Universidade Federal da Paraíba
Centro de Informática
Graduação em Engenharia da Computação

Proposta de testes para serious games voltados ao condicionamento físico e reabilitação com aplicação no jogo GiroJampa

Rebeca Raab Bias Ramos

João Pessoa - PB
2024

Rebeca Raab Bias Ramos

Proposta de testes para serious games voltados ao condicionamento físico e reabilitação com aplicação no jogo GiroJampa

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal da Paraíba (UFPB), como requisito para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: Liliane dos Santos Machado

João Pessoa - PB
2024

Catalogação na publicação
Seção de Catalogação e Classificação

Ramosp Ramos, Rebeca Raab Bias.
Proposta de testes para serious games voltados ao condicionamento físico e reabilitação com aplicação no jogo girojampa / Rebeca Raab Bias Ramos. - João Pessoa, 2024.
57 f. : il.

Orientação: Liliane dos Santos Machado.
TCC (Graduação) - UFPB/CI.

1. Testes unitários automatizados. 2. Cobertura de código. 3. Refatoração. 4. Desenvolvimento de jogos. 5. Unity. 6. Coverlet. I. Machado, Liliane dos Santos. II. Título.

UFPB/CI CDU 004.4

Elaborado por Michelle de Kássia Fonseca Barbosa - CRB-738

Rebeca Raab Bias Ramos

Proposta de testes para serious games voltados ao condicionamento físico e reabilitação com aplicação no jogo GiroJampa

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal da Paraíba (UFPB), como requisito para obtenção do grau de Bacharel em Engenharia da Computação.

Trabalho aprovado. João Pessoa - PB, 17 de novembro de 2023:

Documento assinado digitalmente
gov.br LILIANE DOS SANTOS MACHADO
Data: 03/06/2024 10:58:02-0300
Verifique em https://validar.iti.gov.br

Liliane dos Santos Machado
Orientador

Documento assinado digitalmente
gov.br CLAUIRTON DE ALBUQUERQUE SIEBRA
Data: 03/06/2024 11:26:41-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Clauirton Albuquerque Siebra

Documento assinado digitalmente
gov.br ZILDOMAR CARLOS FELIX
Data: 04/06/2024 10:37:23-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Zildomar Carlos Felix
Examinador

João Pessoa - PB
2024

Agradecimentos

Em primeiro lugar, a Deus, para Ele seja toda honra e glória para sempre. Sou a prova viva da sua bondade e carinho, demonstrado em tantos momentos, que incluem também a conclusão deste curso. O Senhor que é o Doutor dos doutores me ensinou durante esta trajetória com muita paciência, a arte de insistir e não desistir. Quando cheguei no curso tinha um sonho que era ser engenheira e uma promessa de conclusão com Deus, devo dizer que não foi fácil, foi um caminho de lágrimas, aprendizado, reanimo, e amor. Durante o andamento do curso o Senhor preparou surpresas, me fortaleceu com suas palavras, me capacitou, me fez entender que quando Ele quer tudo acontece.

Aos meus pais, que me incentivaram nos momentos difíceis, e se alegraram por cada conquista. Meu pai com sabedoria, que foi instrumento de Deus em tantos momentos, dando palavras de ânimo e de fé, que fizerem alicerces para meus objetivos, sua presença, seu exemplo de esforço, me encorajaram e tiveram papel importante nessa conquista. Minha mãe com sua força, e apoio incondicional de muitas formas, sempre ao meu lado torcendo e orando por cada avanço meu, vibrando por cada conquista, sua dedicação, doação e apoio foram fundamentais para a minha conclusão do curso.

Ao meu esposo, que é meu companheiro nessa jornada. Um verdadeiro presente que ganhei de Deus ao longo da graduação, um amigo de todas as horas, que me ajudou em diversos momentos com seu amor, presença, e paciência. Aprendo com ele todos os dias e me inspiro na sua dedicação e amor.

As minhas irmãs, cada uma da sua maneira solicitas a ajudar, participaram de toda essa caminhada, tornaram os momentos mais simples e trouxeram mais alegria nessa jornada, agradeço por todas as rizadas e apoio que foram tão importantes nessa etapa.

A professora Liliane, por todo conhecimento transmitido, por seu apoio desde a iniciação cientifica, me incentivando a almejar sonhos maiores, por ter sido minha orientadora e ter desempenhado tal função com dedicação e amizade. Agradeço também aos professores Clauirton e Zildomar, pela disponibilidade, suas correções e ensinamentos que me permitiram evoluir academicamente e apresentar um melhor desempenho no meu processo de formação profissional, especialmente na conclusão do curso.

Aos amigos que fiz durante a graduação, com quem convivi intensamente durante os últimos anos, pelo companheirismo e pela troca de experiências que me permitiram crescer não só como pessoa, mas também como formando.

Por fim, agradeço a todos aqueles que contribuíram, de alguma forma, para a realização deste trabalho.

Resumo

Este trabalho explorou a implementação abrangente de testes unitários automatizados no jogo GiroJampa, utilizando o *framework NUnit* juntamente com a ferramenta de cobertura de código *Coverlet*. Os testes foram aplicados extensivamente após a refatoração das classes do jogo, o que permitiu entender melhor os fluxos de desenvolvimento e preparar os *scripts* do jogo para serem testados. Os testes automatizados unitários, escritos em C#, cobriram desde operações fundamentais até fluxos complexos. A reavaliação do código e das partes não testadas foi baseada no percentual de cobertura das classes do GiroJampa, calculado pela relação entre as linhas de código cobertas por testes e o total de linhas de código do jogo. Os resultados destacam uma cobertura significativa dos testes nas classes refatoradas, que apresentaram uma cobertura de testes maior do que as classes não refatoradas. A implementação de testes automatizados unitários no jogo GiroJampa revelou-se vantajosa, trazendo melhorias significativas ao processo de desenvolvimento. Esses testes proporcionaram uma identificação rápida e eficiente de problemas e facilitaram a revisão e correção de código. Além disso, serviram como uma documentação dinâmica do sistema, simplificando seu entendimento e manutenção. Como resultado, a confiança na estabilidade e na evolução contínua do jogo aumentou, assegurando que novas funcionalidades fossem integradas sem comprometer as existentes. Assim, os testes automatizados unitários mostraram-se essenciais para a qualidade e a sustentabilidade do desenvolvimento do GiroJampa. Como trabalhos futuros, é válido criar mais casos de testes para as classes que não sofreram refatoração, além de explorar mais ferramentas de relatório de cobertura de código. Sendo assim, este trabalho contribui para a literatura no âmbito do desenvolvimento e construção de testes unitários automatizados de jogos em ambientes *Unity*.

Testes Unitários Automatizados, Cobertura de Código, Refatoração, Desenvolvimento de Jogos, Unity, Coverlet

Abstract

This work explored the comprehensive implementation of automated unit tests in the game GiroJampa, using the NUnit framework along with the Coverlet code coverage tool. The tests were extensively applied after refactoring the game's classes, which allowed for a better understanding of the development flows and preparation of the game scripts for testing. The automated unit tests, written in C#, covered everything from fundamental operations to complex flows. The reassessment of the code and untested parts was based on the coverage percentage of GiroJampa classes, calculated by the ratio of lines covered by tests to the total lines of code in the game. The results highlight significant test coverage in the refactored classes, which showed greater test coverage than the non-refactored classes. The implementation of automated unit tests in the game GiroJampa proved to be advantageous, bringing significant improvements to the development process. These tests provided quick and efficient problem identification and facilitated code review and correction. Additionally, they served as a dynamic documentation of the system, simplifying its understanding and maintenance. As a result, confidence in the stability and continuous evolution of the game increased, ensuring that new functionalities could be integrated without compromising existing ones. Thus, automated unit tests proved essential for the quality and sustainability of GiroJampa development. For future work, it is valid to create more test cases for the classes that were not refactored, as well as to explore more code coverage reporting tools. Therefore, this work contributes to the literature in the field of automated unit test development and construction for games in Unity environments.

