Detecting Video Game-Specific Bad Smells in Unity Projects

Antonio Borrelli
University of Sannio
Benevento, Italy
aborrelli@unisannio.it

Vittoria Nardone
University of Sannio
Benevento, Italy
vnardone@unisannio.it

Giuseppe A. Di Lucca
University of Sannio
Benevento, Italy
dilucca@unisannio.it

Gerardo Canfora
University of Sannio
Benevento, Italy
canfora@unisannio.it

Massimiliano Di Penta
University of Sannio
Benevento, Italy
dipenta@unisannio.it

ABSTRACT

The growth of the video game market, the large proportion of games targeting mobile devices or streaming services, and the increasing complexity of video games trigger the availability of video game-specific tools to assess performance and maintainability problems. This paper proposes UnityLinter, a static analysis tool that supports Unity video game developers to detect seven types of bad smells we have identified as relevant in video game development. Such smell types pertain to performance, maintainability and incorrect behavior problems. After having defined the smells by analyzing the existing literature and discussion forums, we have assessed their relevance with a survey involving 68 participants. Then, we have analyzed the occurrence of the studied smells in 100 open-source Unity projects, and also assessed UnityLinter's accuracy. Results of our empirical investigation indicate that developers well-received performance- and behavior-related issues, while some maintainability issues are more controversial. UnityLinter is, in general, accurate enough in detecting smells (86%-100% precision and 50%-100% recall), and our study shows that the studied smell types occur in 39%-97% of the analyzed projects.

CCS CONCEPTS

• **Software and its engineering → Application specific development environments**;

KEYWORDS

Video Game Development; Bad Smells; Static Analysis; Linters

ACM Reference Format:
Antonio Borrelli, Vittoria Nardone, Giuseppe A. Di Lucca, Gerardo Canfora, and Massimiliano Di Penta. 2020. Detecting Video Game-Specific Bad Smells in Unity Projects. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.3387454

1 INTRODUCTION

Video games represent a conspicuous and increasing share of the software development market. In 2018, the video game industry has generated 134.9 billion dollars, with over 10% increase over 2017 [25]. Such a market is changing continuously also in terms of platforms on which video games are deployed. In the past, video games mainly targeted consoles and desktop computers; nowadays mobile devices account for nearly half of the market [24], and the current trend is the streaming of video game contents.

While the video game market is increasing, development skills in this area still represent a niche. Just to give an idea, Stack Overflow features over 1.5M discussions tagged [java] and 1.2M tagged Android, while only 50k are about Unity3D. It is therefore clear how in this context developers may need suitable support while creating their video games, helping them to avoid introducing performance bottlenecks, or making the game difficult to maintain and evolve.

Static code analysis tools (SCAT) are a typical support developers have while coding. Such tools, known also as "linters" (from the first tool developed by Johnson for the C language [28]) analyze the source code or the compiled (*e.g.,* bytecode) program to highlight several problems. These include, among others, likely bugs (*e.g.,* type conversions, or potentially wrong operators applied to certain types), performance issues (*e.g.,* use of programming constructs that are known to be inefficient), security vulnerabilities, or coding style issues (*e.g.,* inadequate commenting or choice of identifiers). The use of SCAT in software development has been investigated by several studies. For example, Johnson *et al.* [27] and Beller *et al.* [18] have studied the usage of static analysis tools in software projects. Other studies focused on investigating the extent to which such tools can be used to detect real faults [45, 48, 50], or how they are used in continuous integration pipelines [49].

Many SCAT are general-purpose, *e.g.,* FindBugs [5], PMD [7] or CheckStyle [4] for Java, Rubocop [1] for Ruby, or Splint [22] for C. Others analyze specific types of applications, *e.g.,* Android Lint [2] detects Android-specific issues.

In this paper, we propose UnityLinter, a linter for video games developed with Unity [10]. While there are many other video game development frameworks (*e.g.,* Unreal [11] or Blender [3]), we have chosen Unity because it is free (within certain usage limits) and for this reason, it has also been adopted in the open-source community as well as for educational purposes. UnityLinter statically analyzes the source code (written in C#) and other artifacts of a Unity video games, and can detect 7 types of video game code smells. Such smells cover different quality aspects of video game development, namely performance, maintainability, and correct behavior.

We first elicited a set of 6 smells, by analyzing existing literature on video game design [38, 40], by discussing with experts, and by analyzing discussions on forums. Then, we surveyed 68 video game developers and gathered their perception and feedback about the smells we wanted to detect. This also helped us to refine the smell detection rules and to detect an additional smell over the 6 we initially conceived. Finally, we analyzed 100 Unity open source projects hosted on GitHub, in order to (i) detect smells and therefore assess the magnitude of the investigated phenomenon, and (ii) manually validate a statistically representative sample of 420 smells to evaluate UnityLinter's precision, and inspect five complete video games to evaluate UnityLinter's recall.

Results of the study indicate that, overall, developers well perceive the smells we defined, in particular, those related to performance and game behavior, while maintenance-related smells are more controversial. Our manual validation indicates that UnityLinter achieves, for the different smell types, 86%-100% precision and 50%-100% recall. Depending on the type, the presence of the studied smells in the analyzed projects varies between 39% and 97% of the analyzed projects.

The contribution of this paper can be summarized as follows:

(1) we define 7 types of bad smells that may occur in video game development validated through a study;
(2) we propose UnityLinter, a linter for video games developed in Unity;
(3) we report the results of a study assessing the diffuseness of the studied smells, within the accuracy limit of UnityLinter;
(4) we make the dataset of the study available [19].

2 BACKGROUND NOTIONS ABOUT UNITY

This section provides a brief introduction to Unity, to let the reader properly understanding the concepts introduced throughout the paper. Fig. 1-a shows a screenshot of the Unity development environment. A Unity project can either be a 2D or 3D project (in the following we will mainly show examples for 3D projects, as 2D is mostly a subset of 3D) composed of one or more *scenes*. A scene describes an interaction area, *e.g.,* a piece of the game world where a player interacts with other objects. The scene, in turn, contains *GameObjects* that are instantiated either statically or dynamically. In our example, the project includes a scene `SampleScene`, whose object hierarchy is visible on the left-side pane of the IDE. More specifically, a scene contains a camera (`Main Camera`), which is the perspective from which the user sees the scene, a light (`Directional Light`), and a `Cube` object (also visible at the center of the screen).

Other than using the typical extension mechanisms of object-oriented (OO) programming, Unity allows developers to decorate an object with various kinds of components, that can be visible on the right-side pane (*i.e.,* the Inspector). More specifically, an object has a default property named `transform`, which specifies the object position, rotation, and scaling, and other components, *e.g.,* a collider that handles collisions with other objects, a material (in our case named MyColor) that specifies the object's visual aspect. Last, but not least, it is possible to add one or more scripts to specify behavior. In our case, the cube has attached a script named *Move Cube* (whose content is shown in Fig. 1-b).

Unity's behavior is specified using scripts written in C#[1]. A relevant class in Unity is `MonoBehaviour`, and C# classes attached to GameObjects inherit from `MonoBehaviour`. A Unity running project works as a loop. First, the `Start()` method of all `MonoBehaviour` classes attached to GameObjects is executed. Then, the `Update()` method is called in each frame. Note that Unity features variants of `Start()` and `Update()` (all covered by UnityLinter), called at different time, *e.g.,* `Awake()` is similar to `Start()`, but is called when a script is loaded, while the `LateUpdate()` methods are called after all `Update()` methods have been invoked. `FixedUpdate()` is invoked with a fixed frequency, *i.e.,* it does not depend on the frame rate.

