UnityLint: A Bad Smell Detector for Unity

Matteo Bosco,
Pasquale Cavoto,
Augusto Ungolo
University of Sannio
Benevento, Italy

Biruk Asmare Muse,
Foutse Khomh
Ecole Polytechnique de Montréal
Montréal, Quebec, Canada

Vittoria Nardone,
Massimiliano Di Penta
University of Sannio
Benevento, Italy

***Abstract*—The video game industry is particularly rewarding as it represents a large portion of the software development market. However, working in this domain may be challenging for developers, not only because of the need for heterogeneous skills (from software design to computer graphics), but also for the limited body of knowledge in terms of good and bad design and development principles, and the lack of tool support to assist them. This tool demo proposes UnityLint, a tool able to detect 18 types of bad smells in Unity video games. UnityLint builds upon a previously-defined and validated catalog of bad smells for video games. The tool, developed in C# and available both as open-source and binary releases, is composed of (i) analyzers that extract facts from video game source code and metadata, and (ii) smell detectors that leverage detection rules to identify smells on top of the extracted facts.**

Tool: https://github.com/mdipenta/UnityCodeSmellAnalyzer

Teaser Video: https://youtu.be/HooegxZ8H6g

***Index Terms*—Video game development; Unity; Bad smells; Static analyzer**

I. INTRODUCTION

While the pandemic period caused considerable losses to the global economy, the video game industry has continued to grow in a remarkable way [1]. In fact, the global video game market size is expected to expand at a compound annual growth rate (CAGR) of 12.9% from 2022 to 2030.

Developing video games follow practices that differ from conventional software development [2]–[4], as it requires specific skills and knowledge, often going beyond the common knowledge of a developer working on conventional software. During video game development, developers face several video game-specific aspects, *e.g.,* reproducing/simulating the environment's physics, animating objects, and rendering special effects. Such aspects make video game design and development complex and could negatively affect the quality (*e.g.,* in terms of performance) and development of produced software (*e.g.,* increasing maintenance costs). In this context, developers may need suitable guidance concerning good (and bad) design and development principles, but also appropriate tool support, *e.g.,* through analyzers helping them to avoid introducing performance bottlenecks, or making the game difficult to maintain and evolve.

The research community has investigated the application of design principles to video game development [5]–[11], and found that conventional code smells fail to capture all quality problems of video game source code [12]–[14]. For this reason, we first conceived a preliminary approach to detect five types of bad smells in Unity [15]. Then, we defined a catalog of 28 bad smells related to video game development [16], by manually analyzing developers' discussions on game engine forums, and by validating them through a survey with video game development professionals.

In this paper, we leverage this catalog and some of the previously defined smell detection approaches [15], and propose UnityLint, a tool to detect video game smells for the Unity video game development framework [17]. We target Unity since it is one of the most popular cross-platform game engines [18]. More specifically, UnityLint detects 18 bad smells out of the 28 (+1 reported in the survey, *i.e.,* Use of anystate in animator controller) defined in the catalog [16].

UnityLint works in two stages. First, analyzers extract facts from the video game source code (C#) and metadata. Then, the smell detector identifies smells by leveraging detection rules on the extracted facts. UnityLint has been conceived as a command line tool, yet the way it has been designed makes it possible to integrate it in continuous integration workflows (*e.g.,* in a GitHub action), or else a graphical front-end for IDEs. The tool has been preliminary evaluated using 70 Unity open-source projects.

There are different scenarios in which UnityLint could be used by both practitioners and researchers:

- Practitioners can leverage the tool during their development activities, *e.g.,* to produce warning reports or to even fail the build.
- Researchers can leverage the tool to conduct an empirical investigation on video game bad smells, on the same lines of how similar studies have been conducted for conventional software [19], [20].
- UnityLint can be easily extended in different ways. On

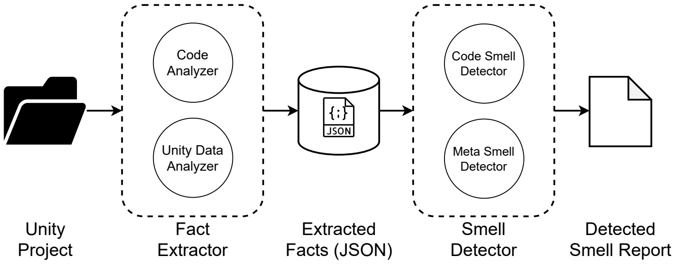

Fig. 1. UnityLint architecture

the one hand, it could be possible to implement further analyzers to support different programming languages and video game development frameworks. On the other hand, it is possible to add detectors for further smells.