Automated Unit Tests, Code Coverage, Refactoring, Game Development, Unity, Coverlet

Lista de ilustrações

Sumário

1 Introdução

1.1 Definição do Tema

A realização de testes de software tem se tornado cada vez mais uma etapa fundamental no desenvolvimento de softwares de qualidade. Isso ocorre pois a realização dos testes permite identificar erros durante as etapas de desenvolvimento, garante a confiança do usuário final e sua satisfação ao utilizar o software, permite assegurar a qualidade e funcionamento correto do produto, além de manter a reputação do negócio no setor.

Por esse motivo, a área de testes de software tem crescido significativamente nos últimos tempos, em especial a automação de testes que está cada vez mais em evidência devido à agilidade e qualidade que pode trazer para o desenvolvimento de sistemas de software. Os testes automatizados podem ser eficazes e de baixo custo de implementação e manutenção e funcionam como um bom mecanismo para controlar a qualidade de sistemas (BERNARDO, 2011).

Geralmente, o mais comum é encontrar exemplos de aplicações de testes automatizados na industria de aplicativos sejam web ou mobile. Entretanto, os mesmos podem e devem ser usados em quaisquer áreas que façam uso de software. Em jogos, é bastante comum o uso do motor de desenvolvimento *Unity*, este mesmo conta com o *NUnit* que é uma das ferramentas mais importantes para garantia de qualidade. É usado para executar testes automatizados no Editor e em plataformas compatíveis. E está disponível para todos os usuários. Além de, ser possível acessar o código-fonte do UTF localmente, analisá-lo durante a depuração e modificá-la.

Dito isso, os resultados apontados pelo 2° Censo da Indústria Brasileira de Jogos Digitais apresentam um crescimento de empresas no Brasil, que produzem games, passando de 142 para 375 no período entre 2013 a 2018. Segundo a pesquisa do Homo Ludens, foram produzidos mais de 1700 jogos, em um período de 2 anos, sendo 43% para dispositivos móveis, 24% para computadores, 10% para plataformas de realidade virtual e aumentada e 5% para os consoles. Dentre esses jogos, cerca de 51% foram classificados como educativos e o restante para entretenimento (SAKUDA; FORTIM, 2018).

Logo, é possível afirmar que os jogos digitais contemplam vários setores, além do entretenimento, podendo adquirir um caráter “sério”, incorporando atividades de educação, pesquisa científica ou capacitação de profissionais em diversas áreas, tais como: saúde, arquitetura e construção civil, esses jogos são comumente denominados de serious games. De acordo com (RITTERFELD; CODY; VORDERER, 2009), a quantidade de profissionais interessados em usar jogos para educar, motivar e incentivar os jogadores, cresceu em

um curto período. Este movimento teve grande crescimento com o surgimento de portais e *websites*, como o "Games for Change1" e o "Games for Health2", que apresentam e incentivam este tipo de ferramentas educacionais, abrindo espaço para várias discussões e chamando a atenção de game designers, educadores e acadêmicos. Algumas propriedades de jogos, como interatividade e apresentação de regras e desafios, podem proporcionar experiências e diversão que motivam o usuário a continuar jogando, está é denominada satisfação (O'BRIEN; TOMS, 2008). Desta forma, os serious games podem ser um meio bastante eficiente para educação, desenvolvendo competências e atitudes, as quais podem ser desempenhadas em alguma situação real, bem como a construção do conhecimento sobre variados temas, relacionados como por exemplo à saúde (AMORIM, 2019).

Isto ocorre com jogos como GiroJampa, que faz uso de realidade virtual para simular uma atividade cotidiana, como um passeio pela orla de João Pessoa (BRAGA et al., 2021). O jogo tem por objetivo fazer com que os jogadores, que são pessoas que estão em fase de adaptação ou fazem uso de cadeiras de rodas, pratiquem atividade física ao mesmo tempo que ganham confiança a aplicar as habilidades desenvolvidas no jogo em uma situação real. Desenvolvido para uso individual (*single player*), o jogo remete a um cenário real para que os jogadores se sintam mais confiantes em situações do dia a dia. O projeto GiroJampa foi desenvolvido baseado em um ponto importante em jogos que é a obtenção da satisfação e durante seu desenvolvimento foram elaborados requisitos de design voltados a jogos específicos da saúde (RAMOS; MACHADO, 2021). Tais requisitos forneceram ao projeto elementos para estimular e entreter o jogador.

1.2 Justificativa

O jogo GiroJampa foi desenvolvido sem a devida consideração para a realização de testes que validassem a implementação do jogo. A abordagem inicial concentrou-se exclusivamente em testes de satisfação e usabilidade, os quais foram realizados apenas após a conclusão do desenvolvimento. Essa estratégia levanta dúvidas quanto à qualidade do código, tornando a manutenção e evolução do software tarefas potencialmente complexas. A construção de software é uma atividade intrinsecamente desafiadora e propensa a diversos tipos de problemas, especialmente os relacionados a erros humanos.

Dentro desse cenário, esta iniciativa propõe-se a submeter o GiroJampa a testes, com o objetivo de avaliar a funcionalidade do jogo. Por meio da simulação antecipada das ações do jogador, busca-se não apenas verificar a aderência do funcionamento às expectativas, mas também avaliar o desenvolvimento do código. Esta abordagem visa facilitar a manutenção do software e garantir sua qualidade, considerando as complexidades inerentes à construção de software e os desafios associados a possíveis problemas, especialmente os relacionados a erros humanos.

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo geral deste trabalho foi, realizar a testagem dos componentes do jogo GiroJampa, usando para isso os conceitos apresentados na literatura para construção de testes automatizados unitários. Com intuito de conferir a funcionalidade do jogo.

1.3.2 Objetivos específicos

No intuito de alcançar o objetivo geral supracitado, os seguintes objetivos específicos foram abordados neste trabalho:

- Compreender o conceito de testes de software e sua aplicação;
- Mapear possíveis casos de testes para o jogo GiroJampa;
- Expor o processo de desenvolvimento dos testes unitários automatizados;
- Evidenciar as estratégias utilizadas na codificação dos testes unitários automatizados;
- Discutir os impactos no jogo GiroJampa, comparando se houveram impactos significativos após a implementação dos testes automatizados unitários no jogo;

1.4 Estrutura do Trabalho

Este trabalho está organizado da seguinte forma: a Seção 2 apresenta os conceitos gerias e revisão da literatura, a seção 3 aborda a metodologia aplicada, na qual é discutida em sintase a descrição do problema, materiais e métodos usados. A Seção 4 apresenta uma discussão sobre os resultados encontrados e impactos. A Seção 5 apresenta as considerações finais.

