large-scale software systems. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering [Internet]. New York, NY, USA: Association for Computing Machinery; 2005 [cited 2025 Sept 22]. 214–23. (ASE '05). https://doi.org/10.1145/1101908.1101941
20. Schumacher J, Zazworka N, Shull F, Seaman C, Shaw M. Building empirical support for automated code smell detection. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement [Internet]. New York, NY, USA: Association for Computing Machinery; 2010 [cited 2025 Sept 22]. 1–10. (ESEM '10). https://doi.org/10.1145/1852786.1852797
21. Li W, Shatnawi R. An empirical study of the bad smells and class error probability in the post-release

object-oriented system evolution. J Syst Softw [Internet]. 2007 July 1 [cited 2025 Sept 23]; 80(7): 1120–8. https://www.sciencedirect.com/science/article/pii/S0164121206002780

22. Yamashita A, Moonen L. Do developers care about code smells? An exploratory survey. In: 2013 20th Working Conference on Reverse Engineering (WCRE) [Internet]. 2013 [cited 2025 Sept 23]. 242–51. https://ieeexplore.ieee.org/abstract/document/6671299
23. Dhambri K, Sahraoui H, Poulin P. Visual Detection of Design Anomalies. In: 2008 12th European Conference on Software Maintenance and Reengineering [Internet]. 2008 [cited 2025 Sept 23]. 279–83. https://ieeexplore.ieee.org/abstract/document/4493326
24. Murphy-Hill E, Black AP. An interactive ambient visualization for code smells. In: Proceedings of the 5th international symposium on Software visualization [Internet]. New York, NY, USA: Association for Computing Machinery; 2010 [cited 2025 Sept 22]. 5–14. (SOFTVIS '10). https://dl.acm.org/doi/10.1145/1879211.1879216
25. Carneiro G de F, Silva M, Mara L, Figueiredo E, Sant'Anna C, Garcia A, et al. Identifying Code Smells with Multiple Concern Views. In: 2010 Brazilian Symposium on Software Engineering [Internet]. 2010 [cited 2025 Sept 23]. 128–37. https://ieeexplore.ieee.org/abstract/document/5629742
26. Silva I de J, Santos MSR, Ramos LL, Carvalho LP da S. VISMELLS: An Interactive Visualization for Identifying and Evaluating the Effects of Code Smells on Software Projects. In: 2018 XLIV Latin American Computer Conference (CLEI) [Internet]. 2018 [cited 2025 Sept 23]. 40–9. https://ieeexplore.ieee.org/abstract/document/8786346
27. 2Misiak M, Schreiber A, Fuhrmann A, Zur S, Seider D, Nafeie L. IslandViz: A Tool for Visualizing Modular Software Systems in Virtual Reality. In: 2018 IEEE Working Conference on Software Visualization (VISSOFT) [Internet]. 2018 [cited 2025 Sept 23]. 112–6. https://ieeexplore.ieee.org/abstract/document/8530137
28. Schreiber A, Misiak M. Visualizing Software Architectures in Virtual Reality with an Island Metaphor. In: Chen JYC, Fragomeni G, editors. Virtual, Augmented and Mixed Reality: Interaction, Navigation, Visualization, Embodiment, and Simulation. Cham: Springer International Publishing; 2018; 168–82.
29. Wijayawardena ASK, Abeysekera R, Maduranga MWP. A Systematic Review of 3D Metaphoric Information Visualization. Int J Mod Educ Comput Sci [Internet]. [cited 2025 Sept 23]; 15(1): 73. https://www.mecs-press.org/ijmecs/ijmecs-v15-n1/v15n1-6.html
30. Wijesiriwardana C, Wimalaratne P, Abeysinghe T, Shalika S, Ahmed N, Mufarrij M. Secure CodeCity: 3-dimensional visualization of software security facets. Journal of the National Science Foundation of Sri Lanka. 2023 Oct 10 [cited 2025 Sept 23]; https://jnsfsl.sljol.info/articles/10.4038/jnsfsr.v51i3.11201
31. Moreno-Lumbreras D, Gonzalez-Barahona JM, Robles G, Cosentino V. The influence of the city metaphor and its derivates in software visualization. J Syst Softw [Internet]. 2024 Apr 1 [cited 2025 Sept 23]; 210: 111985. https://www.sciencedirect.com/science/article/pii/S0164121224000281
32. Jeffery CL. The City Metaphor in Software Visualization. In: Computer Science Research Notes [Internet]. Západočeská univerzita; 2019 [cited 2025 Sept 23]. http://wscg.zcu.cz/wscg2019/2019-papers/!!_CSRN-2801-18.pdf
33. Lenarduzzi V, Lomio F, Huttunen H, Taibi D. Are SonarQube Rules Inducing Bugs? In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER) [Internet]. 2020 [cited 2025 Oct 10]. 501–11. https://ieeexplore.ieee.org/document/9054821/
34. Merino L, Fuchs J, Blumenschein M, Anslow C, Ghafari M, Nierstrasz O, et al. On the Impact of the Medium in the Effectiveness of 3D Software Visualizations. In: 2017 IEEE Working Conference on Software Visualization (VISSOFT) [Internet]. 2017 [cited 2025 Sept 23]. 11–21. https://ieeexplore.ieee.org/document/8091182/
35. Gonçalves PW, Fregnan E, Baum T, Schneider K, Bacchelli A. Do explicit review strategies improve code review performance? Towards understanding the role of cognitive load. Empir Softw Eng [Internet]. 2022 May 7 [cited 2025 Oct 27]; 27(4): 99. https://doi.org/10.1007/s10664-022-10123-8
36. Wettel R, Lanza M. CodeCity: 3D visualization of large-scale software. In: Companion of the 30th international conference on Software engineering [Internet]. New York, NY, USA: Association for Computing Machinery; 2008 [cited 2025 Oct 10]. 921–2. (ICSE Companion '08). https://doi.org/10.1145/1370175.1370188