In the script shown in Fig. 1-b, the `Start()` method gets the reference to a component of type *Material* attached to the cube. By changing one of its properties, the cube color is changed upon loading the scene. In each frame (`Update()` method), a rotation of one degree over the z-axis is applied to the cube. Besides rotating the cube, the `Update()` also dynamically instantiates another object (a bullet) using the `Instantiate` method. In this case, the reference to the GameObject is not available in the code but, rather, the object is loaded from a prefab. In Unity a prefab is a reusable asset (typically a GameObject or a composition of GameObjects with other attached components) stored in a `.prefab` file. A developer can create a prefab by simply dragging a GameObject from a project's hierarchy to the project resources (lower pane).

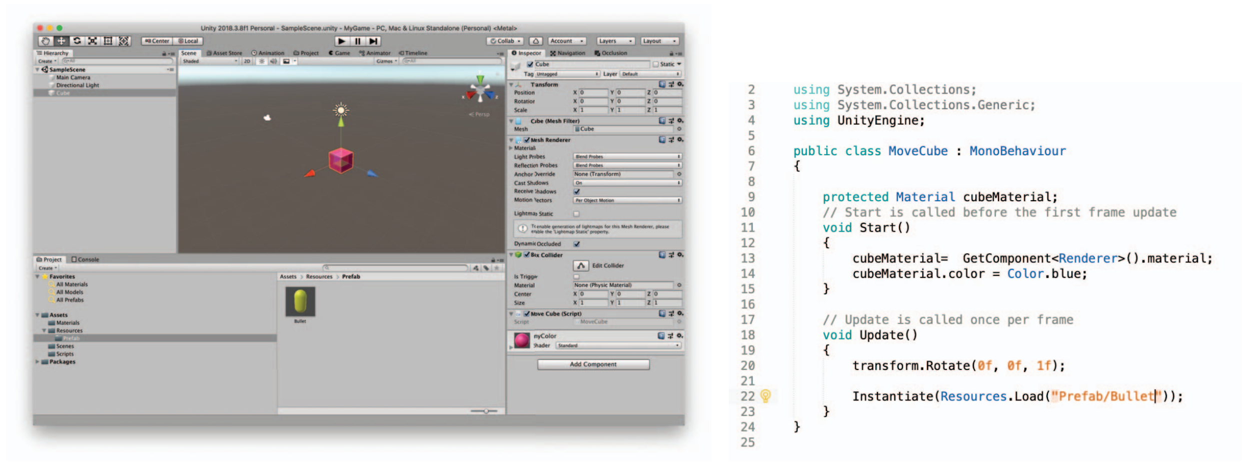
3 ELICITATION OF RELEVANT VIDEO GAME SMELLS

While it is not a goal of this paper to identify any possible smell that could affect a Unity video game, before implementing UnityLinter we conducted a small investigation to determine what kinds of smells to detect. To this aim, we relied on multiple sources. Specifically we (i) discussed our goal with experts in video game development, (ii) relied on the content of some textbooks [38, 40], highlighting good practices and patterns in video game development, and, finally, (iii) we leveraged informal knowledge from discussion forums, namely the Unity Forum, some Reddit channels related to video games and other Unity development forums. In particular, to mine potential smell-related discussions, we performed a search on the forums using the following keywords: (i) "good practices", (ii) "bad practices" and (iii) "common mistakes". On the Unity Forum, we analyzed discussions in the "support" and "community" sections. The former collects Unity official manuals, the latter features questions from the developers' community. Besides discussions partially referring to possible bad practices, we found 29 pages (reported in the online appendix), among those in the Unity Forum and Reddit, specifically targeting bad practices in Unity development.

Based on the various sources of information, we found the following problems being discussed by developers, and also mentioned in video game design books:

- The use of `Find` methods in runtime code, and, in general, operations on string literals;
- Instantiating and destroying game objects too often;

[1]Until 2017, Unity also supported Javascript.



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MoveCube : MonoBehaviour
{

    protected Material cubeMaterial;
    // Start is called before the first frame update
    void Start()
    {
        cubeMaterial=  GetComponent<Renderer>().material;
        cubeMaterial.color = Color.blue;
    }

    // Update is called once per frame
    void Update()
    {
        transform.Rotate(0f, 0f, 1f);

        Instantiate(Resources.Load("Prefab/Bullet"));
    }
}

```

(a) Unity IDE

(b) Monobehaviour script

Figure 1: Unity overview: a) The Unity IDE; b) An example of MonoBehaviour script.

- The presence of performance-intensive tasks in `Update()` loops;
- Frame rate not properly taken into account when scripting logic or animation;
- The use of public global variables, affecting information hiding; and
- Lack of separation of concerns, *i.e.,* creating incohesive scripts with too many responsibilities.

Based on such considerations, we defined 6 smell types. This choice has been driven by different factors: (i) need to cope with problems that are specific to Unity and video games, and not with generic issues, such as lack of information hiding, that could affect any software project; (ii) capability to define a heuristic to detect the smell statically, even if the detection is approximate; and (iii) considering smells that cover a range of different kinds of symptoms Unity projects could exhibit.

In the following, we briefly describe the 6 types of smells we have identified. We categorize them according to their negative effect they may cause, *i.e.,* on performance, maintainability and behavior.

3.1 Performance Smells

Allocating and destroying GameObjects in updates. In Unity, GameObjects can be allocated statically, *i.e.,* dragging them in the scene hierarchy through the IDE, or dynamically, *i.e.,* instantiating them using the `Instantiate` as explained in Section 2. When objects are no longer needed, they could be destroyed using the Destroy method. Both `Instantiate` and `Destroy` are computationally expensive, and their use in `Update()` method might affect the game performances. If the game requires to allocate and destroy many objects at runtime (*e.g.,* bullets in a shooting game) it is advised to use an object pool. In other words, a pool of GameObjects is allocated at runtime and, when required, GameObjects are retrieved from the pool and disposed there when no longer needed. The source code in Fig. 1-b exhibits the described smell, as the bullet is dynamically load and instantiated at every frame.

Getting a GameObject reference finding it by name. In Unity, a GameObject could gets the reference of another one by searching it by name. In the example of Fig. 1-a, a GameObject could get the reference of the Cube using the instruction:

```
GameObject.Find("Cube")
```

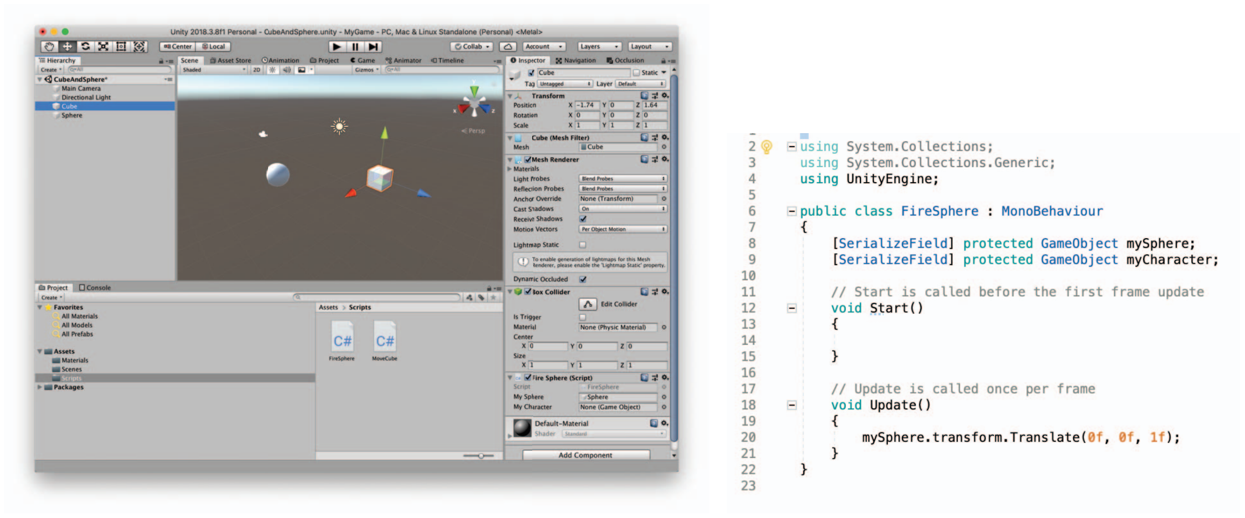
This access method is discouraged because it affects performances. A similar problem (`FindViewById`) is known also in Android [47]. Therefore, it is preferable to avoid this reference mechanism, and get access to other objects through explicit dependencies instead.

Heavyweight Update methods. Since all `Update()` methods are invoked at every frame, performing computationally-intensive operations in such methods might negatively affect performances. Sometimes this is just unavoidable; however, when possible, it is advised to factor out computations that do not require to be executed every time, moving them.

3.2 Maintainability Smells

Lack of separation of concerns. A MonoBehaviour script that implements at the same time different responsibilities makes the projects difficult to evolve. For example, let us imagine that a GameObject implements a player (*e.g.,* a person walking in an open world). A typical mistake is to implement all the logic of the player in the same MonoBehaviour script, *i.e.,* getting the inputs from the controller, determining the object state (*e.g.,* "can walk", "can jump", "can fire", etc.), and moving the GameObject. It is advisable, instead, to factor out input handling in a separate class hierarchy (using a Strategy design pattern) so that the source code attached to the player object does not have to change if, for example, one wants to implement an artificial intelligence-handled player. Similarly, the state management and the player animation should be handled in separate scripts.

Coupling objects through the IDE Inspector. In Unity, it is possible to couple a MonoBehavior script to other objects through the IDE. This works as follows. If a script declares public fields, or private/protected fields with the `[SerializedField]` attribute,



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class FireSphere : MonoBehaviour
{
    [SerializeField] protected GameObject mySphere;
    [SerializeField] protected GameObject myCharacter;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        mySphere.transform.Translate(0f, 0f, 1f);
    }
}
```

(a) (b)

Figure 2: Example of static coupling between GameObjects through the IDE Inspector.

such fields will appear as script properties in the Unity inspector. Then, from the IDE, the developer can just drag objects into such properties to create a coupling. In the example of Fig. 2-b, the MonoBehaviour script has two `[SerializedField]` attributes, `mySphere` and `myCharacter`. As shown in Fig. 2-a, both appear as properties in the inspector (right pane). Since the developer has dragged the Sphere into the property, the latter appears filled with a link to the Sphere object. Instead, there is no object linked to the `myCharacter` property.

The main negative effect of this programming practice is the lack of understandability, *i.e.,* couplings will not be visible from the source code, but only from the Inspector. Much worse, if a developer edits the source code, or renames scripts or objects, couplings are lost, and they need to be restored manually. There are alternatives to this, including the use of a messaging system to couple different objects in the game. However, such alternatives might be, in some cases, sub-optimal for what concerns performances.

3.3 Behavioral Smells

Animation speed depends on the frame rate. If a script performs a translation, rotation, or scaling of a fixed magnitude to an object in an Update method, such a transformation is repeated at every frame. As a result, the animation might be more or less fast depending on the frame rate. This smell is also evident in the Sphere translation of Fig. 2-b. To avoid the smell, a typical solution adopted is to multiply the magnitude by `Time.deltaTime`, which returns the delta time between two subsequent frames. In other words, the source code in Fig. 2-b becomes:

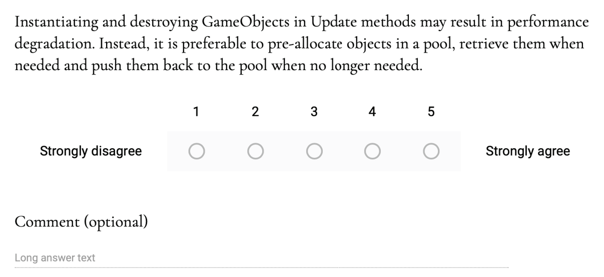
```
mySphere.transform.Translate(0f, 0f,
          1f*Time.deltaTime);
```

In such a way, the animation speed will be independent of the frame rate.

4 SMELL RELEVANCE ASSESSMENT

After having identified possible bad smells to detect, we wanted to investigate whether they are considered as relevant by developers, and possibly worthwhile of being addressed. Therefore, we address our first research question:

RQ$_1$: *To what extent are the considered bad smells relevant for video game developers?*



Instantiating and destroying GameObjects in Update methods may result in performance degradation. Instead, it is preferable to pre-allocate objects in a pool, retrieve them when needed and push them back to the pool when no longer needed.

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

Comment (optional)

Long answer text

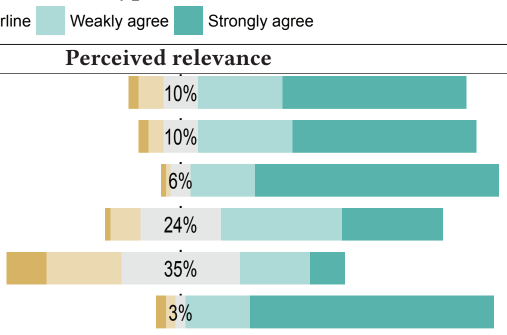
Figure 3: RQ$_1$ survey questionnaire: example of question.

4.1 Relevance Assessment Methodology

To address **RQ$_1$**, we conduct a survey with experts in video game development. The survey is composed of five sections:

- A first section, where demographic information about respondents is collected. More specifically, we asked about: (i) the domain in which they work; (ii) their role in the organization (*e.g.,* developer, project manager); (iii) the years of experience in software development; and (iv) more specifically, the years of experience in development with Unity.
- Three sections in which we ask developers to provide their perceived relevance about smells related to *Performance*, *Maintainability*, and *Behavior*. For each smell, we provide a brief description of the problem, outlining its consequences

Table 1: Perceived relevance of the 6 smell types.



Smell	Perceived relevance		
Allocating and destroying GameObjects in updates	10%	10%	79%
Heavyweight Update methods	7%	10%	82%
Getting a GameObject reference finding it by name	3%	6%	91%
Lack of separation of concerns	10%	24%	66%
Coupling objects through the IDE Inspector	34%	35%	31%
Animation speed depends on the frame rate	6%	3%	91%

and ways to avoid or mitigate the smell. Then, we ask to provide a relevance score in a 5-level Likert scale [41]. Finally, for each question, the respondent could add an optional open comment. An example of question for the smell "Allocating and destroying GameObjects in updates" is reported in Fig. 3.

- A final section with questions about the general, perceived usefulness of a Unity linter, *i.e.,* (i) whether they perceive the availability of such a tool useful, and (ii) whether they would be willing to adopt it (note: somebody could perceive it as useful, but not for her/him, just for some categories of users, *e.g.,* junior developers). Finally, we ask to provide free comments about possible smells the respondent perceive as relevant but that was not considered in our study.

To recruit participants, we posted the questionnaire on *Reddit* channels related to video games and Unity development, namely *gamedev* and unity3d. When we posted the questionnaire, we added a short message explaining its purpose, the estimated duration (10-15 min.) and a message telling that we would only use the collected data in aggregated, anonymized form.