2 Conceitos Gerais e Revisão da Literatura

Nesta seção serão discutidos alguns conceitos e principais características na elaboração de testes de software, abordando processos para sua construção, o papel dos testes para uma aplicação de qualidade e suas características, conversões de escrita, e outros aspectos que permeiam o desenvolvimento do mesmo. Além de discorrer sobre o jogo a ser testado o GiroJampa.

2.1 Engenharia de Software

A crescente dependência da sociedade moderna em sistemas computacionais coloca uma ênfase sem precedentes na qualidade e confiabilidade do software. Nesse contexto, a Engenharia de Software assume um papel fundamental para projetar, desenvolver e manter sistemas de software eficientes e de alta qualidade.

A Engenharia de Software se trata da área da computação que se preocupa com os fatores do desenvolvimento de software, desde o início de sua construção até a sua fase de execução e manutenção (SOMMERVILLE, 2011), contribuindo assim com a qualidade do produto final.

Pressman e Maxim (PRESSMAN; MAXIM, 2016) definem a engenharia de software em 4 camadas, como mostra a Figura 1:

- **Ferramentas**: Responsável por fornecer suporte automatizado ou semi automatizado aos métodos e processos.
- **Métodos**: Responsável pelo fornecimento de informações técnicas para o desenvolvimento do software. Modelagem de projeto, comunicação, análise dos requisitos e testes são algumas de suas atividades.
- **Processo**: Base da engenharia de software responsável por manter as camadas de tecnologia coesas e permitir que o software seja desenvolvido dentro do prazo e de forma racional.
- **Foco na qualidade**: camada que sustenta a engenharia de software.

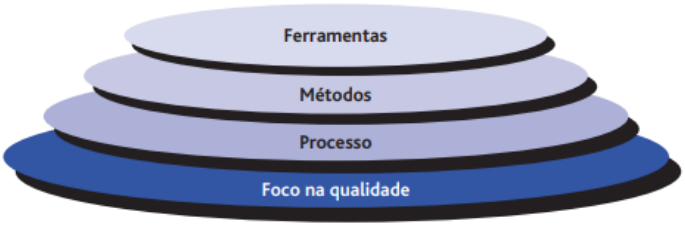


Figura 1 – Camadas da engenharia de software.

Fonte: (PRESSMAN; MAXIM, 2016)

Sendo assim, a Engenharia de Software desempenha um papel crítico na evolução e inovação tecnológica. Ao fornecer diretrizes, práticas e padrões para o desenvolvimento de software, ela permite que equipes de desenvolvimento criem produtos de alta qualidade, confiáveis e eficientes.

A Engenharia de Software abrange uma variedade de metodologias (*frameworks*) de processo que estabelecem o alicerce para um processo de engenharia de software completo por meio da identificação de um pequeno número de atividades metodológicas aplicáveis a todos os projetos de software, independentemente de tamanho ou complexidade. Além disso, a metodologia de processo engloba um conjunto de atividades de apoio (*umbrella activities*) aplicáveis a todo o processo de software(PRESSMAN; MAXIM, 2016).

2.1.1 Modelos de Processo

Um processo define quem está fazendo o quê, quando e como para atingir determinado objetivo (JACOBSON, 1999). Em outras palavras, processo é um conjunto de atividades, ações e tarefas realizadas na criação de algum artefato. Uma atividade se esforça para atingir um objetivo amplo (por exemplo, comunicar-se com os envolvidos) e é utilizada independentemente do campo de aplicação, do tamanho do projeto, da complexidade dos esforços ou do grau de rigor com que a engenharia de software será aplicada. Uma ação (por exemplo, projeto de arquitetura) envolve um conjunto de tarefas que resultam em um artefato de software fundamental (por exemplo, um modelo arquitetural). Uma tarefa se concentra em um objetivo pequeno, porém bem-definido (por exemplo, realizar um teste de unidades), e produz um resultado tangível(PRESSMAN; MAXIM, 2016).

Os processos de software, às vezes, são categorizados como dirigidos a planos ou processos ágeis. Processos dirigidos a planos são aqueles em que todas as atividades são planejadas com antecedência, e o progresso é avaliado por comparação com o planejamento inicial. Em processos ágeis, o planejamento é gradativo, e é mais fácil alterar o processo de maneira a refletir as necessidades de mudança dos clientes. Cada abordagem é apropriada para diferentes tipos de software (BOEHM; TURNER, 2003). Geralmente, é necessário encontrar um equilíbrio entre os processos dirigidos a planos e os processos ágeis.

Existem vários processos de desenvolvimento de software diferentes, mas todas devem incluir quatro atividades fundamentais para a engenharia de software:

- **Especificação de software**: A funcionalidade do software e as restrições a seu funcionamento devem ser definidas.
- **Projeto e implementação de software**: O software deve ser produzido para atender às especificações.
- **Validação de software**: O software deve ser validado para garantir que atenda às demandas do cliente.
- **Evolução de software**: O software deve evoluir para atender às necessidades de mudança dos clientes.

Um modelo de processo de software é uma representação simplificada de um processo de software. Cada modelo representa uma perspectiva particular de um processo e, portanto, fornece informações parciais sobre ele. Por exemplo, um modelo de atividade do processo pode mostrar as atividades e sua sequência, mas não mostrar os papéis das pessoas envolvidas (SOMMERVILLE, 2011). Cada modelo de processo também prescreve um fluxo de processo (também denominado fluxo de trabalho) (PRESSMAN; MAXIM, 2016).

Existem diferentes modelos de desenvolvimento de software, cada um com suas próprias abordagens e características. Três modelos notáveis são o Modelo em Cascata (ou Clássico), o Modelo Incremental e o Modelo Evolucionário. Cada um desses modelos possui vantagens e desvantagens específicas, tornando-os mais apropriados para diferentes contextos de desenvolvimento de software. A escolha do modelo mais adequado dependerá das características do projeto, dos requisitos do cliente e da dinâmica do ambiente em que o software será utilizado.

2.1.1.1 Modelo Cascata

O Modelo em Cascata, também conhecido como modelo clássico, adota uma abordagem sequencial e sistemática no desenvolvimento de software. Nesse modelo, cada fase do processo de desenvolvimento deve ser concluída antes que a próxima etapa seja iniciada. Esta abordagem assegura que não haja avanço sem a finalização prévia da fase anterior, como ilusta a Figura 2. As vantagens desse modelo incluem a ênfase no planejamento, garantia da qualidade em cada etapa e a igual importância atribuída a todas as fases. Contudo, ele pode ser menos adaptável a mudanças de requisitos, ocasionar agrupamento de testes de sistema, bem como tornar custoso o processo de correção de erros em estágios iniciais. Portanto, o Modelo em Cascata é mais indicado para projetos

em que os requisitos estão bem definidos desde o início e possuem baixa probabilidade de mudanças substanciais (VALENTE, 2020).