UnityLint is available as open-source under the MIT License on GitHub [1], and also released in the form of binaries.

II. A SMELL DETECTOR FOR UNITY

Fig. 1 depicts the workflow of UnityLint. The tool consists of two main components: (i) *Fact Extractor*, which extracts and collects data needed for smell detection. This goal is achieved by analyzing source code and metadata files, storing the extracted facts in JSON files, and (ii) *Smell Detector* which, by taking as input the JSON files produced by the *Fact Extractor*, leverages detection rules for identifying video game smells. This two-steps architecture has multiple advantages. First, it facilitates the definition of new detection rules, implemented by querying the JSON files directly or through helper APIs (*e.g.,* to search for variable definition/usage, methods, etc.) the tool makes available, or even developing a detector based on machine learning approaches. Second, by implementing an analyzer that produces a compatible intermediate representation, it would be possible to apply the rules (sometimes as they are, sometimes with small changes) to analyze games developed with other game engines and programming languages. Last, but not least, the extracted facts can be leveraged for other purposes.

The *Fact Extractor* is composed of a *Code Analyzer* and a *Unity Data Analyzer*. The *Code Analyzer* parses the C# source code and produces a JSON representation containing information such as imported libraries, class structures, variables definition, and usages, or invoked functions. The *Unity Data Analyzer* processes files related to Unity assets often created from its IDE. These include scene files (containing static game objects and their dependencies), prefabs (*i.e.,* reused objects) animations, and other assets.

The *Smell Detector* is composed of a *Code Smell Analyzer* and a *Meta Smell Analyzer* and they implement rules described in Table I to identify the video game smells. In the current implementation, the tool detects 18 different video game smells among those empirically defined in a previous work [16]. Out of the 28 smells defined by [16], we did not implement 11 smells, either related to problems (mostly rendering and animation-related) that could not be detected by statically analyzing code and metadata, or to Unity modules (multiplayer in particular) currently being deprecated and replaced. Plus, we implemented a smell (use of anystate in animators) not part of the catalog but suggested by a practitioner during the catalog validation.

UnityLint is implemented in C# language and for the source code analysis leverages the Roslyn [21] compiler and its API. It works natively on the Windows operating system, or, through Mono [22], on Linux and MacOS.

Table I describes the detected smells, divided into their categories, and details the detection rule defined to identify them. The last two columns report the results of a preliminary evaluation.

III. UNITYLINT IN ACTION

UnityLint can be used in two ways, *i.e.,* through its wrapper (named `ShellStarter`), that runs the whole toolchain on a given folder (which contains a Unity project), or (ii) running the single components (*Code Analyzer*, *Unity Data Analyzer*, *Code Smell Analyzer*, and *Meta Smell Analyzer*) individually. The former is useful to run UnityLint with default options and to analyze multiple projects (*e.g.,* to conduct a study), whereas the latter can be useful to specify advanced options of individual tools (*e.g.,* restrict the set of smells to use, or change the output options), or if one is interested in using only some of them. In the first case, the tool can be executed, for example, by running (“mono” is only for *nixes OSs):

```
 mono UnityLinter/ShellStarter.exe -d
/games/ -v
```

where the `-d` switch specifies the directory where the Unity projects to analyze are located, and `-v` enables the verbose output (otherwise the tool is silent). UnityLint stores both the intermediate outputs in the Results/Examples directory. For example, the file `Results/Example/Code/CodeAnalysis.json` stores the result of source code analysis in JSON. Listing 1 shows an excerpt related to a method invocation (`cubeRef.transform.Rotate`).

```
{ "Name": "Update",
      ....
  "ReturnType": "void",
  "Parameters": [],
  "Invocations": [
  "Name": "cubeRef.transform.Rotate",
  "FullName": "cubeRef.transform.Rotate",
```

Listing 1: CodeAnalysis.json excerpt (method invocation)

Under `Results/Example/Code/SmellResults`, the tool creates a JSON file for each code smell type, for example Listing 2 shows a weak temporization due to the invocation of a *transform* without `Time.deltaTime` scaling, and obtained by analyzing a longer version of the code in Listing 1.

```
"Name": "Weak Temporization",
"Occurrency": 1,
"Smells": [
            {  "Script": "..../RotateCube.cs",
               "Name": "Update",
               "Line": 17 }
          ]
```

Listing 2: Example of weak temporization detection

Listing 3 shows an example of data extracted by the *Unity Data Analyzer* from the Unity assets and stored under `Results/Data/mainResults`, in this case some properties of a *Rigidbody* attached to a game object (*i.e.,* the use

---

[1] https://github.com/mdipenta/UnityCodeSmellAnalyzer

TABLE I
SMELLS DESCRIPTION AND DETECTION RULES.