We report the results of RQ_1 by showing the perceived relevance in form of diverging stacked bar charts. We also discuss the open comments made by the respondents.

4.2 RQ_1: Relevance Assessment Results

After two weeks of keeping the survey open, we obtained a total of 68 responses. 26 respondents had less than 5 years of development experience, 26 between 5 and 10 years, and 15 more than 10. About Unity, 47 respondents had less than 5 years of experience, 19 between 5 and 10, and 1 more than 10. One did not provide any answer to demographic information. Most of the respondents (61) were professional developers, besides 3 product owners, 2 students, and one manager. In terms of development domain (multiple answers allowed), most of the answers mentioned "Video game development" (48) and Virtual reality (15). Others include generic software development/consulting (6), computer graphics (4), content platform provider (3), or medical software (2).

Table 1 reports, in the form of diverging stacked bar charts the perceived relevance of the 6 smell types. The three percentages shown in the graph indicate the proportion of disagreements, neutral responses, and agreements respectively. Also, dark/light green indicates the proportion of "Strongly agree" and "Agree" responses, whereas dark/light brown indicates the proportion of "Strongly disagree" and "Disagree" responses.

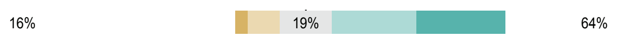
Allocating and destroying GameObjects in updates. Respondents generally agree (79% of positive answers) about the usefulness of using object pools instead of allocating/destroying in `Update()` methods. At the same time, some respondents were neutral or even disagreed. For some of them, the object pool may even introduce bugs in the implementation, whereas others indicated this is a good practice, but they apply it only where the specific implementation requires it. In other words, an occasional allocation/destroy is acceptable, whereas a massive allocation/destruction of objects makes the object pool worthwhile. Also, pool size and allocation frequency create a trade-off between using the pool or not, because a large pool might result in a waste of allocated memory. Finally, one respondent mentioned how the more recent C# runtime makes garbage collection (activated upon `Destroy`) more efficient than before.

Heavyweight Update methods. In this case, 82% of the respondents agree about the smell. At the same time, they also indicate that (i) as expected, a proper analysis of this smell requires a run-time profiler, (ii) Unity handles long call stacks pretty well, (iii) nested loops may be a good indicator of potential problems, but only when the level of nesting is over 3-4. Finally, respondents suggest using coroutines (a Unity mechanism for multi-threaded execution) to alleviate this problem.

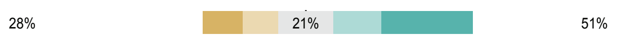
Getting a GameObject reference finding it by name. Almost all respondents (91%) indicate that the use of `Find` is a bad practice. Besides performance-related effects, `Find` also makes the source code fragile in case somebody renames objects. One respondent indicated that the use of `Find` can still be a good practice when retrieving a reference in an object hierarchy (for example, retrieving the reference to the head in a humanoid model from the upper-level container object). Respondents recommended creating coupling through the inspector, instead.

Lack of separation of concerns. Respondents generally agree (66%) about the usefulness of separating concerns. However, there is a substantial percentage (24%) of neutral responses, where respondents also said "It depends", mentioning that an excessive fragmentation of responsibilities in multiple MonoBehaviour scripts may result in a waste of effort, code more difficult to be understood, and, above all, it may negatively affect performances.

Coupling objects through the IDE Inspector. This is the most controversial of our smells: 31% of positive responses, 34% negative, and 35% of neutral. While respondents agree that an excessive level of coupling makes the project difficult to maintain,



(a) UnityLinter is useful



(b) I would use it

Figure 4: Survey responses on whether: (a) respondents perceive UnityLinter as useful, and (b) they would use it.

coupling through the inspector has numerous advantages. First, it is better than other solutions (use of `Find` or messaging systems) in terms of performances. Second, it is easy to use even for non-programmers (*e.g.,* graphics experts) that participate in the development. Third, it is useful during testing activities, as it allows developers to attach/detach objects.

Animation speed depends on the frame rate. There is a general agreement (91% of positive answers) in this case. At the same time, respondents indicate that not only `Time.deltaTime` should be used to scale transforms and make them frame rate-independent but, also, to perform transforms inside `FixedUpdate()` instead of `Update()`. While `Update()` is executed once per frame, `FixedUpdate()` execution frequency depends on the frame rate and, for this reason, would not look faster or slower.

Fig. 4 shows, again in the form of diverging stacked bar charts, results related to the perceived usefulness of UnityLinter, and whether the respondent would use it if available as a tool. 64% of the respondents agreed about the usefulness of a UnityLinter, while only 51% indicated that they would use it if available as a tool. We conjectured that responses to these answers could depend on the respondents' development experience (general and, in particular, with Unity). We checked this relationship using permutation tests [16] (non-parametric alternative to the Analysis of Variance), and we found no significant results, *i.e.,* the user perception of UnityLinter's usefulness, and the willingness to adopt it as a tool do not depend on the developers' experience.

Finally, respondents provided some suggestions for further smell detection. These include:

(1) Analysis of naming conventions. Note that, within proper limits, this is available for language like Java (*e.g.,* with CheckStyle) and in general a lot of research on this topic has been carried out [12, 31].
(2) User interface-related smells.
(3) Lack of caching when using `GetComponent()`.
(4) Empty methods in MonoBehaviour classes (generated by the IDE) that cause unnecessary delays.

Interestingly, respondents also pointed out how many of the Unity tutorials contain smells among the ones we suggest and, in general, code respondents believe to be smelly.

While suggestions (1) and (2) are out of scope for this work, we followed up on suggestion (3) accounting for it when implementing the Getting a GameObject reference finding it by name, and (4) by implementing a seventh smell detector, i.e., A MonoBehaviour class contains empty methods.

RQ$_1$ Summary: Developers generally agree about the usefulness of defecting performance-related issues, although they point out that only the excessive use of such practice might produce visible problems. Maintainability issues are less of a concern when the price to pay is reduced performance.

5 UNITY LINTER - ARCHITECTURE AND IMPLEMENTATION

In this section we describe UnityLinter, a static analysis tool able to recognize 7 Unity smells, *i.e.,* the 6 described in Section 3, plus the A MonoBehaviour class contains empty methods.

UnityLinter has been developed as Python scripts, which takes as input a Unity project and, before detecting smells, extracts three pieces of information:

(1) A parse tree of C# files using the *srcML* tool [21].
(2) A call graph using *Doxygen* [30]. *Doxygen* generates documentation starting from the source code. For this purpose, it is able to extract call graphs, inheritance diagrams, and collaboration diagrams.
(3) Data flow information using *srcSlice* [13, 39].

After having extracted the parse tree, call graph, and data flow, we leverage them to identify the smells, using the rules described in the following.