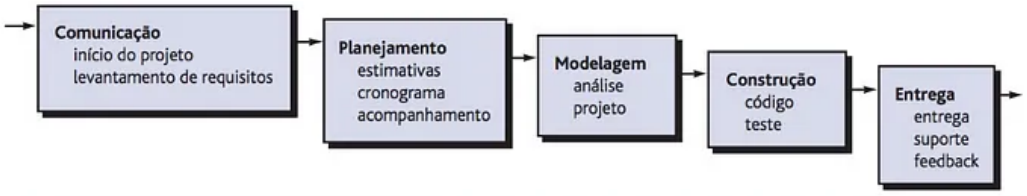


Figura 2 – Modelo Cascata

Fonte: (PRESSMAN; MAXIM, 2016)

Uma das variações do modelo em cascata é o modelo V, apresentado na Figura 3, o qual destaca a relação entre as atividades do processo e as atividades de teste. Uma característica destes modelos é a entrega do produto em sua totalidade, depois da realização de todas as atividades esperadas (ALÉSSIO; SABADIN; ZANCHETT, 2017) (SILVA et al., 2022).

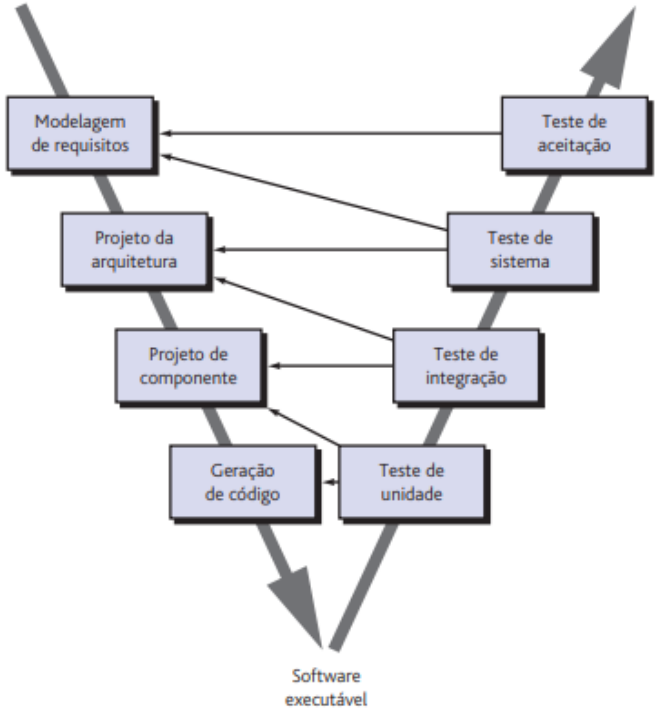


Figura 3 – Modelo V

Fonte: (PRESSMAN; MAXIM, 2016)

2.1.1.2 Modelo Incremental

O modelo incremental propõe entregas em incrementos, ou seja, versões progressivamente mais funcionais do software. Isso permite que os usuários utilizem o software

antes de estar completamente finalizado, reduzindo riscos e permitindo antecipação de problemas. O modelo é adequado quando há necessidade de entregas parciais e é útil para mitigar riscos de falhas em entregas completas. Os autores, Aléssio, Sabadin e Zanchett (ALÉSSIO; SABADIN; ZANCHETT, 2017), comentam que o modelo permite que o sistema seja separado por módulos ou subsistemas. Sua ideia está na produção e entrega de uma versão operacional do sistema para que o cliente possa avaliá-lo a cada ciclo de interação durante seu desenvolvimento, conforme mostra a Figura 4, até que o objetivo do sistema seja alcançado (FERNANDES, 2017).

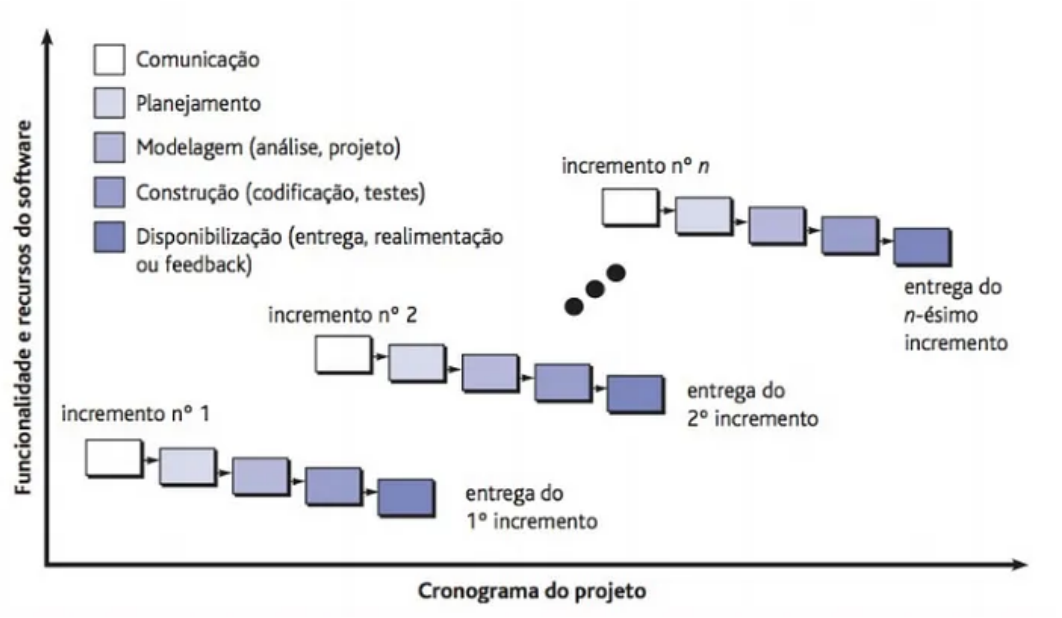


Figura 4 – Modelo Incremental.

Fonte: (PRESSMAN; MAXIM, 2016)

2.1.1.3 Modelo Evolucional

Os modelos evolucionários de processo de desenvolvimento de software têm como objetivo criar uma versão progressivamente mais completa do software a cada iteração. Essa abordagem é especialmente útil para projetos que envolvem mudanças constantes e onde os requisitos não estão completamente definidos desde o início. Dois modelos evolucionários proeminentes são a Prototipação e o Modelo Espiral, como mostra a Figura 5.

- **Prototipação**: aplicado quando o cliente tem uma necessidade legítima, mas carece de detalhes claros sobre os requisitos. Nesse caso, é criado um protótipo inicial que ajuda a identificar e refinar os requisitos do software. O processo envolve iterações em que o protótipo é ajustado conforme as necessidades do usuário se tornam mais claras. O modelo de Prototipação pode ser usado de forma independente ou em conjunto com outros processos (PRESSMAN; MAXIM, 2016).

- **Modelo Espiral**: proposto por Boehm (BOEHM et al., 1998), é caracterizado por iterações contínuas que ocorrem à medida que o desenvolvimento avança. Esse modelo é dividido em atividades metodológicas que representam pontos ao longo do caminho espiral. Cada iteração do Modelo Espiral começa no centro da espiral e prossegue no sentido horário. A primeira iteração pode resultar no desenvolvimento de uma especificação de produto. Conforme a espiral é percorrida novamente, podem ser desenvolvidos protótipos e, subsequentemente, versões cada vez mais sofisticadas do software (PRESSMAN; MAXIM, 2016).

