DevDog: A Gamified Web Platform for Teaching Code Smell Identification

Luciano Ricardo da Silva Júnior
Centro de Estudos, Desenvolvimento e Inovação em Software (CEDIS)
University of Brasília
Brasília, Brazil
lrsj2003@gmail.com

Sergio Antônio Andrade de Freitas
Centro de Estudos, Desenvolvimento e Inovação em Software (CEDIS)
University of Brasília
Brasília, Brazil
sergiofreitas@unb.br

André Luiz Peron Martins Lanna
Centro de Estudos, Desenvolvimento e Inovação em Software (CEDIS)
University of Brasília
Brasília, Brazil
andrelanna@unb.br

Abstract

Code smells are design symptoms whose interpretation depends on context and engineering judgement. Static analyzers can report likely maintainability issues, but learners still need to practice locating suspicious fragments and discriminating among smell types. This paper presents DevDog, a deployed, open-source web platform for deliberate practice in code smell identification. Learners consult a theory guide, inspect contextualized JavaScript snippets, mark lines, assign one of 19 smell categories, and receive oracle-based feedback. A dog metaphor connects scores, mascot reactions, collectible *petiscos*, progressive hints, and per-exercise community statistics. Unlike professional detectors, DevDog makes the learner perform the inspection; unlike conventional online judges, it evaluates design judgements rather than functional outputs. We position DevDog against SonarQube, Code Smell Analyzer, CleanGame, RefGame, TSGame, and URI Online Judge Academic, and clarify its content-authoring and oracle model. The platform uses React, Express, Prisma, PostgreSQL, and Docker and is distributed under the MIT License. An exploratory motivational diagnosis with eight students and a prototype usability test with three students informed design refinements; these activities are formative and do not establish learning effectiveness. **Demo video:** https://zenodo.org/records/20298757.

Keywords

Code Smells, Gamification, Software Engineering Education, Refactoring, Web-based Tool, Programming Education

1 Introduction

Code smells are surface symptoms that may indicate deeper design problems and refactoring opportunities [6, 8]. They are not faults by definition: deciding whether a structure is problematic requires context, design intent, and awareness of trade-offs. Empirical studies associate smells with maintainability and evolution concerns [18, 19], while reviews report subjective definitions and oracles, imbalanced datasets, and concentration on a limited set of languages and smell types [11, 20]. These characteristics make smell identification a judgement-intensive activity rather than a simple rule-recall task.

This creates a pedagogical challenge. Students may memorize names and definitions yet struggle to inspect unfamiliar code, distinguish similar symptoms, and explain maintenance consequences. Automated analysis is useful in professional work, but it can remove the localization and interpretation steps that novices need to exercise. Refactoring education therefore benefits from concrete examples, repeated decisions, and feedback rather than definition-only instruction [17]. A useful learning environment should require students to inspect before revealing the oracle and should preserve enough context for them to justify their classification.

Gamification can support sustained practice when its mechanics reinforce competence, autonomy, and meaningful progress instead of merely adding points or decorative rewards [4, 12, 13]. Evidence across education and software engineering suggests that outcomes depend on the learning activity, the selected mechanics, and the way feedback is integrated into the task [7, 9]. Consequently, a gamified smell-learning tool should connect rewards to code inspection and reflection, not to passive navigation.

DevDog addresses this problem through two connected spaces: *Guia*, for consulting concepts and examples, and *Farejador*, for locating and classifying smells in contextualized code. The platform is deployed at https://devdog.cedis.tec.br; its source code is available at https://github.com/CedisUnB/CodeSmellGamification under the MIT License.

This paper contributes: (i) a deployable workflow in which learners submit explicit line–smell pairs instead of receiving detector findings; (ii) an integrated design combining theory consultation, contextual exercises, immediate feedback, progressive paid hints, anonymous access, and per-exercise statistics; (iii) a reproducible client–server implementation with Docker and open seed data; and (iv) transparent formative evidence and limitations that distinguish design refinement from validation of learning gains. We do not claim a new smell detector, a new gamification theory, or the first game related to code smells. The contribution is the particular low-friction, line-level learning configuration and its open implementation.