5.1 Detection Rules

This section aims to detail the detection rules defined for each smell.
Allocating and destroying GameObjects in updates.: to detect this smell UnityLinter leverages the call graph extracted by *Doxygen*. Basically, we search for the presence of `Instantiate` and `Destroy` invocations either in `Update()` methods, or in methods transitively called by `Update()`, according to the *Doxygen* call graph.
Getting a GameObject reference finding it by name: the detection of this smell is relatively straight-forward and similar to the previous one, in that we look for the presence of `Find`-related invocations in `Update()` methods, or in methods reachable from `Update()`. Based also on the feedback we received in the survey of Section 4, the invocations we match are `Find`, `FindWithTag`, `FindGameObjectWithTag`, and `GetComponent`.
Coupling objects through the IDE Inspector: we analyze parse trees produced by *srcML*, identifying the presence of `public` or `[SerializeField]` fields of type GameObjects or other non-primitive types. These represent fields where the developer typically drags GameObjects to create static couplings. We discard primitive type fields because these are typically used to set constants and other properties through the Unity Inspector. Then, to check whether a script is, indeed linked to a scene (*i.e.,* in any of its GameObjects), UnityLinter retrieves its identifier (*guid*) in its metadata file generated by Unity. Then, it checks whether the *guid* is in any scene metadata.
Heavyweight Update methods: statically-identifying computationally-intensive methods is particularly challenging, as this task is often performed through a profiler (Unity provides a quite thorough profiling infrastructure). Nevertheless, it can be useful to early warn developers while writing source code. To this aim, we look for symptoms of potential heavyweight

`Update()` methods. More specifically, we look for three different kinds of symptoms: (i) a high degree of nesting in loops; (ii) an excessive number of method calls; and (iii) the presence of Unity API invocations known to be particularly expensive.

For the first two cases, a threshold should be set. While a developer could choose herself how to calibrate such thresholds, the approach we follow to set thresholds in our analyses (and in the study reported in Section 6) is to consider as heavyweight all `Update()` methods having loop level of nesting or number of method calls exceeding the third quartile of all `Update()` methods in the project. In the absence of historical data, one could set these thresholds based on previous experience or data from other projects.

For the third case, we consider expensive APIs documented in Unity-related forums [6]. In the current implementation, the considered APIs are related to the `Camera.main` access, and to messaging, *i.e.,*`SendMessage` and `BroadcastMessage`.

A MonoBehaviour class contains empty methods: we implement this smell by simply checking for the presence of empty Start() and Update() methods in MonoBehaviour classes.

Lack of separation of concerns: this smell could be related to several aspects of software development and, in our context, of Unity development. While one possibility could have been to leverage traditional approaches aimed at computing lack of cohesion (for example, by computing the conceptual cohesion metric [36]), we found that this would not work properly in our case. For example, a method could contain an invocation to an object transform and another one changing the object state or accessing a controller. Therefore, we opted for implementing a very specific case of detector able to identify MonoBehaviour scripts containing both access to `GameObject.transform` and to the `Input` class, *i.e.,* the one responsible for handling input controllers.

Animation speed depends on the frame rate: this smell occurs when the magnitude of a GameObject transform is not scaled using `Time.deltaTime`, *i.e.,* when it does not account for the frame rate. A simple detection could check the presence of `Time.deltaTime` in transform operation parameters. However, the transform could also get other variables defined as a function `Time.deltaTime`. To this aim, we leverage *srcSlice* to check for the presence of def-use chains between variable definitions and transform calls. If a transform call does not contain any `Time.deltaTime` reference, nor (transitively) uses variables defined on `Time.deltaTime`, then we highlight the smell.

5.2 Limitations

The current implementation of UnityLinter suffers of some limitations, that could be addressed in future work:

1. *Limited set of smells covered:* as discussed in Section 3, in this first work on video game smell detection we did not consider all possible smells. This would require a systematic analysis of literature and developers' discussion, and can be subject of future work.
2. *Lightweight and approximate detection of heavyweight methods:* as discussed above, static analysis is not particularly suitable to identify computationally-intensive methods. We mitigated this limitation trying to identify from forums what are (some) expensive Unity APIs, and detect their usage in Update methods. Future work should target an accurate (dynamic) profiling of the Unity framework.
3. *Very specific detection of Lack of separation of concerns:* as explained above, we cope with one specific case of lack of separation of concern (mix up of input controller and transform in the same script). Clearly, this is not the only case of interest, and in future, further cases could be handled.
4. *Imprecise call graph construction and data flow analysis: Doxygen* is not a tool explicitly designed for precise construction of call graphs (it is more a lightweight tool conceived for documentation analysis). Also, *srcSlice* performs a very lightweight data flow analysis. However, to the best of our knowledge, these are the only tools available to analyze C# source code. In general, we believe a lightweight analysis is acceptable for SCATs, which should only provide suggestions of candidate smells to developers.

6 STUDY ON UNITYLINTER ACCURACY AND SMELL DIFFUSENESS

After having collected feedback from potential users of UnityLinter, and having benefited from such feedback to refine the detector, we need to evaluate its accuracy. While it might be acceptable for a linter to perform a lightweight, somewhat imprecise analysis, having too many false negatives or false positives would result in a tool useless for developers. Therefore, we ask our second research question:

> **RQ$_2$:** *How accurate is UnityLinter in detecting the considered bad smells?*

Finally, within the limits of UnityLinter's accuracy, we need to investigate whether the considered smells are actually present in existing games. This is because if a smell is extremely rare (*i.e.,* it rarely happens) then it may not be worthwhile to invest effort on developing approaches for handling it. At the same time, also a smell being extremely frequent might pose questions, *i.e.,* the considered symptom is a widely-adopted development practice, and therefore in such a circumstance the smell detector would produce an excessive number of warnings. Our third research question is:

> **RQ$_3$:** *What is the diffuseness of the considered bad smells?*

6.1 Study Methodology

We apply UnityLinter on a set of projects, to (i) validate its accuracy, and (ii) study the magnitude of the investigated smells. To this purpose, we considered 100 Unity game projects. We have downloaded them from GitHub choosing only the projects which have the sentence "Game in Unity" in their description. We have selected the first 100 largest ones (considering the overall repository size).

To address **RQ$_2$**, we perform a manual evaluation of the detected smells. As for the precision, we extracted a stratified random sample of 359 smells among the 5,461 detected one, where strata are computed based on the proportions of different smell types. This sample ensures a ±5% margin of error with a confidence level of 95%. Since one smell (Getting a GameObject reference finding it by name) has a fairly limited number of instances (61), hence reaching a number of 420 smells to be manually analyzed in total.

Table 2: Performance evaluation: Precision.

Smells	Analyzed	TP	FP	Prec.
Allocating and destroying GameObjects in updates	11	11	0	100%
Coupling objects through the IDE Inspector	258	246	12	95%
Heavyweight Update methods	28	14	4	86%
Getting a GameObject reference finding it by name	61	61	0	100%
A MonoBehaviour class contains empty methods	25	25	0	100%
Lack of separation of concerns	26	26	0	100%
Animation speed depends on the frame rate	11	10	1	91%

Table 3: Performance evaluation: Recall.

Smells	Analyzed	TP	FN	Rec.
Allocating and destroying GameObjects in updates	11	7	4	64%
Coupling objects through the IDE Inspector	421	266	155	63%
Heavyweight Update methods	27	19	8	70%
Getting a GameObject reference finding it by name	5	4	1	80%
A MonoBehaviour class contains empty methods	12	12	0	100%
Lack of separation of concerns	28	14	14	50%
Animation speed depends on the frame rate	21	8	13	38%

Table 4: Diffuseness of the detected smells.

Smells	# Total Instances	# Affected Projects	Min	Med	Max
Allocating and destroying GameObjects in updates	167	51	0	1	15
Coupling objects through the IDE Inspector	3910	97	0	25	352
Heavyweight Update methods	412	86	0	3	30
Getting a GameObject reference finding it by name	61	39	0	0	8
A MonoBehaviour class contains empty methods	366	81	0	3	52
Lack of separation of concerns	385	66	0	3	20
Animation speed depends on the frame rate	161	57	0	1	12

The analysis has been performed by two authors, knowledgeable of Unity, that were not involved in the development of UnityLinter (to avoid bias). During a first phase, a set of 20 smell instances (of different types) were jointly coded, and then they completed the task independently, with an agreement of 0.94 and a Cohen's k inter-rater agreement [20] of 0.45, which is a moderate agreement (note that such an agreement is not particularly high because if the very high percentage, 92% of cases where both raters agreed with a *true positive* case). Finally, They discussed and resolved inconsistent assessments.