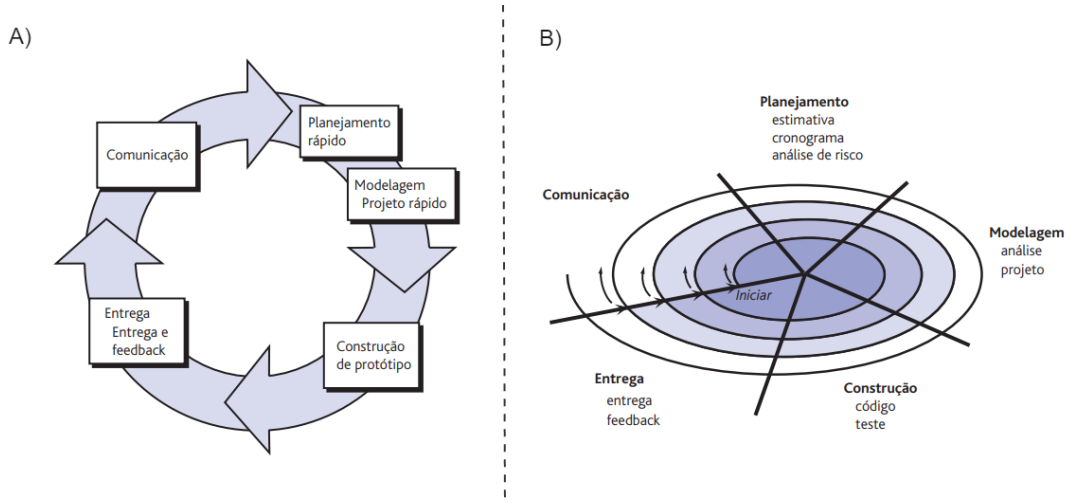


Figura 5 – A) Paradigma da prototipação e B) Modelo espiral Típico

Fonte: (PRESSMAN; MAXIM, 2016)

Em resumo, os modelos evolucionários são abordagens iterativas ou incrementais que permitem a adaptação contínua do software às mudanças e refinamento dos requisitos ao longo do tempo. Eles são particularmente úteis quando os detalhes dos requisitos não estão completamente definidos ou quando a flexibilidade e a capacidade de resposta às mudanças são essenciais para o sucesso do projeto.

2.2 Desenvolvimento Ágil

O desenvolvimento ágil de software é uma abordagem de desenvolvimento caracterizada pela adaptabilidade, ou seja, pela capacidade de absorver mudanças, sejam elas nas forças de mercado, nos requisitos do sistema, na tecnologia de implementação ou nas equipes de projeto (Figura 6) (MATTIOLI et al., 2009).

Nas últimas décadas, as metodologias ágeis de desenvolvimento de software ganharam destaque em relação aos modelos tradicionais, conforme apontado por um estudo recente do Project Manager Institute (PMI), que afirma que 71% das empresas de engenharia de software utilizam esse tipo de método. Entretanto, algumas organizações

ainda utilizam técnicas e métodos tradicionais, que continuam válidos em muitos casos, dependendo do projeto e da empresa (PMI, 2017).

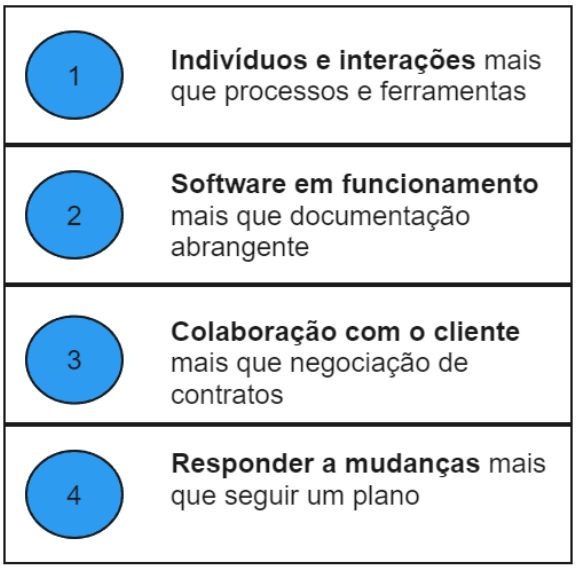


Figura 6 – Manifesto Ágil.

Fonte: Autoria própria.

Trazendo para uma perspectiva mais direcionada à construção de teste, quando o desenvolvimento era em cascata, os testes ocorriam em uma fase separada, após as fases de levantamento de requisitos, análise, projeto e codificação. Além disso, existia uma equipe separada de testes, responsável por verificar se a implementação atendia aos requisitos do sistema. Para checar isso, frequentemente os testes eram manuais, isto é, uma pessoa usava o sistema, informava dados de entrada e verificava se as saídas eram aquelas esperadas. Assim, o objetivo de tais testes era apenas detectar *bugs*, antes que o sistema entrasse em produção (VALENTE, 2020).

Com métodos ágeis, a prática de testes de software foi profundamente reformulada, conforme explicamos a seguir:

- **Grande parte dos testes passou a ser automatizada**: Isto é, além de implementar as classes de um sistema, os desenvolvedores passaram a implementar também código para testar tais classes. Assim, os programas tornaram-se auto-testáveis.
- **Testes não são mais implementados após todas as classes de um sistema ficarem prontas**: Muitas vezes, eles são implementados até mesmo antes dessas classes.
- **Não existem mais grandes equipes de testes**: Ou elas são responsáveis por testes específicos. Em vez disso, o desenvolvedor que implementa uma classe também deve implementar os seus testes.

- **Testes não são mais um instrumento exclusivo para detecção de *bugs***: Os testes ganharam novas funções, como verificar se uma classe continuará funcionando após um *bug* ser corrigido em uma outra parte do sistema. Além disso, os testes passaram também a usados como documentação para o código de produção.

2.3 Qualidade de Software

Qualidade de software é uma das áreas da engenharia de software que objetiva garantir a qualidade do produto através do processo de desenvolvimento definido (PERUCCI; CAMPOS, 2016). Ela aborda um processo metódico, focado nos artefatos e em todas as etapas desenvolvidas, que busca garantir a conformidade de produtos e processos, evitando ou eliminando possíveis defeitos (SAMBO, 2018).

A qualidade do software está diretamente ligada à qualidade do processo de seu desenvolvimento. Sua avaliação refere-se a um processo subjetivo por meio do qual é preciso verificar se o software corresponde ou não a sua finalidade diante das suas características (SOMMERVILLE, 2018).

Os processos de software servem de alicerce para garantir a qualidade dos projetos. Se um software foi desenvolvido tendo como base processos de software, incluindo processos de qualidade, a probabilidade de falhas ou necessidade de retrabalho é consideravelmente menor. Empresas de software que incorporam modelos de processos de software e seguem suas práticas tem maior chance de reconhecimento, uma vez que trabalham sob orientações e padrões de qualidade definidos e recomendados pela comunidade de especialistas do setor de desenvolvimento de software (ALÉSSIO; SABADIN; ZANCHETT, 2017).

Além da implementação correta das funcionalidades do software, sua qualidade depende dos atributos não funcionais. Os atributos mais importantes de qualidade do software referem-se a sua eficiência, confiabilidade, manutenção e segurança (SOMMERVILLE, 2018). A concepção dos responsáveis pela avaliação, incluindo desenvolvedor, usuário e organização, influencia a qualidade do software, avaliando conformidade, aspectos internos de desempenho, custo, facilidade de uso e confiabilidade dos resultados, bem como aspectos de custo, cronograma e conformidade, de acordo com os requisitos estabelecidos (SILVA et al., 2022).

