UNIVERSIDADE FEDERAL DE MINAS GERAIS DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

# Mineração de Testes Automatizados em Repositórios de Jogos

Pesquisa Tecnológica

Aluno: Leandro Diniz Silva
Orientador: André Cavalcante Hora

Belo Horizonte, 2024

**Resumo**

Dentro da engenharia de *software* existem diversas técnicas e padrões que podem ser implementados em um projeto para melhorar sua qualidade e robustez, como testes de *software* automatizados. Mas este recurso não é amplamente utilizado em todas os tipos de programas e o desenvolvimento de jogos é exatamente um deles. Essa falta de aderência é causada por diversos motivos, como a complexidade das interfaces dos jogos e a aleatoriedade presente nas interações do usuário. Este trabalho investiga essas questões utilizando técnicas de mineração de repositórios, analisando o código fonte de diversos projetos de jogos disponibilizados em plataformas de código aberto. A pesquisa busca identificar as abordagens utilizadas, com o objetivo de averiguar como elas são utilizadas e qual o nível de adesão encontrados em jogos *open-source*.

# 1 Introdução

As ferramentas de teste de *software* automatizadas têm aplicabilidade em uma ampla gama de sistemas digitais, contando com suporte de uma variedade de bibliotecas e ferramentas que auxiliam na produção e visualização das verificações feitas no programa. Jogos eletrônicos, mesmo que produzidos em diversas dimensões, além da sua linguagem de programação, também são *softwares* e podem conter técnicas de testes automatizados.

De acordo com Arora, a indústria de jogos tem um alcance global sem precedentes, tendo um valor aproximado de $184.4 bilhões de dólares [1]. E mesmo uma industria desse tamanho entrega produtos em um estado que aparenta estar inacabado e repleto de *bugs* indesejados [7].

Em meio a tal contexto, o presente projeto se propõe em realizar uma investigação no mundo de *game development* e tentar encontrar quais técnicas, bibliotecas, *frameworks* e ferramentas são utilizadas na arquitetura dos testes desses sistemas e com isso, também identificar qual o nível de adesão dessas técnicas. Realizar a mineração dessas informações encontradas no imenso universo de repositórios *open-source* do *GitHub* pode nos ajudar a entender como os contribuintes dos sistemas aplicam práticas de testes automatizados no desenvolvimento de jogos.

O restante do trabalho está organizado como segue. Na Seção 2 é apresentado o Referencial Teórico. Na Seção 3 é descrita a Metodologia de Desenvolvimento empregada para a realização do trabalho. Na Seção 4 é descrito os Resultados encontrados e na última Seção 5 são feitas as Considerações Finais.

# 2 Referencial

Para se desenvolver esse projeto foi necessário estudar e pesquisar por trabalhos relacionadas a esse tema, com o objetivo de entender como os desenvolvedores de jogos tratam o seu ecossistema de desenvolvimento. Este referencial teórico busca explorar as práticas adotadas na criação de testes automatizados em jogos,

investigando metodologias e ferramentas e como elas são diferentes de *softwares* de serviço.

## 2.1 Diferenças no Desenvolvimento de Jogos

De acordo com Whittaker [11] o principal objetivo de um testador é simular as interações entre o usuário e o programa, no entanto, em jogos essas interações não são tão diretas quanto em um *software* comum, pois um jogo tem como principal objetivo entreter e divertir seu consumidor [5][8] e não necessariamente entregar um serviço. Isso cria desafios únicos para o desenvolvimento de testes automatizados.

No mundo de desenvolvimento de jogos, a maior parte dos testes são feitos por um time de profissionais de controle de qualidade [8], também chamado de *QA*, esses especialistas acessam o software por um tipo de *black box* e simulam as interações dos usuários para tentar encontrar problemas. Esse tipo de testes tem suas vantagens e é considerado muito importante [3], como conseguir testar o "fator diversão"de um jogo. Mas existem desvantagens como o tipo de trabalho repetitivo para os funcionários e a probabilidade de erros humanos [4].

Uma outra particularidade encontrada nos *video games* é o alto nível de aleatoriedade [6][7]. Esta aleatoriedade, que é essencial para certas características dos jogos, pode ser causada por diversos fatores, como o uso de *multithreading*, a inteligência artificial dos personagens e a aleatoriedade incorporada em certas ações do jogo [5].