2 Related Tools And Positioning

Professional analyzers, online judges, and educational games cover different parts of the learning problem. SonarQube and its IDE integrations automatically surface quality and maintainability issues in projects as part of development workflows [16]. Code Smell Analyzer detects smells in student code and presents explanations and possible refactorings [15]. These systems can teach from detected issues, but the initial localization decision is automated.

Educational systems provide closer comparisons. CleanGame combines concept questions with smell-identification tasks over Java code, integrates PMD to produce findings, and adds hints, scores, badges, rankings, and task creation [14]. RefGame uses three

games to teach the identification and application of refactoring techniques [1]. TSGame lets instructors assign JUnit test-smell detection and refactoring tasks in web-based game sessions [5]. URI Online Judge Academic, in turn, supports programming classes through executable submissions, test-based correctness, exercise lists, and progress tracking [2]. Table 1 compares these systems according to the unit of work and the action being assessed.

The comparison narrows the novelty claim. DevDog does not introduce an individually novel game mechanic, and CleanGame already established gamified smell-identification practice. DevDog's incremental contribution is the integrated workflow: context-first browser snippets, manual line localization, discrimination among 19 smell types, immediate oracle feedback, progressive hints purchased with earned currency, anonymous-to-registered progress migration, and persistent statistics scoped to each exercise. This configuration intentionally keeps the learner's judgement as the submitted answer while remaining lightweight enough for a lecture, laboratory, or individual study session.

Six pedagogical and technical goals guided the implementation. Table 2 makes the design rationale explicit and prevents the features from appearing as an arbitrary list. The goals also define boundaries: the current release is a deliberate-practice environment, not a repository-scale detector, a refactoring recommender, or a complete learning-management system.

3 The Devdog Tool

DevDog's intended users are students learning refactoring, maintainability, or software quality and instructors seeking short individual or classroom activities. Figure 1 shows the implemented path: theory consultation, exercise selection, line-level classification, and immediate feedback. The montage is framed on white and contrast-enhanced for print while preserving the platform's dark visual identity.

3.1 Learning Flow And Feedback

A learner may start without creating an account. DevDog creates an anonymous user, persists progress in a token-based session, and migrates coins and attempts if the learner later registers. This choice reduces onboarding cost during demonstrations and short classes while retaining a path to longitudinal practice.

In *Farejador*, the learner reads a maintenance scenario, inspects a formatted snippet, selects one or more lines, and assigns a smell type to each selection. The backend compares the submitted line–smell pairs with the exercise oracle, stores the attempt, computes the score, and returns line-level feedback, stars, earned *petiscos*, and a mascot reaction. The interface also reports attempts, best score, rank, participant count, and community average for the current exercise.

A representative activity uses a temperature-conversion routine. The description asks the learner to consider future unit additions and repeated primitive representations. The learner marks the lines judged problematic and chooses, for example, Duplicated Code or Primitive Obsession. If uncertain, the learner can spend *petiscos* to reveal progressively the number of affected lines, the number of distinct smell types, and one target line. A retry then becomes an informed second inspection rather than a blind resubmission.

Let *correct* be matched line–smell pairs, *incorrect* be selected pairs that do not match the oracle, *missed* be oracle pairs not selected, and *total* be the number of oracle pairs. DevDog computes

$$score = \max(0, correct - incorrect - missed)/total \times 100.$$

Scores yield five *petiscos* for 100, four for at least 80, three for at least 60, two for at least 40, and one for at least 20. The progressive hint sequence provides a recovery path while retaining part of the challenge. The scoring rule also penalizes indiscriminate line selection because unmatched selections reduce the result.

3.2 Content Scope, Oracle Construction, And Extension

The first catalog uses JavaScript as a pragmatic scope decision: its compact syntax supports self-contained browser snippets and matches the web-oriented implementation context. This is not a claim that JavaScript is more representative of code smells. Source code is stored as text and answers as line-number/type pairs, so the scoring service is largely language-independent; adding another language still requires new educational content, syntax-highlighting support, and reviewed oracles.