No âmbito do processo de Garantia da Qualidade de Software (SQA), um conjunto abrangente de atividades é implementado para garantir que os processos de desenvolvimento de software sejam não apenas definidos, mas também continuamente otimizados. Estas atividades incluem a elaboração e implementação de padrões e diretrizes, a realização de auditorias e revisões de processos, a definição de métricas de qualidade e o estabelecimento de procedimentos para correção contínua e aprimoramento. O objetivo fundamental do SQA é assegurar que os produtos de software atendam plenamente às especificações e

objetivos estabelecidos, reduzindo a incidência de falhas, minimizando a necessidade de retrabalho e promovendo a eficácia global do processo de desenvolvimento de software. Ao incorporar práticas de SQA, as organizações buscam não apenas conformidade com padrões de qualidade, mas também uma melhoria constante em seus processos, o que contribui para a reputação positiva e o reconhecimento no competitivo setor de desenvolvimento de software (SAMBO, 2018).

O SQA envolve atividades como estabelecer padrões de desenvolvimento, realizar revisões e auditorias para garantir a conformidade, planejar e conduzir testes de qualidade, coletar e analisar dados de erros/defeitos, gerenciar mudanças, promover a educação contínua, gerenciar fornecedores, administrar a segurança e lidar com a gestão de riscos (PRESSMAN; MAXIM, 2016). A aplicação de ferramentas pela SQA verifica se os padrões, técnicas e métodos definidos na engenharia de software estão sendo aplicados corretamente pelos profissionais envolvidos (SANTOS; OLIVEIRA, 2017).

2.4 Testes de Software

Com o avanço tecnológico e a crescente dependência de sistemas computacionais em nossa sociedade, a qualidade do software tornou-se um fator crítico para o sucesso de organizações e empresas. Os testes de software desempenham um papel fundamental nessa busca pela excelência, pois garantem que os programas e sistemas desenvolvidos atendam aos requisitos funcionais, sejam confiáveis e ofereçam uma experiência satisfatória aos usuários.

Testes de software são uma parte essencial do desenvolvimento de uma aplicação, sendo um conjunto de atividades que têm como objetivo avaliar e verificar a qualidade e o desempenho de um programa ou sistema de computador. Essas atividades são realizadas para identificar defeitos, erros e falhas de funcionalidade antes que o software seja disponibilizado ao público ou implementado em um ambiente de produção. Essa prática sistemática envolve a execução de cenários de teste, onde o software é submetido a uma variedade de entradas e condições, com o intuito de verificar seu comportamento e verificar se os resultados são coerentes com o esperado.

Inicialmente, precisamos conhecer a diferença entre Defeitos, Erros e Falhas (ENGINEERS, 1990). A Figura 7, expressa a diferença entre esses conceitos. Defeitos fazem parte do universo físico (a aplicação propriamente dita) e são causados por pessoas, por exemplo, através do mal uso de uma tecnologia. Defeitos podem ocasionar a manifestação de erros em um produto, ou seja, a construção de um software de forma diferente ao que foi especificado (universo de informação). Por fim, os erros geram falhas, que são comportamentos inesperados em um software que afetam diretamente o usuário final da aplicação (universo do usuário) e pode inviabilizar a utilização de um software. Dessa forma, ressaltamos que teste de software revela simplesmente falhas em um produto

(NETO, 2007).

Figura 7 – Defeito x Erro x Falha.

Fonte: Adaptado de (NETO, 2007) e (JORDAN, 2018)

A atividade de teste é composta por alguns elementos essenciais que auxiliam na formalização desta atividade. Esses elementos são os seguintes:

- **Caso de Teste**: Descreve uma condição particular a ser testada e é composto por valores de entrada, restrições para a sua execução e um resultado ou comportamento esperado (CRAIG; JASKIEL, 2002)).
- **Procedimento de Teste**: É uma descrição dos passos necessários para executar um caso (ou um grupo de casos) de teste (CRAIG; JASKIEL, 2002).
- **Critério de Teste**: Serve para selecionar e avaliar casos de teste de forma a aumentar as possibilidades de provocar falhas ou, quando isso não ocorre, estabelecer um nível elevado de confiança na correção do produto (ROCHA, 2001). Os critérios de teste podem ser utilizados como:
  - **Critério de Cobertura dos Testes**: Consiste na identificação de partes do programa que devem ser executadas para garantir a qualidade do software e indicar quando o mesmo foi suficientemente testado, o mesmo serve para indicar trechos que estão ou não sendo testados (DELAMARO; JINO; MALDONADO, 2013). Ou seja, determinar o percentual de elementos necessários por um critério de teste que foram executados pelo conjunto de casos de teste.
  - **Critério de Adequação de Casos de Teste**: Quando, a partir de um conjunto de casos de teste T qualquer, ele é utilizado para verificar se T satisfaz os requisitos de teste estabelecidos pelo critério. Ou seja, este critério avalia se os casos de teste definidos são suficientes ou não para avaliação de um produto ou uma função (ROCHA, 2001).
  - **Critério de Geração de Casos de Teste**: Quando o critério é utilizado para gerar um conjunto de casos de teste T adequado para um produto ou função, ou seja, este critério define as regras e diretrizes para geração dos casos de

teste de um produto que esteja de acordo com o critério de adequação definido anteriormente (ROCHA, 2001).

2.4.1 Testes de Software, por que tornar um teste automatizado?

Para assegurar a qualidade do programa os testes de software devem ser repetidos frequentemente durante os ciclos de desenvolvimento. Cada vez que o código-fonte é modificado, os testes de software devem ser repetidos. Repetir manualmente esses testes é caro e demorado. Uma vez criados, os testes automatizados podem ser executados repetidamente sem custo adicional e são muito mais rápidos do que os testes manuais. O teste de software automatizado pode reduzir o tempo de execução de testes repetitivos de dias para horas. Uma economia de tempo que se traduz diretamente em economia de custos.

Para Spadini, Aniche, e Bacchelli (SPADINI; ANICHE; BACCHELLI, 2018), o teste automatizado é um processo essencial para a qualidade do software. Os testes podem ajudar a assegurar que o código é robusto, ou seja, as funcionalidades do sistema funcionam sob diferentes cenários de uso, e que o código atende às necessidades de performance e segurança.

Para Jamil, et. al (JAMIL et al., 2016), o teste de software refere-se a encontrar bugs, erros ou requisitos ausentes no sistema de software desenvolvido. Para os autores, esta é uma investigação que fornece às partes interessadas o conhecimento exato sobre a qualidade do produto.

Os testes automatizados consistem na utilização de *scripts*, ferramentas e abordagens de automação para executar testes de forma rápida, repetitiva e consistente. Diferentemente dos testes manuais, que demandam tempo e recursos humanos significativos, os testes automatizados permitem que desenvolvedores e equipes de qualidade possam testar de maneira ágil, integrando os testes diretamente ao processo de desenvolvimento. Como quem realiza o teste é a máquina através da execução automática de um programa (de teste), este fator tornar a execução dos testes mais rápida e eficiente (VALENTE, 2020). Logo, o objetivo é que a máquina monte o cenário, execute a ação, e valide o código sem a necessidade de interferências.

Algumas das vantagens de se automatizar testes segundo Rafi et al. (RAFI et al., 2012) e Gokulnath (GOKULNATH, 2023), com base em uma revisão sistemática da literatura e pesquisa com profissionais:

- **Aumento na cobertura do teste**: Mais *scripts* podem ser testados ao mesmo tempo, aumentando a cobertura do teste. Isso ajuda a economizar tempo e diminui a carga dos testadores manuais. O teste de software automatizado tem o potencial de melhorar a qualidade do software, expandindo o alcance e o rigor dos testes. Ele

permite a execução de testes demorados que normalmente são evitados em testes manuais, permitindo que sejam executados sem supervisão. Além disso, facilita a execução de testes em vários computadores com diversas configurações.