Existe uma dinamicidade inerente nas interfaces de jogos, além do seu grande nível acoplação com o código [6], o que cria diversos obstáculos ao se tentar automatizar testes de integração. Alguns desenvolvedores tentam usar técnicas de *template matching*, mas pela constante mudança de telas e a quantidade de detalhes pode-se gerar testes não determinísticos, ou seja testes *flakys* [4].

## 2.2 Testes Automatizados

Para facilitar a criação de testes automatizados, alguns pesquisadores criaram suas próprias ferramentas especializadas. Um desses recursos encontrados [2] utiliza de uma máquina de estados que divide o sistema em duas partes, em domínio, todas as interações da tela que não acontecem por interação dos jogadores, como o inicio do jogo ou a introdução de um novo obstáculo na tela, e em avatar, onde são encontradas todas as modificações que acontecem no mundo causadas por ações de um *player*, por exemplo o personagem controlado pular e destruir algum bloco.

Outro meio de testar é utilizar de ferramentas já existentes nas *engines* de criação de jogos. A Unity oferece um *framework* de testes chamado *Unity Test Frameework* (UTF) e para facilitar a automação dos testes consegue dividir os mesmos em dois tipos *Edit Mode* e *Play Mode* [6]. O primeiro modo são testes que podem ser realizados sem a necessidade de iniciar o jogo, e o segundo somente quando o *software* estiver aberto.

Criar e utilizar essas ferramentas pode ser considerado como um custo de desenvolvimento, mas esses custos normalmente são negligenciados pois os programadores de jogos sabem que existe uma grande chance de seu código ser jogado fora caso alguma mudança no design aconteça [5]. Além disso, existe o problema da duplicação de código, onde pode existir diversas funções que tem o mesmo objetivo mas que são escritas com um nível de otimização diferente para cada caso, com o objetivo de melhorar a performance do jogo.

# 3 Metodologia

A principal abordagem utilizada nessa pesquisa foi a mineração de softwares, onde esses foram exclusivamente jogos eletrônicos *open-source*, utilizando ferramentas de mineração de repositórios para identificar técnicas e padrões aplicados de testes automatizados em desenvolvimento de jogos.

## 3.1 Ferramentas

Para realizar a mineração de repositórios foi utilizado a linguagem de programação *Python*, ela foi escolhida porque nela existe o framework *Pydriller*, que dispõem de ferramentas que facilitam a mineração de repositórios *GitHub*, ajudando a extrair informações como código fonte, *commits*, desenvolvedores e arquivos modificados [9].

Além dessa ferramenta também foi utilizado da *API* do *GitHub* para extrair algumas informações dos repositórios que não eram retornadas em outras ferramentas como as linguagens utilizadas, *tags* e tópicos definidos pelos colaboradores do software.

## 3.2 Atividades

Em prol de se alcançar o objetivo do projeto foi iniciada uma busca de repositórios válidos, esses foram encontrados utilizando das listas *Awesome* do *GitHub*. Essas listas são um aglomerado de repositórios de código aberto dos mais diversos tipos de softwares, e são organizadas e mantidas com bastante zelo pelas suas comunidades.

Com uma lista inicial de cento e trinta e quatro repositórios, possivelmente válidos da *Awesome Open Source Games*, foi feita uma identificação utilizando a *API* do *GitHub* para se encontrar quais linguagens esses jogos foram escritos, pois cada um pode utilizar de um *stack* diferente de tecnologias no seu desenvolvimento e os *frameworks* de testes automatizados estão fortemente ligados a linguagem de código do *software*.

Tendo descoberto as linguagens utilizadas, foi feita uma busca pelos *frameworks* de testes mais utilizados e quais são os seus padrões de nomenclaturas de arquivos, esses resultados podem ser vistos na tabela 1. Foi necessário realizar esse passo para encontrar quais repositórios não continham nenhum arquivo de teste e quais repositórios tinham a possibilidade de ter testes automatizados.

| Linguagem | Nomenclatura de Testes |
|---|---|
| C# | `*Test*` |
| C | `*test*` |
| Java | `Test*`, `*Test`, `*Test*` |
| JavaScript | `*.test.js`, `*.spec.js` |
| PHP | `*Test.php` |
| Python | `test_*` |
| Ruby | `*_test.rb`, `*_spec.rb` |
| TypeScript | `*.test.ts`, `*.spec.ts` |
| Scala | `*Spec*`, `*Test*` |

Tabela 1: Nomenclatura de Testes por Linguagem.