The release covers 19 categories inspired by Fowler's catalog: Mysterious Name, Duplicated Code, Long Method, Long Parameter List, Global Data, Mutable Data, Divergent Change, Shotgun Surgery, Feature Envy, Data Clumps, Primitive Obsession, Repeated Switches, Lazy Element, Speculative Generality, Temporary Field, Message Chains, Middle Man, Large Class, and Comments. This is neither an exhaustive taxonomy nor a claim that Fowler's catalog is the only valid one. Table 3 groups the implemented categories by the reasoning they are intended to elicit.

The project team authored the current content from intended maintenance scenarios and Fowler-inspired definitions rather than generating it with a detector. Each exercise combines a title, contextual description, author-assigned difficulty, code text, and explicit `SmellLine` records. This approach allows deliberately constructed boundary cases and multi-smell examples, but it makes oracle quality dependent on author judgement. Difficulty is likewise a content-author label rather than a level calibrated from learner performance.

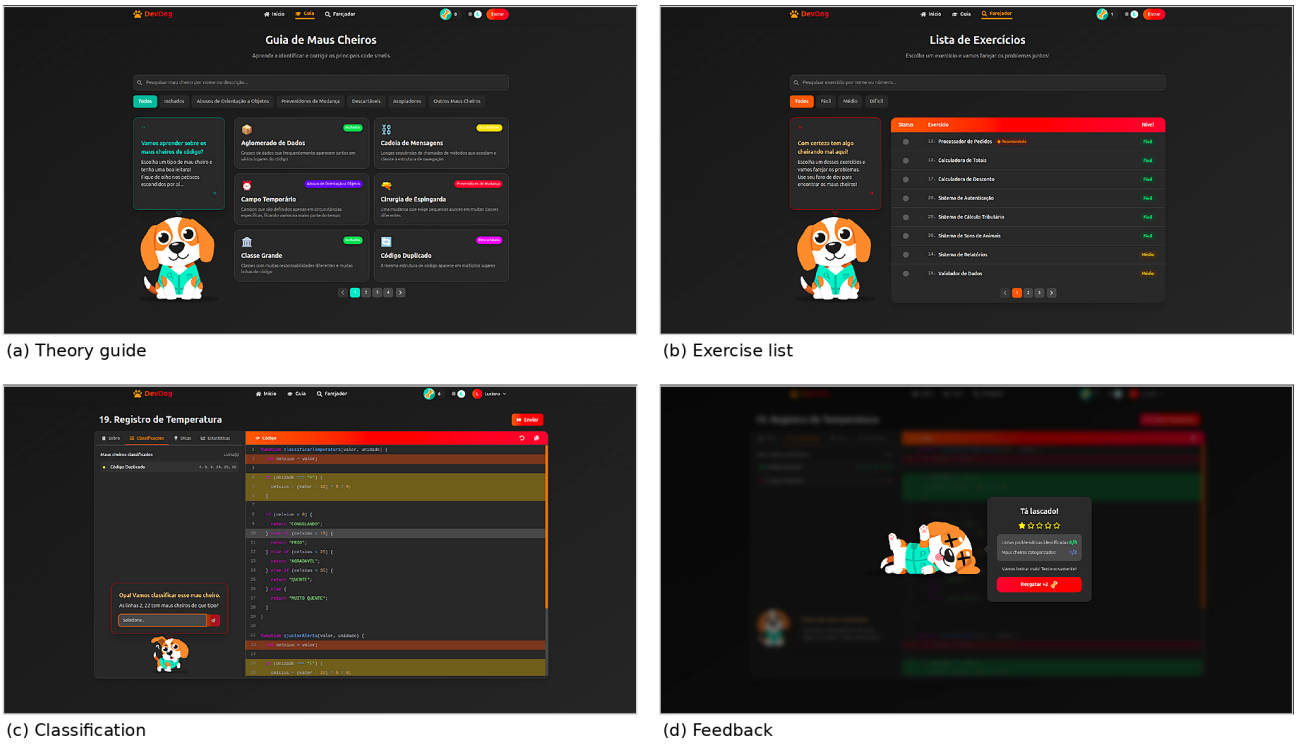
Instructors can extend the catalog by adding an `Exercise` and nested `SmellLine` records to `api/prisma/seed.js` and running the repository's seed command. The technical format is simple, but robust exercise authoring is not: the author must create a plausible maintenance context, choose defensible smell labels, account for alternative interpretations, and keep line numbers synchronized when code changes. The present release therefore supports reproducible source-level extension but not a graphical authoring workflow or multi-expert adjudication. Both are explicit next steps.