- **Maior precisão**: Testes contínuos aumentam as chances de erros quando feitos manualmente, mas em testes automatizados, testes repetitivos podem ser executados com a mesma precisão.
- **Economia de custos**: Com testes automatizados, os casos de teste são executados em uma velocidade mais rápida e os *bugs* são identificados no início do ciclo de desenvolvimento e corrigidos. Apesar dos altos custos iniciais, uma vez que a estrutura automatizada é configurada, há uma redução geral nos custos.
- **Reduz o tempo de teste de regressão**: A regressão automatizada praticada com ferramentas de teste permite que os testadores manuais sejam liberados da execução de testes de regressão monótonos. Os testadores têm tempo para realizar tarefas de valor agregado.
- **Executa tarefas que não podem ser executadas por testadores manuais**: Certos testes, como os testes de aplicativos da Web controlados, podem ser simulados com testes automatizados, mas não podem ser executados por testadores manuais.
- **Economia de tempo**: Com o teste automatizado, os *scripts* manuais também são automatizados e o tempo de teste de regressão é reduzido. Como os testes são executados 24 horas por dia, 7 dias por semana, os testes automatizados economizam tempo. O teste de automação economiza um tempo significativo ao executar testes mais rapidamente em comparação com o teste manual. Testes longos que podem consumir muito tempo para testadores manuais podem ser executados sem supervisão, permitindo que os testadores se concentrem em outras tarefas críticas.
- **Reutilização de *scripts* de teste**: Em testes automatizados, o mesmo *script* pode ser usado com pequenas alterações. Assim, a reutilização dos *scripts* de teste facilita o processo de teste, e também, os mesmos *scripts* podem ser armazenados e reutilizados para repetir o teste quando houver necessidade.
- **Útil para os testadores**: Como os testes podem ser executados automaticamente quando o código-fonte é alterado e notificar os testadores sobre qualquer problema, é muito útil tanto para os testadores quanto para os desenvolvedores.
- **O retorno do investimento é alto**: a automação ajuda as empresas a concluir o processo de teste mais rapidamente com maior precisão e cobertura, resultando em um alto retorno do investimento.

Uma forma interessante de classificar testes automatizados é por meio de uma pirâmide de testes, originalmente proposta por Mike Cohn (COHN, 2010). Como mostra a próxima Figura 8, essa pirâmide particiona os testes de acordo com sua granularidade.

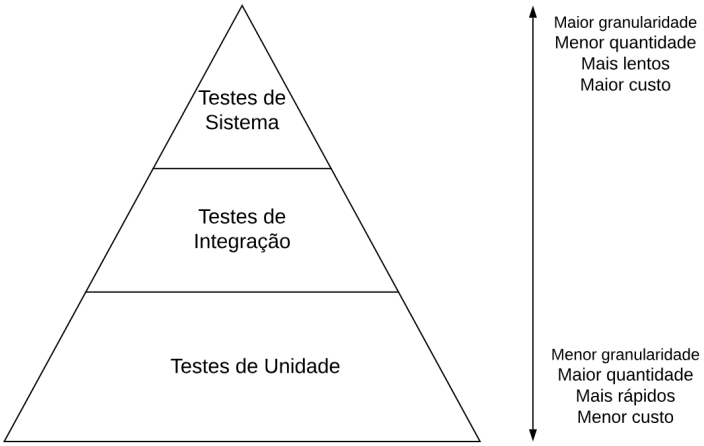


Figura 8 – Pirâmide de testes.

Fonte: (VALENTE, 2020)

2.4.2 Fases da atividade de testes

A atividade de teste é complexa. São diversos os fatores que podem colaborar para a ocorrência de erros, como já foi mencionado. Por isso, a atividade de teste é dividida em fases com objetivos distintos. De uma forma geral, podemos estabelecer como fases o teste de unidade (seção 2.4.3), o teste de integração (seção 2.4.4) e o teste de sistemas(seção 2.4.5).

Além dessas três fases de teste, destaca-se, ainda, o que se costuma chamar de "teste de regressão". Esse tipo de teste não se realiza durante o processo "normal" de desenvolvimento, mas sim durante a manutenção do software. A cada modificação efetuada no sistema, após a sua liberação, corre-se o risco de que novos defeitos sejam introduzidos. Por esse motivo, é necessário, após a manutenção, realizar testes que mostrem que as modificações efetuadas estão corretas, ou seja, que os novos requisitos implementados (se for esse o caso) funcionam como o esperado e que os requisitos anteriormente testados continuam válidos (DELAMARO; JINO; MALDONADO, 2013).

Independentemente da fase de teste, existem algumas etapas bem definidas para a execução da atividade de teste. São elas: 1) planejamento; 2) projeto de casos de teste; 3) execução; e 4) análise. Como é possível observar na figura 9.

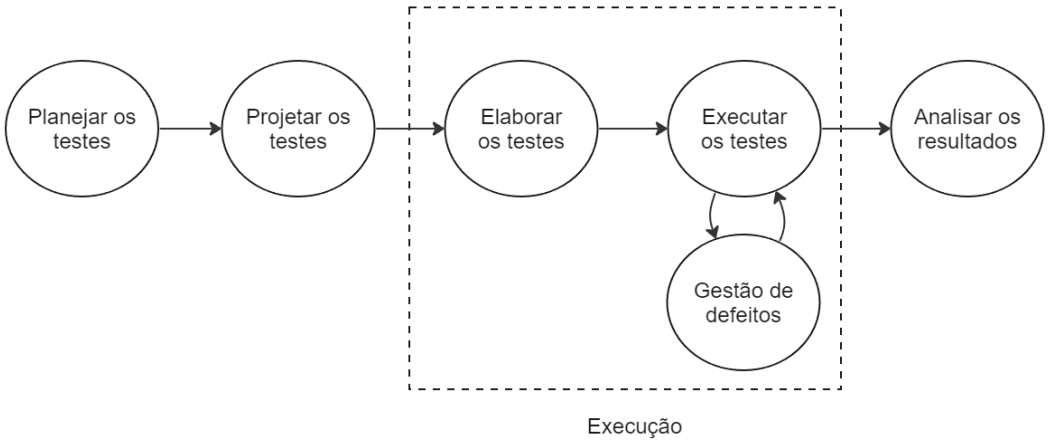


Figura 9 – Etapas para construção de testes

Fonte: Adaptado de (CUNHA, 2010)

2.4.3 Testes de Unidade

Testes de unidade ou testes unitários são testes aplicados nas menores unidades de código para garantir que haja uma integridade na aplicação que ela cumpra com as definições do escopo (figura 10). Segundo Thomas e Hunt (THOMAS; HUNT, 2019), uma unidade é um método ou função testável de uma aplicação que exerce uma pequena área específica da funcionalidade do código a ser testado.