Name	**Description**	**Rule**	**Pr**	**Rc**
		Design and Game Logic		
Bloated assets	Reusable assets containing a suspiciously high number of components	The number of total components into metadata is greater than a threshold value	100%	100%
Creating components/objects at run-time	Game objects are created/destroyed at every frame instead of using an object pool	Instantiate and Destroy methods in *Update()*, *FixedUpdate()*, or *LateUpdate()* methods	100%	100%
Dependencies between objects	There is a strong dependency between all classes present in the scripts	All local and instance variables in conjunction with *GetComponent* methods invocations and variables types belong to other classes	86%	93%
Lack of separation of concerns	The game logic does not clearly separate concerns related to inputs, physics, rendering, etc.	Use, in the same script, of different Unity modules, e.g., animators and inputs	54%	75%
Poor design of object state management	Complex game object state management without using appropriate design solutions, e.g., the state pattern	Nested and complex conditional statements (*i.e.,* if, if-else, switch-case) within the game's main loop	69%	92%
Static coupling	Dependencies between gameobjects created visually through the IDE	Identification of [SerializedField] object attributes and analysis of dependencies in Scene metadata	78%	97%
Search by string/ID	Game objects/components are searched at run-time using their string identifier/tag	Game objects/components are searched using *Find* methods within the game main loop	100%	–
Singleton vs. static	Use of singleton where a static variable would just suffice	Detection of singleton design pattern by checking the class constructor and attributes	100%	–
Weak temporization strategy	Game object *transform* depends on the frame rate	*Update()* and dependent methods use transform without scaling values with *Time.deltaTime*	86%	100%
		Animation		
Continuously checking position/rotation	A game continuously checks whether the object is within a boundary	Checking (directly or indirectly) a *transform position/rotation* parameters into conditional statements within the game's main loop	–	–
Multiple Animators over model component	A game object uses multiple animators or components handling animations for the same reusable object	Searching for game objects having more than one Animator or animation-related components	100%	100%
Too many key frames in animations	An animation contains too many keyframes	Into animation metadata, the variable *m_Curve* has a number of *time* values greater than a threshold	100%	100%
Use of anystate in animator controller	Animators have transitions that can start from an undetermined state	Presence of outgoing state transitions from anystate state	100%	100%
		Physics		
Heavyweight physics computation	A game performs heavyweight physics computation in ints main loop	Checking if the game object physic is modified within the Update method	100%	100%
Improper mesh settings for a collider	A sub-optimal choice of collider for a game object	Using collider custom (*i.e.,* Mesh Collider) instead of simple collider type provided by Unity	100%	100%
Setting object velocity and override forces	Objects' velocity is directly modified, instead of operating through Forces/Physics	The values of *velocity* and/or *angularVelocity* of a *Rigidbody* object are directly modified into scripts	–	–
		Rendering		
Lack of optimization when rendering objects	Object drawing/rendering not properly optimized	Searching (in the metadata) for the *m_EnableRealtimeLightmaps* parameter with an assigned value greater than 0	100%	100%
Sub-optimal, expensive choice of lights, shadows, or reflections	Some lights that can be baked are, instead, rendered in real-time, or when there is excessive usage of (unnecessary) shadows and reflections	Static (not animated) object emitting a real-time light; Objects with animation script emitting a baked light	100%	100%

of gravity and a collision detector set to 2, *i.e.,* continuous-dynamic). Then, under

```
Results/Example/Data/MetaSmellResults,
```

the tool creates a JSON file for each smell detected from the Unity meta data, for example, Listing 4 shows an example of heavy physics computation (also) resulting from the analysis of the facts reported in Listing 3.

If one wants to run the tools individually (recommended to set specific options), for example the command:

```
mono CSharpAnalyzer.exe -p projects/War
-s -r MyRes
```

which executes the source code analysis analyzing the project (`-p`) in the directory projects/War, embedding the raw text of statements (`-s`) in the JSON output near each construct storing results (`-r`) in the MyRes directory. Then,

```
mono CodeSmellAnalyzer.exe -d
MyRes1/CodeAnalysis.json -f smellList.txt
-r MyRes1
```

detects code smells from the results of the code analysis specified by the `-d` option. The `-f` option allows specifying a (restricted) list of smells to detect.