Após essa filtragem a lista de repositórios, que inicialmente continha cento e trinta e quatro, foi reduzida a quarenta e oito candidatos. Isso aconteceu devido a maioria dos jogos, oitenta e dois deles, não conter nenhum arquivo de teste em todo seu código fonte.

Outra causa secundária da redução da quantidade total, foi a inexistência dos repositórios citados nas lista *Awesome*, pois mesmo que elas sejam cuidadosamente curadas projetos podem ser apagados do *GitHub* e não removidos da lista até as suas próximas atualizações.

O último passo realizado foi a análise do código fonte, de todos os repositórios que poderiam conter testes automatizados, buscando encontrar a quantidade de testes unitários, integração e de sistema além do nível que a cobertura de código esses procedimentos alcançavam no *software*. Os resultados simplificados dessa exploração dos dados podem ser vistos na tabela 2.

| Repositório | Unitário | Integração | Sistema | Cobertura |
|---|---|---|---|---|
| kenrick95/c4 | 0 | 2 | 0 | N/D |
| David20321/FTJ | 0 | 20 | 0 | N/D |
| lichess-org/lila | 700 | 0 | 0 | N/D |
| basicallydan/skifree.js | 27 | 0 | 0 | N/D |
| mitallast/diablo-js | 0 | 1 | 0 | N/D |
| CamHenlin/Roguish | 11 | 0 | 0 | N/D |
| antionio/game-off-2013 | 0 | 5 | 0 | N/D |
| freeciv/freeciv-web | 0 | 11 | 4 | N/D |
| CodeArtemis/TriggerRally | 87 | 8 | 0 | N/D |
| cheshire137/blicblock-js | 54 | 17 | 0 | N/D |
| sharkdp/cube-composer | 22 | 0 | 0 | N/D |
| dart-lang/pop-pop-win | 17 | 0 | 0 | N/D |
| munificent/hauberk | 6 | 0 | 0 | N/D |
| KrofDrakula/squirts | 3 | 0 | 0 | N/D |

Tabela 2: Testes e Coberturas por Projeto

# 4 Resultados

Dentro dos quarenta e oito projetos, que foram filtrados pela existência de arquivos de testes, somente quatorze deles, exibidos na tabela 2, realmente continham algum tipo de teste automatizado. Isso aconteceu pois a maioria dos outros repositórios tinham arquivos de testes manuais, que se encaixavam na nomenclatura dos seus *frameworks*, mas nenhum automatizado.

Um ponto que fica aparente dentre esses jogos é a falta do uso da cobertura de códigos, onde todos os códigos verificados não continham nenhum arquivo com registro dessa informação, como mostrado na tabela 2 pela referência N/D da coluna cobertura, mesmo os *softwares* que fizeram toda uma engenharia de testes automatizados não utilizaram desta técnica, que poderia ser implementada, em certos casos, diretamente com as bibliotecas já utilizadas.

## 4.1 *Frameworks* de Teste

Em relação aos *frameworks* utilizados, pode ser observado na tabela 3 que cada *software* fez uso somente de uma dessas ferramentas, com exceção do **lichess-org/lila** que utilizou do MUnit para a parte dos seus códigos escrito em Scala e o Vitest nos seus arquivos Javascript.

Um caso especial desses jogos foi o do **David20321/FTJ**, pois seu código foi escrito com uso da *engine* Unity, então o *framework* de teste utilizado foi o Unity Test Framework (UTF). O UTF do NUnit, que é uma biblioteca utilizada para criar testes unitários em ambientes que utilizam da linguagem C#.