Assessing the recall on all 100 projects was not feasible. Instead, the two assessors manually inspected five complete projects (randomly chosen among the 100) to identify possible smells, reaching a moderate Cohen's k inter-rater agreement (0.48). Finally, they resolved inconsistent evaluations, and compared their results with those of UnityLinter. The analyzed GitHub repositories are: *ArtemSobolevPI-53/3D-Racing-Game-in-Unity*, *pchen4South/unity-jam*, *smcguire56/GestureBasedUIProject*, *zanval/MiniLD62*, and *B-e-n-j-a-m-i-n-S-a-v-a-g-e_Simple-VR-Game-in-Unity*.

As outcome of the validation, we report, for each smell type, the achieved precision and recall. Also, we discuss the reasons for false positives and negatives.

To address **RQ$_3$** we report, for each smell type: (i) the number of detected instances across all the studied projects, (ii) the percentage of affected projects and (iii) the percentage of affected source code files. We also discuss some exemplar cases of detected smells.

6.2 RQ$_2$ results: UnityLinter accuracy

Table 2 reports the precision, for different smell types, achieved on the manually analyzed instances. As the table shows, the precision ranges between 86% of Heavyweight Update methods and 100% reached by different smell types, *i.e.,* Allocating and destroying GameObjects in updates, Getting a GameObject reference finding it by name, A MonoBehaviour class contains empty methods and Lack of separation of concerns. The different levels of precision can be explained by how UnityLinter identifies the different bad smell types.

The main source of imprecision for Animation speed depends on the frame rate is due to the data flow analysis of *srcSlice*. As explained in Section 5.1, the smell detection is based on presence of a def-use chain between a variable assigned to `Time.deltaTime` and a `transform` operation. As for the Heavyweight Update methods, as explained in Section 5.2, UnityLinter is only able to provide a very rough approximation. It turns out that several cases in which a method performs a large number of invocations do not represent a serious concern, because the invoked methods are relatively inexpensive to execute. Finally, the false positives for Coupling objects through the IDE Inspector are related to MonoBehaviour scripts which contain public attributes, which however are not linked to a GameObject through the IDE, but dynamically, using a `GetComponent` method.

As for the recall, results from the five manually-inspected projects are shown in Table 3. The recall ranges between 50% of Lack of separation of concerns and 100% of A MonoBehaviour class contains empty methods. Lack of separation of concerns mostly misses cases of GameObject transformations not contemplated in our rules, which, in the future, need to be extended because of a plethora of available GameObject transformation APIs available in Unity. A MonoBehaviour class contains empty methods is very straight-forward to detect, and this explains the 100% recall.

For Animation speed depends on the frame rate, false negatives are due to limitations in the data flow analysis performed by *srcSlice*, which is currently the only available data flow analyzer for C#. To improve UnityLinter, we may need to develop, in future, our own data flow analyzer.

For Allocating and destroying GameObjects in updates and Getting a GameObject reference finding it by name, false negatives are due to *Doxygen*, which partially reconstructed the call graph chains. Also, in this case, only a better call graph analyzer could improve the recall of UnityLinter.

As for Coupling objects through the IDE Inspector, false negatives are related to public field declarations that the C# parser could not properly identify.

RQ$_2$ Summary: UnityLinter precision ranges between 86% and 100%, whereas recall between 50% and 100%. Sources of imprecision and limited recall are due to the approximate data flow analysis, and other heuristics of a lightweight analysis.

6.3 RQ$_3$ results: Smells' diffuseness.

Table 4 reports the diffuseness of the studied smell types across the 100 analyzed projects. Clearly, this data must be interpreted keeping into account the precision and recall of UnityLinter, discussed in RQ$_2$.

As the table shows, the largest number of smell instances (and the largest proportion of affected projects) occur for Coupling objects through the IDE Inspector: 97% of the analyzed projects are affected by this smell. This is unsurprising, based on the results of RQ$_1$. Indeed, many developers consider this practice, *i.e.,* creating coupling through the inspector, as a useful (and in general harmless) development practice.

86% of the analyzed projects are affected by Heavyweight Update methods. Truly, as we explained in Section 5, a linter like UnityLinter (but the same can be said of other tools such as FindBugs [5] or Android Lint [2]) could only highlight potential problems in such cases. Only a dynamic analysis and profiling may reveal whether or not these represent actual cases of performance problems.

About Lack of separation of concerns, despite UnityLinter has a fairly limited recall, we still found 366 cases (and 66% of the projects) in which developers mix up direct access to input controllers with other operations such as access to transforms. Fortunately, Unity has recently released a new input management mechanism [9] which would help to make the source code cleaner.

The least common smell is Getting a GameObject reference finding it by name: it affects 39% of the analyzed projects. On the one hand, as per RQ$_1$ results, this smell is known and well-perceived by developers. At the same time, the high usage of coupling through the Inspector reduces the use of `Find` operations. Other smells with a relatively low perception are Allocating and destroying GameObjects in updates (51% of the analyzed projects affected). As also indicated in feedback obtained in the RQ$_1$ survey, when objects need to be created and released with a high frequency, developers use an object pool. As per Animation speed depends on the frame rate (57% of projects affected) developers properly scale transforms using `Time.deltaTime`, or use `FixedUpdate()`.

Finally, only 66% of the projects are affected by A MonoBehaviour class contains empty methods. As previously explained, this smell occurs when developers do not delete empty method templates automatically generated by the IDE.

RQ$_3$ Summary: The studied smell types affect a proportion of projects ranging between 39% and 97%. While Coupling objects through the IDE Inspector is highly diffused (but also considered as an acceptable development practice), smells such as Getting a GameObject reference finding it by name, Allocating and destroying GameObjects in updates, and Animation speed depends on the frame rate are more acknowledged by developers and also occur in fewer projects.

7 THREATS TO VALIDITY

Threats to *construct validity* concern the relationship between theory and observation. Such threats may affect RQ$_1$ because we asked developers to provide their perceived importance of smells given a short description of the problem. To mitigate this threat, we provided some explanatory examples. Also, no respondent indicated issues about possible misunderstanding in the questions. Another threat can be related to the assessment of heavyweight `Update()` methods, because, as explained in Section 5, UnityLinter provides a very lightweight estimate. Finally, a threat could be due to mistakes in the implementation of UnityLinter. While we could not exclude such mistakes, we carefully tested it on several video game examples before conducting the study.

Threats to *internal validity* concern factors internal to our study that could influence our results. Primarily, such threats can be due to the manual assessment of precision and recall, which could suffer from subjectiveness and incompleteness. We mitigated the former by conducting an initial validation phase with multiple annotators analyzing smell instances in pairs and discussing them. The incompleteness was mitigated by letting two independent annotators assessing the recall, and then comparing and discussing their results. Also, as explained in Section 5.2, UnityLinter suffers from the imprecision of *srcSlice* and *Doxygen*.

Threats to *external validity* concern the generalization of our findings. First, the set of detected smells is surely incomplete. A specific study is needed to thoroughly investigate a broad set of smells that can affect video games. Our work, instead, represents a first attempt to define some smells and provide detectors for them. Second, some smells (*e.g.,* Lack of separation of concerns) have been defined for very specific cases and can be extended in future work. Finally, our study involved the analysis of open source games. It is of paramount importance to extend the study to further video games (including commercial ones) that can be more representative of the market.