The *Unitty Data Analyzer* can be invoked using:

```
guid": "65d94dcccbb9e4c46962c78ae98ca414",
"file_path": ".../Cylinder.prefab",
"name": "Cylinder",
"type": "prefab",
.....
"id": "7942783696874373333",
"Rigidbody": [
....
{ "m_UseGravity": "1" },
{ "m_CollisionDetection": "2"}
...
```

Listing 3: Example of Rigidbody data

```
guid": "65d94dcccbb9e4c46962c78ae98ca414",
"file_path": ".../Cylinder.prefab",
"name": "Cylinder",
"type": "prefab",
...
```

Listing 4: Example of heavy physics computation

```
mono UnityDataAnalyzer.exe -d
projects/War/Assets/Prefabs -r out
```

In this case, we specify the assets to be analyzed, and are only analyzing the Prefabs from the project (and not other assets). Finally, the smell detector for metadata can be executed through the command:

```
mono MetaSmellAnalyzer.exe -d out -c -r
outDataSmells
```

Further details about the syntax of the individual tools can be found in the project README file.

IV. Preliminary Evaluation

We have performed a preliminary evaluation of UnityLint. To evaluate the precision, we have detected smells on 70 open-source Unity projects hosted on GitHub. More in detail, we selected C# projects with more than 100 commits and at least one commit since October 2021, and excluded forks to avoid duplicates. We queried projects using the tool provided by Dabic *et al.* [23]. Then, the subset obtained using the above query is further filtered using the project topic list. We selected projects with videogame-related topics, in particular “Unity” and “Videogame”. Finally, the remaining projects are manually inspected to select only projects developed using Unity Engine.

Then, we ran UnityLint on the projects. For the smells that require a threshold (*e.g., Too many key frames*, see the README in the tool repository) we considered the third quartile of the values measured in the repository. From the detected smells, we extracted a statistically significant (95% confidence level, ±5% confidence interval) sample of 377 smells, stratified over the smell types.

It should be underlined that we computed this sample excluding the occurrences of the *Dependency Between Objects’* smell. We excluded it since this type of smell has a high number of occurrences compared with other types and including this number into the computation of a statistically-significant sample resulted in an unbalanced stratified sample, *i.e.,* the majority of samples to validate belonged to this type of smell leaving the other type with a few numbers of samples. Thus, we assessed *Dependency Between Objects’* smell separately from the other smell types, randomly selecting for it 290 samples (95% confidence level, ±5% confidence interval from the population of that smell). In total, we manually validated 667 samples.

The precision assessment has been performed by two authors not involved in the tool implementation. The two authors independently assessed each smell in the sample and discussed disagreements. We computed the Cohen’s k [24] inter-rater agreement which resulted to be 0.56 (moderate).

To evaluate the recall, three authors manually inspected 6 projects (by looking at the source code and visually inspecting the other artifacts through the Unity IDE) to identify possible smells. The manually identified smells were then compared against those detected by UnityLint.

Table I reports the precision (Pr) and the recall (Rc), for each smell type, achieved on the manually analyzed instances. As the table shows, besides the generally good performances, there are some smells that were not detected in our dataset. Therefore, for them we do not have an empirically-assessed accuracy, yet we have carefully tested the detectors through multiple code examples. Also, for some smells (*Lack of separation of concerns* or *Poor design of object state management*) the precision is lower. For the former, UnityLint indicates possible excesses of mix-ups (*e.g.,* controller handling and animations in the same script), yet some of them could be intentional and hard to separate. For the latter, we notice (further studies are needed though) how developers simply prefer to design game state management with cascades of conditional statements. Also, we plan to improve the detection of these smells with further heuristics.

We also computed micro and macro precision and recall, where micro precision and recall weigh the occurrences of different smells (*i.e.,* the numerator is the number of true positives of that class), whereas macro precision and recall are the mean precision and recall across the different smells. Their values are 78% micro precision, 92% macro precision, 94% Micro Recall, and 85% macro recall.

V. Conclusion and Future Work

This paper described UnityLint, a smell detector toolkit for Unity. UnityLint is based on a subset of an empirically derived catalog of bad smells for video games [16]. UnityLint detects 18 smells among those defined in the catalog, and in a preliminary evaluation has achieved a micro-precision of 78%, a macro-precision of 92%, a micro-recall of 94%, and a macro-recall of 85%.

UnityLint can be used by developers as a linter, *e.g.,* by integrating it in continuous integration pipelines, as well as by researchers for studying the quality of Unity projects.