4 Architecture And Reproducibility

DevDog follows a layered client–server architecture. The React/Vite frontend contains route-level pages for the home, guide, exercise list, exercise detail, authentication, and errors, plus reusable components for code viewing, line selection, tips, statistics, mascot states, and result dialogs. An Axios service layer calls an Express API responsible for anonymous and registered access, exercise retrieval, attempt evaluation, hint purchase, recommendation, and statistics.

Table 1: Concrete positioning of DevDog against representative code-quality and educational systems.

System	Primary unit and purpose	Assessed learner/developer action	Relationship to DevDog
SonarQube / IDE integrations [16]	Projects analyzed automatically by quality and security rules	Interpret and respond to issues already located by the analyzer	Professional verification is automated; DevDog withholds findings so learners first localize and classify design symptoms.
Code Smell Analyzer [15]	Student source code submitted for smell diagnosis and refactoring guidance	Inspect explanations after automated detection	DevDog uses curated scenarios to practice human inspection rather than automate the diagnosis step.
CleanGame [14]	Java repositories and gamified post-training smell tasks	Answer quizzes and identify PMD-supported smells with hints and rankings	Closest prior system; DevDog uses short contextual snippets, explicit line–type pairs, progressive paid hints, and per-exercise comparison.
RefGame [1]	Three games about identifying and applying refactorings	Select or apply a refactoring technique	DevDog concentrates on the prior decision: recognizing and discriminating among smell categories before choosing a refactoring.
TSGame [5]	JUnit test-smell detection and removal tasks assigned by teachers	Detect and refactor test smells during game sessions	TSGame targets test code and remediation; DevDog targets 19 production-code smell categories and line-level localization.
URI Online Judge Academic [2]	Executable programs evaluated against expected outputs	Submit a solution judged by functional tests	DevDog adapts the short-challenge and immediate-feedback model to structured design judgements, not program outputs.



(a) Theory guide (b) Exercise list

(c) Classification (d) Feedback

Figure 1: Implemented DevDog workflow: (a) theory guide, (b) exercise list and recommendation, (c) line selection and smell classification, and (d) immediate result feedback.

Prisma maps the backend to PostgreSQL. JSON Web Tokens, bcrypt, and CORS support session and access concerns.

Table 4 maps the main modules to their interfaces. The separation is relevant for adoption and future evolution: an instructor dashboard or an authoring client can consume the same API without reimplementing scoring, while the browser client can evolve independently from the database schema.

Table 2: Pedagogical and technical goals that guided DevDog.

Goal	Design consequence
Practice first	Require repeated inspection of contextual code instead of only recalling definitions.
Low-friction access	Allow anonymous entry and preserve progress if the learner later registers.
Explicit localization	Make the learner select suspicious lines before feedback is revealed.
Type discrimination	Require a smell category for each selected line, exposing conceptual confusions.
Productive support	Provide immediate results and progressive hints without revealing the full oracle at once.
Reproducibility	Use common web technologies, Docker, persistent attempts, and structured seed data.

The persistent model is intentionally compact. `User` stores identity state, anonymity, e-mail, password hash, coins, and merge information. `Exercise` stores context, difficulty, and code. `SmellLine` links an exercise to a smell type and line number. `Attempt` stores the user, exercise, score, correct counts, and raw selections. Centralizing oracle comparison in the attempt endpoint ensures that displayed feedback and stored records use the same rule.

The exercise-list endpoint implements a minimal recommendation strategy: beginners receive an easy exercise, and subsequent recommendations move to medium and hard levels after completion of the preceding level. This is a deterministic progression rule, not an adaptive learner model. The distinction matters because the system does not currently infer mastery, misconception profiles, or optimal sequencing from historical data.