| Nome | Repositório | Estrelas | Framework | Arquivos de teste |
|---|---|---|---|---|
| Connect Four | kenrick95/c4 | 240 | Vitest | 3 |
| FTJ | David20321/FTJ | 313 | UTF | 10 |
| lichess | lichess-org/lila | 14755 | MUnit, Vitest | 153 |
| SkiFree.js | basicallydan/skifree.js | 507 | Mocha | 2 |
| Diablo.js | mitallast/diablo-js | 936 | JUnit | 2 |
| Roguish | CamHenlin/Roguish | 88 | QUnit | 3 |
| Room for Change | antionio/game-off-2013 | 136 | JUnit | 2 |
| Freeciv-web | freeciv/freeciv-web | 1969 | unittest | 8 |
| TriggerRally | CodeArtemis/TriggerRally | 316 | Nodeunit | 167 |
| Blicblock | cheshire137/blicblock-js | 16 | RSpec | 4 |
| cube-composer | sharkdp/cube-composer | 1994 | Mocha | 2 |
| Pop, Pop, Win! | dart-lang/pop-pop-win | 147 | Dart testing | 11 |
| hauberk | munificent/hauberk | 1969 | Dart testing | 5 |
| squirts | KrofDrakula/squirts | 29 | Mocha | 1 |

Tabela 3: Sobre os Repositórios.

## 4.2 Testes de Unidade

Dado que os jogos digitais podem ter uma grande quantidade de funções, que vão da matemática resultante de uma interação aos cálculos e restrições necessárias da física de todo o sistema, o número de testes automatizados encontrados foi

pequeno em quase todos os repositórios, com exceção do repositório dois que teve o maior acervo de testes unitários.

Um exemplo de testes de unidade, que verificam uma parte bem definida do sistema, pode ser observado no trecho de código 1. Neste código existe uma *suit* de testes chamadas *"a board"* que apuram diversas regras do tabuleiro de xadrez e como ele deve reagir na maiorias das casos.

Por exemplo no início do jogo todas as peças devem estar distribuídas corretamente, essa regra pode ser verificada na linha 4 do trecho de código 1 e na linha 8 temos a ação de garantir que o usuário não possa fazer jogadas ilegais como tentar movimentar um bloco sem uma peça.

```
1  val board = Board()
2  "a board" should {
3    ...
4    "have pieces by default" in {
5      board.pieces must not beEmpty
6    }
7    ...
8    "not allow an empty position to move" in {
9      board move 'e5 to 'e6 must beFailure
10   }
11   ...
12 }
```

Trecho de Código 1: Testes Unitários Xadrez lichess.

## 4.3 Testes de Integração

Já os testes de integração podem ser encontrados na metade dos repositórios analisados, mas aparecem em pouca quantidade, e de acordo com as mensagens de seus *commits*, testam determinadas interações que geraram algum problema recorrente que foi identificado durante o desenvolvimento. Esses testes não foram desenvolvidos com o objetivo de garantir um nível de confiança da interação de múltiplas partes do sistema, mas sim escritos para verificar se alguma falha foi corrigida.

No repositório **cheshire137/blicblock-js** é possível encontrar uma quantidade pequena de testes de integração, esses podem não testar todas as requisições do sistema com sua *API*, mas são escritos de modo que as suas verificações precisem passar por um servidor para testar uma interação, mesmo que em algumas partes do código *stubs* sejam utilizados para simular essa interação.

O trecho de código 2 é um grupo de testes das requisições *POST*, que estão relacionados ao sistema de pontuação dos usuários e fazem verificações, como se a criação de um *score* esta sendo feita corretamente, se a classe retornada realmente é uma de pontuação e se o *redirect* que é retornado está correto. Em todas as chamadas da *API* é enviado uma sessão de usuário válida, *valid_season* nas linhas 5, 10 e 16, para que o servidor identifique que quem esta usando suas rotas está de acordo com as permissões necessárias do sistema

```
describe "POST create" do
  describe "with valid params" do
    it "creates a new Score" do
      expect {
        post :create, {:score => valid_attributes},
            valid_session
      }.to change(Score, :count).by(1)
    end

    it "assigns a newly created score as @score" do
      post :create, {:score => valid_attributes},
          valid_session
      expect(assigns(:score)).to be_a(Score)
      expect(assigns(:score)).to be_persisted
    end

    it "redirects to the created score" do
      post :create, {:score => valid_attributes},
          valid_session
      expect(response).to redirect_to(Score.last)
    end
  end
  ...
end
```

Trecho de Código 2: Testes de Integração Blicblock.