Future work aims at (i) further improving the detection rules, (ii) detecting further smells, (iii) porting the tool to other video game development frameworks (*e.g.,* Unreal).

REFERENCES

[1] Grand View Research, "Video game market size, https://www.grandviewresearch.com/industry-analysis/video-game-market (last access: 20/12/2022)," 2022.

[2] B. B. Marklund, H. Engström, M. Hellkvist, and P. Backlund, "What empirically based research tells us about game development," *Comput. Games J.*, vol. 8, no. 3-4, pp. 179–198, 2019. [Online]. Available: https://doi.org/10.1007/s40869-019-00085-1

[3] E. R. Murphy-Hill, T. Zimmermann, and N. Nagappan, "Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development?" in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. ACM, 2014, pp. 1–11. [Online]. Available: https://doi.org/10.1145/2568225.2568226

[4] A. V. Kamienski and C. Bezemer, "An empirical study of q&a websites for game developers," *Empir. Softw. Eng.*, vol. 26, no. 5, p. 115, 2021. [Online]. Available: https://doi.org/10.1007/s10664-021-10014-4

[5] A. Ampatzoglou and A. Chatzigeorgiou, "Evaluation of object-oriented design patterns in game development," *Information and Software Technology*, vol. 49, no. 5, pp. 445–454, 2007.

[6] N. H. Barakat, "A framework for integrating software design patterns with game design framework," in *Proceedings of the 2019 8th International Conference on Software and Information Engineering*, 2019, pp. 47–50.

[7] G. R. Figueiredo and G. L. Ramalho, "Gof design patterns applied to the development of digital games," *Proceedings of SBGames*, vol. 15, 2015.

[8] X.-C. Kounoukla, A. Ampatzoglou, and K. Anagnostopoulos, "Implementing game mechanics with gof design patterns," in *Proceedings of the 20th Pan-Hellenic Conference on Informatics*, 2016, pp. 1–4.

[9] J. W. Murray, *C# game programming cookbook for Unity 3D*. CRC Press, 2014.

[10] R. Nystrom, *Game programming patterns*. Genever Benning, 2014.

[11] J. Qu, Y. Song, and Y. Wei, "Applying design patterns in game programming," in *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2013, p. 1.

[12] V. Khanve, "Are existing code smells relevant in web games? an empirical study," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 2019, pp. 1241–1243. [Online]. Available: https://doi.org/10.1145/3338906.3342504

[13] V. Agrahari and S. Chimalakonda, "A catalogue of game-specific anti-patterns," in *ISEC 2022: 15th Innovations in Software Engineering Conference, Gandhinagar, India, February 24 - 26, 2022*. ACM, 2022, pp. 8:1–8:10. [Online]. Available: https://doi.org/10.1145/3511430.3511436

[14] G. C. Ullmann, C. Politowski, Y. Guéhéneuc, F. Petrillo, and J. E. Montandon, "Video game project management anti-patterns," *CoRR*, vol. abs/2202.06183, 2022. [Online]. Available: https://arxiv.org/abs/2202.06183

[15] A. Borrelli, V. Nardone, G. A. Di Lucca, G. Canfora, and M. Di Penta, "Detecting video game-specific bad smells in unity projects," in *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*. ACM, 2020, pp. 198–208. [Online]. Available: https://doi.org/10.1145/3379597.3387454

[16] V. Nardone, B. Asmare Muse, M. Abidi, F. Khomh, and M. Di Penta, "Video game bad smells: What they are and how developers perceive them," *ACM Trans. on Software Eng. and Methodology*. [Online]. Available: https://mdipenta.github.io/files/tosem-gamesmells.pdf

[17] Unity, "Unity engine, https://unity.com (last access: 20/12/2022)," 2022.

[18] "2021 gaming report - unity, https://create.unity.com/2021-game-report," accessed: 2023-03-22.

[19] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Trans. Software Eng.*, vol. 43, no. 11, pp. 1063–1088, 2017.

[20] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empir. Softw. Eng.*, vol. 23, no. 3, pp. 1188–1221, 2018.

[21] "The .net compiler platform (Roslyn API), https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/," accessed: 2023-01-13.

[22] "Mono: an open source implementation of microsoft's .net, https://www.mono-project.com/," accessed: 2023-01-13.

[23] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 2021, pp. 560–564.

[24] J. Cohen, "A coefficient of agreement for nominal scales," *Educ Psychol Meas.*, vol. 20, 1960.