Docker Compose orchestrates PostgreSQL, migrations, API, and web containers. The public repository provides environment examples, schema, migrations, seed data, and execution instructions; the documented workflow starts the containers and runs `docker compose exec api npm run seed`. These assets support inspection and local replication of the deployed behavior. The MIT License permits reuse and modification, although pedagogical adoption still requires instructors to inspect and validate the supplied examples and oracles for their course context.

5 Gamification And Instructional Use

5.1 Design Of The Game Elements

The gamification design combined contextual analysis, motivational profiling, ideation, and prototyping. A judge-based assessment mapped candidate mechanics to Octalysis core drives [3, 10]. The highest means were Unpredictability and Curiosity (3.64), Ownership and Possession (3.55), Development and Accomplishment (3.50), and Empowerment of Creativity and Feedback (3.36). An adapted Intrinsic Motivation Inventory was administered to eight students in a software engineering course. In this exploratory dataset, removing one anxiety item changed Cronbach's alpha from 0.63 to 0.73; the highest means were Value/Usefulness (3.63), Effort/Importance (3.54), and Perceived Competence (3.50).

Table 5 shows how these observations were translated into implemented mechanics. Rewards are attached to attempts rather than page visits, hints consume an earned resource, and social comparison is local to an exercise. The design therefore does not assume that points themselves produce learning; their function is to structure repetition, strategic support, and visible progress around the inspection task.

5.2 Classroom And Research Affordances

For instructors, DevDog can be used at different moments of a software quality unit. Before a lecture, a small exercise set can reveal which smells students already recognize. During class, learners can solve individually, compare line selections in pairs, and then discuss disputed classifications with the instructor. After class, repeated attempts and progressive hints support deliberate practice. This sequence uses the oracle for timely feedback while preserving discussion about the subjective or context-dependent cases.

The two learning spaces support different pedagogical actions. *Guia* can provide just-in-time consultation before or between attempts; *Farejador* requires an observable decision that can be stored. The community average can help an instructor identify an exercise that is broadly difficult, but it should not be interpreted as a normed proficiency score because participation, repeated attempts, and hint use are not controlled. Likewise, the current rank is local to an exercise and intentionally avoids making a global leaderboard the dominant incentive.

DevDog also records data that can support future studies: the exercise, selected lines, selected smell types, score, attempt history, and user/exercise association. These records enable analyses of confusion pairs, over-selection, repeated errors, progression across difficulty levels, and the relation between hint use and later attempts. They do not, by themselves, establish learning. A valid learning study would need unseen assessment tasks, a defined intervention, and analysis that separates practice effects from prior experience and instructor support.

The current instructional affordances also expose design limitations. Line-based answers are convenient for immediate feedback, but some smells are more naturally attached to blocks, methods, classes, or change histories. Short snippets reduce setup and reading cost, yet they cannot reproduce all repository-scale dependencies. These trade-offs are acceptable for an introductory practice tool provided that the article and instructors do not equate success on curated snippets with complete professional competence.

Table 3: Coverage and learning focus of the current DevDog exercise catalog.

Smell group	Examples in the catalog	Learning focus
Naming and locality	Mysterious Name, Comments, Lazy Element	Relate readability problems to maintenance cost and identify cases where names or explanatory comments signal missing refactorings.
Size and complexity	Long Method, Large Class, Long Parameter List	Recognize excessive responsibilities, oversized routines/classes, and parameter lists that hide cohesive domain concepts.
Duplication and change	Duplicated Code, Shotgun Surgery, Divergent Change, Repeated Switches	Connect repeated logic and scattered changes to maintainability risks and future evolution effort.
Data and abstraction	Data Clumps, Primitive Obsession, Temporary Field, Global Data, Mutable Data	Identify missing abstractions, hidden coupling, uncontrolled state, and domain concepts represented only by primitives.
Coupling and delegation	Feature Envy, Message Chains, Middle Man	Notice behavior placed near the wrong data, excessive navigation across object graphs, and pass-through abstractions.
Speculative design	Speculative Generality	Distinguish useful extension points from unused flexibility created for hypothetical futures.

Table 4: Main DevDog modules, responsibilities, and interfaces.