8 RELATED WORK

This section discusses related work about (i) design principles in video games, and (ii) empirical studies on video game development.

8.1 Design principles in video game development

Different authors have studied how to apply design principles in the context of video game development [14, 17, 23, 29, 38, 40, 44], either by defining specific patterns, or by applying GoF design patterns (DPs) [26] in the context of video game development.

Nystrom [40] proposes a revisited version of some GoF DPs (namely command, flyweight, observer, prototype, singleton, and state), and a set of specific DPs for the video game domain. In particular, Nystrom defines thirteen design patterns grouped into four categories: sequencing patterns (related to time issues), behavioral patterns (to define and refine several behaviors in a way they are easy to maintain), decoupling patterns, and optimization patterns (to speed up a game).

The bad smells we have defined and identified are related to the Nystrom's DPs. For instance, *Allocating and destroying GameObjects in updates* is related to Nystrom's optimization patterns, while *Heavyweight Update methods* and *Animation speed depends on the frame rate* are related to both sequencing and optimization categories. The *Getting a GameObject reference finding it by name* is related to optimization and decoupling patterns, while *Coupling objects through the IDE Inspector* is related to the decoupling patterns, and the *Lack of separation of concerns* related to behavioral patterns.

The book by Murray [38] also proposes various kinds of design solutions for Unity video games, including the use of object pools or virtual controllers.

Barakat *et al.* [17] propose to integrate creational and behavioral DPs (namely state, strategy, prototype, and observer) with a specific game design framework to provide the developers with some hints on what DP to use with the main game aspects. Such an integration would make easier reuse and maintenance tasks.

Ampatzoglou *et al.* [15] studied the correlation between design pattern application, software defects, and debugging rate in 97 Java open source games. Even if the overall number of design pattern instances does not correlate with defect frequency and debugging effectiveness, some specific design patterns appeared to have a significant impact on the number of reported bugs and debugging rate.

Ampatzoglou *et al.* [14] studied the impact of DPs on the maintainability of two open-source games. Their result indicated that DPs help to increase maintainability, while increasing the project size. Also, Ampatzoglou *et al.* [29] leveraged DPs to implement game rules and logic. They show that the use of DPs in this context helps to avoid introducing undesired complexity, and to increase reusability, maintainability and flexibility. Figueiredo *et al.* [23] conducted an experiment that showed the positive impact of GoF patterns in video game development. Other work reported design experiences in the context of video game development. Qu *et al.* [44] reported an experience of application of DPs to solve a number of problems in video game development, including sprite and map management, or handling the game state.

While several papers and books can be found in the literature about best practices to use in gaming development to improve their quality, no work is, at the best of authors' knowledge, about bad smells in video games, *i.e.,* how and how much suggested best practices are not applied in game development. The only tool related to our work is Rider [8], a commercial IDE for Unity development which highlights the use of expensive Unity APIs. Our catalog goes beyond that because it features different kinds of smells.

8.2 Empirical studies on video game development

Some studies have focused on the video game development process. Stacy and Nandhakumar [46] conducted an interview-based study, and found that developers perceive video game development different from traditional software development. Murphy-Hill *et al.* [37] studied, through a combination of interviews and surveys, the video game development process in Microsoft. They highlighted substantial differences with the development of traditional applications, including less requirement engineering and design, fairly limited reuse, and a completely different approach to testing. Pascarella *et al.* [42] conducted a similar study (from a quantitative point of view) in the open-source, analyzing the typical changes occurring in video games, the fault distribution, and the programmers' perception of the video game development process.

Petrillo *et al.* [43] studied the typical problems that occur during video game development. These are mainly related to the planning and include unrealistic features, optimistic schedules, and exceeding the planned budget. Other studies have investigated the types of bugs that occur in video games. In particular, Lewis *et al.* [32] has proposed a hierarchical taxonomy of faults occurring in video game projects.

Our work looks video game development from a different perspective (*i.e.,* product vs. process) than the previous work. While the aforementioned studies are related to understanding the nature of bugs in video game development, our work (and the RQ_1 survey in particular) indicates that, besides traditional bugs, performance issues and to some extent maintainability issues are particularly relevant in video games development. In this scenario, an early discovery of problems through specialized linters can be therefore beneficial.

Other studies have focused on mining video game data to gather information about the development process. Lin *et al.* have studied the evolution of video games by mining data from the Steam platform [33–35]. In particular, they investigated the reviewing mechanism in games [35], where the number of played hours is considered as an important factor to determine the usefulness of a review. Another mechanism for collecting feedback is represented by early-access games [34]. Finally, they studied the presence of "zero-day" updates in video games [33] correlating them with the overall update frequency. Work on static analysis and mining runtime data are largely complementary to achieve an effective and efficient video game development process.

9 CONCLUSION

This paper described UnityLinter, a static analyzer capable of identifying 7 types of bad smells in Unity projects. Such smells cover various aspects of video game development, including performance issues (*e.g.,* use of `Instantiate` and `Destroy` in `Update()` methods without relying on object pools), maintainability issues (*e.g.,* mix-up of input handling and GameObject transforms in the same script), and behavioral issues (inappropriate scaling of transforms with respect to frame rate).

We first assessed the relevance of the smells through a survey with 68 participants. Results indicate that performance smells are generally more perceived than maintainability smells, and some smells such as Coupling objects through the IDE Inspector are considered as an acceptable practice. Then, we assessed the UnityLinter precision and recall, which, on the analyzed samples, vary in the range [89%-100%] and [50%-100%] respectively. Finally, we reported and discussed the diffuseness of Unity smells on 100 open source video games, showing how the studied smells practice affect a relatively large proportion of the analyzed projects, *i.e.,* between 39% and 97%.

There are several directions for future work, mostly focused towards improving approach, also based on the collected information from our first evaluation. First, we plan to use better code analyzers or heuristics to improve UnityLinter accuracy. Indeed, many sources of imprecision and limited recall are due to the use of lightweight data flow and call graph analysis.

Second, we plan to detect a large variety of smells, and to better generalize some of the already detected smells, such as Lack of separation of concerns, but also refine others such as Heavyweight Update methods, for which we plan to perform an empirical analysis of the performance of a broad set of Unity APIs. Finally, we plan to conduct an in-field evaluation, to determine the extent to which the use of UnityLinter would actually help developers.

REFERENCES

[1] [n. d.]. A Ruby static code analyzer and formatter. https://github.com/rubocop-hq/rubocop (Last access: 01/01/2020). ([n. d.]).

[2] [n. d.]. Android Lint. https://developer.android.com/studio/write/lint (Last access: 01/01/2020). ([n. d.]).

[3] [n. d.]. Blender. http://unity3d.com/unity/ (Last access: 01/01/2020). ([n. d.]).

[4] [n. d.]. CheckStyle. http://checkstyle.sourceforge.net/ (Last access: 01/01/2020). ([n. d.]).

[5] [n. d.]. FindBugs. http://findbugs.sourceforge.net/ (Last access: 01/01/2020). ([n. d.]).

[6] [n. d.]. Performance recommendations for Unity https://docs.microsoft.com/en-us/windows/mixed-reality/performance-recommendations-for-unity (Last access: 01/01/2020) . ([n. d.]).

[7] [n. d.]. PMD. https://pmd.github.io/ (Last access: 01/01/2020). ([n. d.]).

[8] [n. d.]. Rider for Unity. https://www.jetbrains.com/dotnet/promo/unity/ (Last access: 01/01/2020) . ([n. d.]).