## 4.4 Testes de Sistema

De todos os projetos lidos, em somente um deles foi encontrado testes de sistemas, e essas verificações foram criadas com o objetivo de observar se algumas telas de menu do jogo estavam sendo geradas corretamente de acordo com as interações do usuário com o sistema. Uma dessas apurações da interface é encontrada no trecho de código 3, onde o jogo é iniciado e as verificações de existência de três abas do menu são realizadas.

```
casper.test.begin('Test that game list is responding.', 4,
    function suite(test) {
    casper.start("http://localhost/game/list", function() {
        test.assertHttpStatus(200);
        test.assertExists('#single-player-tab', 'Test that
            game-list contains expected tabs.');
        test.assertExists('#multi-player-tab', 'Test that
            game-list contains expected tabs.');
        test.assertExists('#play-by-email-tab', 'Test that
            game-list contains expected tabs.');
    });

    casper.run(function() {
        test.done();
```

```
11      });
12  });
```

Trecho de Código 3: Testes de Sistema Freeciv-web

# 5 Conclusão

Jogos digitais são *softwares* tão quanto um aplicativo de escutar música, mas diversos problemas podem ser encontrados ao se aplicar técnicas de testes automatizados, dificuldades estas que não são encontrados frequentemente em sistemas de serviços. Isso é refletido em sistemas de código aberto, na maior parte deles nenhum tipo de estrutura foi encontrada para se realizar essas automações com um alto nível de garantia.

Nos repositórios analisados que continham algum tipo de teste, somente um deles continha um processo bem definido de testes automatizados, onde testes de unidade e integração foram escritos para boa parte do sistema. Mas os outros repositórios encontrados, tinham uma quantidade pequena de testes, e esses poucos testes buscavam testar somente funcionalidades que estavam em um estado que não correspondia as expectativas dos desenvolvedores ou geravam resultados não esperados, funções que continham *"bugs"*.

Nos repositórios que foram encontrados testes, foi possível observar que as ferramentas utilizadas foram os *frameworks* de testes da linguagem em que o código foi escrito, somente um dos quatorze projetos utilizou de uma ferramenta não diretamente acoplada a linguagem, fazendo uso do MUnit. Sendo assim foi possível identificar que nos repositórios *open-source* minerados, não existe uma cultura madura o suficiente de desenvolvimento de testes automatizados até o presente momento.

# 6 Referências Bibliográficas


[1] Krishan Arora. The gaming industry: A behemoth with unprecedented global reach. *Forbes*, November 17 2023.

[2] Sidra Iftikhar, Muhammad Zohaib Iqbal, Muhammad Uzair Khan, and Wardah Mahmood. An automated model based testing approach for platform games. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 426–435. IEEE, 2015.

[3] Jussi Kasurinen, Maria Palacin-Silva, and Erno Vanhala. What concerns game developers? a study on game development processes, sustainability and metrics. In *2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 15–21, 2017.

[4] Gabriel Lovreto, Andre T Endo, Paulo Nardi, and Vinicius HS Durelli. Automated tests for mobile games: An experience report. In *2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pages 48–488. IEEE, 2018.

[5] Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. 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*, pages 1–11, 2014.

[6] Nathalya Stefhany Pereira, Phyllipe Lima, Eduardo Guerra, and Paulo Meirelles. Towards automated playtesting in game development. In *Anais Estendidos do XX Simpósio Brasileiro de Jogos e Entretenimento Digital*, pages 349–353. SBC, 2021.

[7] Cristiano Politowski, Yann-Gaël Guéhéneuc, and Fabio Petrillo. Towards automated video game testing: Still a long way to go. In *Proceedings of the 6th international ICSE workshop on games and software engineering: engineering fun, inspiration, and motivation*, pages 37–43, 2022.

[8] Cristiano Politowski, Fabio Petrillo, and Yann-Gaël Guéhéneuc. A survey of video game testing. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 90–99, 2021.

[9] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911, New York, New York, USA, 2018. ACM Press.

[10] Marco Tulio Valente. Engenharia de software moderna. *Princípios e Práticas para Desenvolvimento de Software com Produtividade*, 1, 2020.

[11] James A Whittaker. What is software testing? and why is it so hard? *IEEE software*, 17(1):70–79, 2000.