Module	Responsibility/interface
Frontend pages and components	Route screens and reusable code viewer, exercise panel, tips, results, mascot states, and navigation.
API service layer	HTTP client for authentication, exercises, attempts, hints, recommendation, and statistics.
Controllers	Anonymous access, account migration, exercise listing, oracle comparison, score computation, hints, and statistics.
Database	Persistent `User`, `Exercise`, `SmellLine`, and `Attempt` records.
Docker Compose	Reproducible orchestration of PostgreSQL, migrations, API, and web client.

Table 5: Motivational goals and implemented mechanics.

Goal	Implemented mechanics
Competence and progress	Difficulty, score, stars, attempts, best score, and immediate feedback.
Curiosity and exploration	Mascot reactions, optional hints, hidden *petiscos*, and recommended next exercise.
Ownership and strategy	Earned *petiscos* are accumulated and spent to unlock hints.
Relatedness with low pressure	Per-exercise average, participant count, and rank instead of a global leaderboard.
Low-friction participation	Anonymous use with later migration to a registered account.

6 Formative Evaluation And Limitations

6.1 Activities And Retained Evidence

Two early activities informed the design. The first was the motivational diagnosis described above with eight students in a software engineering course. It identified candidate motivational priorities and supported decisions such as reducing access barriers, making assistance optional, and connecting progress to visible feedback. Because the instrument preceded the final implementation, it is evidence about design inputs rather than evidence that DevDog caused motivation or learning.

The second activity was a usability test of a medium-fidelity prototype with three software engineering students. Participants completed four tasks: exploring the home page, consulting theory content, selecting an exercise, and solving a smell-identification challenge. The first three navigation-oriented tasks were completed more quickly than the fourth, which required code analysis, line selection, smell classification, and interpretation of feedback. The available study records describe the task structure and qualitative conclusions but do not include the raw per-participant timing sheet or a detailed demographic profile. Therefore, the results are reported without unverified numerical timing data.

Qualitative feedback described the prototype as clear, simple, intuitive, and fluid. Participants valued the mascot and visual identity, the separation between guide and practice, and the resemblance of *Farejador* to an online judge. Reported friction included insufficient examples or visual variety in theory pages, low visibility of the hint function, and unclear guidance when selecting lines. The final implementation responds through guide examples, progressive hints, explicit line-interaction cues, stronger result details, and multiple mascot states. These links between observations and changes are design traceability, not an effectiveness test.

Table 6 makes this traceability explicit. It separates observations actually retained in the project records from the design interpretation and from claims that remain untested. This distinction is important: a positive comment about the mascot can justify retaining it, but cannot establish that mascot-mediated feedback improves learning; similarly, completion of a prototype task indicates that the interaction can be followed, not that the learner can transfer smell knowledge to unseen code.

6.2 Threats To Validity

Construct validity. Motivation ratings and prototype usability are not proxies for learning gains. The scoring rule measures agreement with a curated line-level oracle, while real smell judgements may concern larger code regions and admit alternative interpretations. Stars, *petiscos*, and rank are interaction indicators rather than validated measures of competence.

Table 6: Traceability from formative evidence to the deployed design.

Retained evidence	Design interpretation	Response and claim boundary
High exploratory means for value, effort, and competence	Keep the practice task central and make progress visible.	Scores, stars, attempts, and feedback were retained; no causal motivational effect is claimed.
Mascot and visual identity were positively mentioned	A coherent character may make feedback memorable.	Multiple mascot states were implemented; learning impact remains untested.
Guide/practice separation was understood	Two explicit learning spaces reduce navigation ambiguity.	*Guia* and *Farejador* remain distinct but linked.
Hints were difficult to notice	Assistance needs a visible, staged interaction.	A dedicated tips area reveals progressively stronger hints for *petiscos*.
Line selection required clearer guidance	The primary answer action needs explicit cues and feedback.	Selection states and result highlighting were strengthened; first-use learnability still requires study.
Theory pages needed examples and visual variety	Definitions alone are insufficient preparation for practice.	Guide examples were expanded; content quality still requires independent expert validation.