[9] [n. d.]. Unity - Introducing the new Input System. https://blogs.unity3d.com/2019/10/14/introducing-the-new-input-system/ (Last access: 01/01/2020) . ([n. d.]).

[10] [n. d.]. Unity. http://unity3d.com/unity/ (Last access: 01/01/2020). ([n. d.]).

[11] [n. d.]. Unreal Engine. https://www.unrealengine.com/en-US/ (Last access: 01/01/2020). ([n. d.]).

[12] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 38–49.

[13] Hakam W Alomari, Michael L Collard, Jonathan I Maletic, Nouh Alhindawi, and Omar Meqdadi. 2014. srcSlice: very efficient and scalable forward static slicing. *Journal of Software: Evolution and Process* 26, 11 (2014), 931–961.

[14] Apostolos Ampatzoglou and Alexander Chatzigeorgiou. 2007. Evaluation of object-oriented design patterns in game development*Department of Applied Informatics. *Information and Software Technology, vol. 49, pp, 445ñ454* (2007).

[15] A. Ampatzoglou, A. Kritikos, E. M. Arvanitou, A. Gortzis, F. Chatziasimidis, and I. Stamelos. 2011. An Empirical Investigation on the Impact of Design Pattern Application on Computer Game Defects. In *Proceedings of 15th International Academic MindTrek Conference: Envisioning Future Media Environments, MindTrek 2011, Tampere, Finland, 2011)*.

[16] Rose D. Baker. 1995. Modern permutation test software. In *Randomization Tests*. Marcel Decker.

[17] Nahla H. Barakat. 2019. A Framework for integrating software design patterns with game design framework. In *Proceedings the 2019 8th International Conference on Software and Information Engineering, ICSIE 2019, Cairo, Egypt, April 09 - 12, 2019*.

[18] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 470–481.

[19] Antonio Borrelli, Vittoria Nardone, Giuseppe Di Lucca, Gerardo Canfora, and Massimiliano Di Penta. 2020. Detecting Video Game-Specific Bad Smells in Unity Projects - Dataset https://tinyurl.com/UnitySmells https://tinyurl.com/UnitySmells,. (2020).

[20] J Cohen. 1960. A coefficient of agreement for nominal scales. *Educ Psychol Meas.* (1960).

[21] Michael L. Collard, Huzefa H. Kagdi, and Jonathan I. Maletic. 2003. An XML-Based Lightweight C++ Fact Extractor. In *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*. 134–143.

[22] David E. Evans. 1996. Static Detection of Dynamic Memory Errors. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*. 44–53. https://doi.org/10.1145/231379.231389

[23] Roberto Tenorio Figueiredo and Geber Lisboa Ramalho. 2016. GOF design patterns applied to the Development of Digital Games. In *Proceedings of SBGames 2015, November 11th - 13th, 2015, Teresina, Brazil*.

[24] Forbes. [n. d.]. The Business Of Video Games: Market Share For Gaming Platforms in 2019. https://www.forbes.com/sites/kevinanderton/2019/06/26/the-business-of-video-games-market-share- for-gaming-platforms-in-2019-infographic/#66ce258b7b25 (Last access: 01/01/2020). ([n. d.]).

[25] GameIndustry.biz. [n. d.]. Global games market value rising to $134.9bn in 2018. https://www.gamesindustry.biz/articles/2018-12-18-global-games-market-value-rose-to-usd134-9bn-in-2018 (Last access: 01/01/2020). ([n. d.]).

[26] E. Gamma, R. Helm, R.Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley.

[27] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Software Engineering (ICSE), 2013 35th International Conference on.* IEEE.

[28] S. C. Johnson. 1978. Lint, a C Program Checker. In *Comp. Sci. Tech. Rep.* 78–1273.

[29] Xeni-Christina Kounoukla, Apostolos Ampatzoglou, and Konstantinos Anagnostopoulos. 2016. Implementing Game Mechanics with GoF Design Patterns. In *Proceedings of the 20th Pan-Hellenic Conference on Informatics, PCI-16, Patras, Greece, Nov. 10-12, 2016, ACM, New York, NY, USA*.

[30] Robert S Laramee. 2011. *Bob's Concise Introduction to Doxygen.* Technical Report. Technical report, The Visual and Interactive Computing Group, Computer.

[31] Surafel Lemma Abebe and Paolo Tonella. 2013. Automated Identifier Completion and Replacement. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*. 263–272.

[32] Chris Lewis, Jim Whitehead, and Noah Wardrip-Fruin. 2010. What went wrong: a taxonomy of video game bugs. In *Proceedings of the fifth international conference on the foundations of digital games.* ACM, 108–115.

[33] Dayi Lin, Cor-Paul Bezemer, and Ahmed E. Hassan. 2017. Studying the urgent updates of popular games on the Steam platform. *Empirical Software Engineering* 22, 4 (2017), 2095–2126.

[34] Dayi Lin, Cor-Paul Bezemer, and Ahmed E. Hassan. 2018. An empirical study of early access games on the Steam platform. *Empirical Software Engineering* 23, 2 (2018), 771–799.

[35] Dayi Lin, Cor-Paul Bezemer, Ying Zou, and Ahmed E. Hassan. 2019. An empirical study of game reviews on the Steam platform. *Empirical Software Engineering* 24, 1 (2019), 170–207.

[36] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. 2008. Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems. *IEEE Trans. Software Eng.* 34, 2 (2008), 287–300.

[37] Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. 2014. Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?. In *Proceedings of the 36th International Conference on Software Engineering.* ACM, 1–11.

[38] Jeff W. Murray. 2014. *C# Game Programming Cookbook for Unity 3D.* CRC Press, New York.

[39] Christian D Newman, Tessandra Sage, Michael L Collard, Hakam W Alomari, and Jonathan I Maletic. 2016. srcSlice: a tool for efficient static forward slicing. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 621–624.

[40] R Nystrom. 2014. *Game Programming Patterns* (1st edition ed.). Lightning Source Inc.

[41] A. N. Oppenheim. 1992. *Questionnaire Design, Interviewing and Attitude Measurement.* Pinter Publishers.

[42] Luca Pascarella, Fabio Palomba, Massimiliano Di Penta, and Alberto Bacchelli. 2018. How is video game development different from software development in open source?. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. 392–402.

[43] Fábio Petrillo, Marcelo Soares Pimenta, Francisco M. Trindade, and Carlos Dietrich. 2009. What went wrong? A survey of problems in game development. *Computers in Entertainment* 7, 1 (2009), 13:1–13:22.

[44] Junfeng Qu, Yinglei Song, and Yong Wei. 2013. Applying Design Patterns in Game Programming. In *Proceedings of The International Conference on Software Engineering Research and Practice 2013, (SERP2013), CSREA Press.*

[45] Jaime Spacco, David Hovemeyer, and William Pugh. 2006. Tracking defect warnings across versions. In *Proceedings of the 2006 international workshop on Mining software repositories.* ACM, 133–136.

[46] Patrick Stacey and Joe Nandhakumar. 2009. A temporal perspective of the computer game development process. *Information Systems Journal* 19, 5 (2009), 479–497.

[47] Mario Linares Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2014. Mining energy-greedy API usage patterns in Android apps: an empirical study. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India.* ACM, 2–11.

[48] Fadi Wedyan, Dalal Alrmuny, and James M Bieman. 2009. The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In *2009 International Conference on Software Testing Verification and Validation.* IEEE, 141–150.

[49] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*. 334–344.

[50] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P Hudepohl, and Mladen A Vouk. 2006. On the value of static analysis for fault detection in software. *IEEE transactions on software engineering* 32, 4 (2006), 240–253.