Internal validity. The small studies did not isolate the effects of the mascot, hints, scoring, or other mechanics. Researcher moderation, task ordering, novelty effects, and participants' prior programming experience may have influenced the observed impressions. Because the usability activity concerned a prototype, changes in the deployed version also prevent direct attribution from those observations to current behavior.

External validity. The evidence comes from few students in one context, and the catalog currently uses short JavaScript examples. Results cannot be generalized to other institutions, professional developers, other languages, repository-scale code, or all smell taxonomies. Anonymous low-friction access may also produce usage patterns different from a graded classroom deployment.

Content validity. The project team authored the exercises and oracles without a reported independent expert-adjudication procedure, and difficulty labels have not been calibrated with learner data. A stronger release process should involve at least two independent software-quality experts, document disagreements, test examples with target learners, and preserve versioned oracle rationales. A stronger effectiveness study should use unseen pre/post tasks validated by independent software-quality experts, report participant background and complete task-level data, and analyze logs together with qualitative explanations.

6.3 Protocol Required For Confirmatory Claims

A subsequent classroom study should test whether DevDog improves learners' ability to localize and classify smells in unseen code and whether any improvement transfers beyond short JavaScript snippets. Engagement and usability should remain complementary outcomes rather than substitutes for learning.

A defensible design would use pre-, immediate post-, and delayed post-tests with unseen examples matched by smell category, code length, context, and expected difficulty. The assessment set should be independent of the training catalog and validated by at least two software-quality educators. Besides line–smell agreement, a rubric should score written rationales because plausible design judgements may differ from a narrow line oracle. Reporting should include per-task results, participant background, attrition, time on task, hint use, and exposure to each exercise.

Where class size permits, DevDog should be compared with the same explanations and examples without the reward and hint economy; otherwise, a crossover or matched-cohort design should document instructor and ordering effects. Analysis should report effect sizes and uncertainty, account for repeated measures, and inspect errors by smell. Consent, data minimization, a frozen exercise version, preregistered outcomes, and publication of assessment materials and aggregate results are also required. Until then, DevDog is an open practice environment whose educational effectiveness remains a testable hypothesis.

7 Conclusion

DevDog is a deployed and open platform for practicing code smell identification through contextual examples, explicit line–smell decisions, immediate feedback, progressive hints, and low-friction access. Its contribution is not a new detector or the first gamified smell-learning system; it is an integrated, reproducible configuration that keeps design judgement as the submitted answer and connects theory, practice, feedback, persistence, and per-exercise comparison in a short-challenge workflow.

The evidence remains formative. Next steps are independent oracle review, difficulty calibration, graphical authoring for instructors, region-based annotations, more languages, refactoring follow-up tasks, instructor dashboards, and classroom studies with unseen pre/post assessments. These extensions should preserve the distinction between engagement indicators and demonstrated learning.

Artifact Availability

DevDog, its source code, and the demonstration video are publicly available at https://devdog.cedis.tec.br, https://github.com/CedisUnB/CodeSmellGamification, and https://zenodo.org/records/20298757, respectively.

Acknowledgments

ChatGPT was used as a language-editing assistant and to suggest alternative wording for portions of Sections 2, 3.2, 5.2, and 6.2–6.3. The authors critically evaluated and verified all assisted content and assume full responsibility for the manuscript.

References

[1] W. Aljedaani, A. Ghammam, M. W. Mkaouer, and M. Kessentini. 2024. From Boring to Boarding: Transforming Refactoring Education with Game-Based Learning. In *Proceedings of GAS '24*. ACM, 20–27. DOI: 10.1145/3643658.3643924.

[2] J. L. Bez, N. A. Tonin, and P. R. Rodegheri. 2014. URI Online Judge Academic: A Tool for Algorithms and Programming Classes. In *Proceedings of ICCSE '14*. IEEE, 149–152. DOI: 10.1109/ICCSE.2014.6926445.

[3] Y.-K. Chou. 2015. *Actionable Gamification: Beyond Points, Badges, and Leaderboards*. CreateSpace. ISBN 9781511744041.

[4] S. Deterding, D. Dixon, R. Khaled, and L. Nacke. 2011. From game design elements to gamefulness: defining gamification. In *Proceedings of MindTrek '11*. ACM, 9–15. DOI: 10.1145/2181037.2181040.

[5] A. R. Fasolino and P. Tramontana. 2024. Test Smells Learning by a Gamification Approach. In *Proceedings of Gamify '24*. ACM, 30–33. DOI: 10.1145/3678869.3685687.

[6] M. Fowler. 2018. *Refactoring: Improving the Design of Existing Code* (2nd ed.). Addison-Wesley. ISBN 9780134757599.

[7] J. Hamari, J. Koivisto, and H. Sarsa. 2014. Does gamification work? A literature review of empirical studies on gamification. In *Proceedings of HICSS '14*. IEEE, 3025–3034. DOI: 10.1109/HICSS.2014.377.

[8] M. V. Mäntylä and C. Lassenius. 2006. Subjective evaluation of software evolvability using code smells. *Empirical Software Engineering* 11, 3 (2006), 395–431. DOI: 10.1007/s10664-006-9002-8.

[9] D. Porto, G. Jesus, F. Ferrari, and S. Fabbri. 2021. Initiatives and challenges of using gamification in software engineering: a systematic mapping. *Journal of Systems and Software* 173 (2021), 110870. DOI: 10.1016/j.jss.2020.110870.

[10] C. S. Ramos, M. A. S. Farias, S. A. A. Freitas, M. V. P. Martins, J. M. Alves, and L. C. S. Pinto. 2024. A Process to Identify Players' Motivational Profiles for Designing a Gamification Project. In *ICCSA 2024*. Springer, 49–67. DOI: 10.1007/978-3-031-64608-9_4.

[11] J. P. dos Reis, F. B. e Abreu, G. de F. Carneiro, and C. Anslow. 2020. Code smells detection and visualization: a systematic literature review. arXiv:2012.08842 [cs.SE].

[12] R. M. Ryan and E. L. Deci. 2000. Self-determination theory and the facilitation of intrinsic motivation, social development, and well-being. *American Psychologist* 55, 1 (2000), 68–78. DOI: 10.1037/0003-066X.55.1.68.

[13] M. Sailer, J. U. Hense, S. K. Mayr, and H. Mandl. 2017. How gamification motivates: effects of game design elements on psychological need satisfaction. *Computers in Human Behavior* 69 (2017), 371–380. DOI: 10.1016/j.chb.2016.12.033.

[14] H. M. dos Santos, V. H. S. Durelli, M. Souza, E. Figueiredo, L. T. da Silva, and R. S. Durelli. 2019. CleanGame: Gamifying the Identification of Code Smells. In *Proceedings of SBES '19*. ACM, 10 pages. DOI: 10.1145/3350768.3352490.

[15] T. F. M. Sirqueira, A. H. M. Brandl, E. J. P. Pedro, R. de Souza Silva, and M. A. P. Araújo. 2016. Code Smell Analyzer: A Tool to Teaching Support of Refactoring Techniques Source Code. *IEEE Latin America Transactions* 14, 2 (2016), 877–884. DOI: 10.1109/TLA.2016.7437235.

[16] SonarSource. 2026. What is SonarQube? https://docs.sonarsource.com/what-is-sonarqube. Accessed July 16, 2026.

[17] I. Tan and C. M. Poskitt. 2024. Fixing your own smells: adding a mistake-based familiarisation step when teaching code refactoring. arXiv:2401.01011 [cs.CY].

[18] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. 2015. When and why your code starts to smell bad. In *Proceedings of ICSE '15*. IEEE, 403–414. DOI: 10.1109/ICSE.2015.59.

[19] A. Yamashita and L. Moonen. 2013. Do developers care about code smells? An exploratory survey. In *Proceedings of WCRE '13*. IEEE, 242–251. DOI: 10.1109/WCRE.2013.6671299.

[20] M. Zakeri-Nasrabadi, S. Parsa, E. Esmaili, and F. Palomba. 2023. A systematic literature review on code smell datasets and validation mechanisms. *Journal of Systems and Software* 204 (2023), 111755. DOI: 10.1016/j.jss.2023.111755.