A ideia fundamental do teste de unidade é testar apenas uma parte da funcionalidade isoladamente, o que significa que todas as outras partes da aplicação das quais a função depende não são usadas pelo teste. Em vez disso, eles são imitados. Imitar um objeto *JavaScript* é criar um falso que simula o comportamento do objeto real. No teste de unidade, o objeto falso é chamado de *mock* e o processo de criá-lo é chamado de *mocking*. Ou seja, *mock* é um teste duplo que ajuda a emular e examinar as interações entre o sistema em teste e suas dependências (KHORIKOV, 2020).

Os testes unitários são a base da piramide de testes e têm como objetivo verificar a corretude do código em nível de unidades individuais, como funções, métodos ou classes. Cada unidade é testada de forma isolada para garantir que ela produza os resultados esperados com diferentes entradas e condições. Esses testes são escritos pelos próprios desenvolvedores, geralmente utilizando estruturas de teste como *JUnit* (para Java) ou *Pytest* (para Python).

Quando consideramos o software orientado a objetos, o conceito de unidades se modifica. O encapsulamento controla a definição de classes e objetos. Isso significa que cada classe e cada instância de uma classe (objeto) empacotam atributos (dados) e as operações que manipulam esses dados. Uma classe encapsulada é usualmente o foco do teste de unidade. No entanto, operações (métodos) dentro da classe são as menores

unidades testáveis. Como uma classe pode conter um conjunto de diferentes operações, e uma operação em particular pode existir como parte de um conjunto de diferentes classes, a tática aplicada ao teste de unidade precisa ser modificada (PRESSMAN; MAXIM, 2016).

Ou seja, no paradigma de orientação a objetos seriam as classes e seus métodos as unidades a ser testadas, seu objetivo é garantir que cada parte do código tenha sido desenvolvida de acordo com o que foi especificado eque não haja comportamentos inesperados (LIMA; LEAL; MESQUITA, 2023).

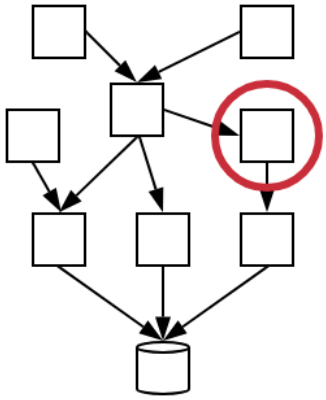


Figura 10 – Escopo de testes de unidade.

Fonte: (VALENTE, 2020)

2.4.4 Testes de Integração

Teste de integração é quando os módulos (unidades de código) são testados em grupo para garantir que não haja nenhuma quebra naquilo que foi feito unitariamente e naquilo que está sendo integrado junto. O cenário ideal é que sejam feitos testes de unidades primeiro e, depois disso, seja feito o teste de integração que busca compreender se os módulos funcionarão conjuntamente (KRIGER, 2021).

Em outras palavras, o teste de integração é uma atividade sistemática que tem como objetivo verificar a construção da estrutura do software que está sendo desenvolvido e a sua comunicação entre módulos, Figura 11. Estes, testam a classe de maneira integrada ao serviço que usam. Um teste de DAO, por exemplo, que bate em um banco de dados de verdade, é considerado um teste de integração. Testes como esses são especialmente úteis para testar classes cuja responsabilidade é se comunicar com outros serviços (ANICHE, 2015).

Devido ao software orientado a objetos não ter uma estrutura óbvia de controle hierárquico, as estratégias tradicionais de integração descendente e ascendente têm pouco significado. Além disso, integrar operações uma de cada vez em uma classe (a abordagem convencional de integração incremental) frequentemente é impossível devido “às interações diretas e indiretas dos componentes que formam a classe” (BERARD, 1993). Há duas

estratégias diferentes para teste de integração de sistemas OO (BINDER, 1994). A primeira, teste baseado em sequência de execução (*thread-based testing*), e a segunda abordagem, baseado em uso (*use-based testing*) (PRESSMAN; MAXIM, 2016).

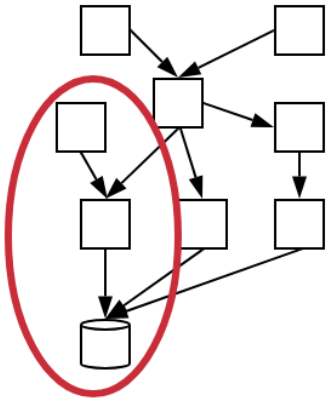


Figura 11 – Escopo de testes de integração.

Fonte: (VALENTE, 2020)

2.4.5 Testes de Sistemas

O teste de sistema centra-se no comportamento e nas capacidades de todo um sistema ou produto, Figura 2.4.5. Consideram-se as execuções das tarefas do sistema de ponta a ponta (*End to End Test*) e os comportamentos não funcionais exibidos ao executar tais tarefas (PAIVA, 2021).

Na prática, o objetivo é executar o sistema sob o ponto de vista do seu usuário final, varrendo as funcionalidades em busca de falhas. Os testes são executados em condições similares - de ambiente, interfaces sistêmicas e massas de dados - àquelas que um usuário utilizará no seu dia-a-dia de manipulação do sistema. De acordo com a política de uma organização, podem ser utilizadas condições reais de ambiente, interfaces sistemáticas e massas de dados (SILVA; ABREU; AGUIAR, 2017). Porém, é mais difícil de ser escrito, exige maior trabalho de manutenção e leva mais tempo para executar, além de ser mais caro.

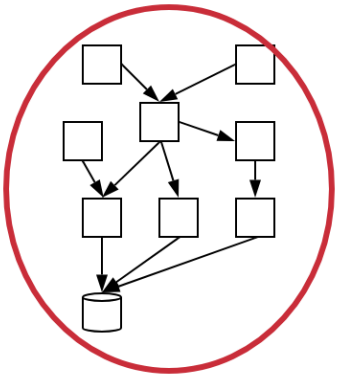


Figura 12 – Escopo de testes de sistema.

Fonte: (VALENTE, 2020)

2.5 Características e Convenções de escrita de testes

Por padrão, todos os testes vão ter a mesma estrutura: definição do teste, nome da unidade, classe, ou função que vai ser testada e função *matcher* que verifica o esperado com o que se tem de fato. Uma das convenções mais comuns na escrita de testes é acrescentar o sufixo Test as classes de teste. Por exemplo, NotaFiscalTest, sabe que ela contém testes automatizados para a classe NotaFiscal (ANICHE, 2015).

A separação do código de teste e código de produção é outra prática comum. Em Java, é normal termos source *folders* separados; em C#, os testes ficam em DLLs diferentes. É também bastante comum que o pacote (ou *namespace*, em C#) da classe de teste seja o mesmo da classe de produção. Isso facilita a busca pelas classes de teste.

Geralmente os testes são baseados em *asserts*. Onde é passando como parâmetros, o valor esperado, ou seja, o valor que quer se obter como verdade ao testar e o valor calculado que é o que vai ser retornado pela classe que foi implementada, da seguinte forma: *assert*(esperado, calculado). O *Framework* de teste vai comparar ambos valores e informar se o teste passou ou não.

Após a execução dos testes é necessária a execução de um processo de depuração para a identificação e correção dos defeitos que originaram essa falha.

2.6 Depuração

A depuração de software é comumente definida como a tarefa de localização e remoção de defeitos. Normalmente, ela é entendida como o corolário do teste bem-sucedido, ou seja, ela ocorre sempre que um defeito é revelado. No entanto, defeitos podem ser revelados em diferentes fases do ciclo de vida do software e a depuração possui características diferentes, dependendo da fase em que se encontra o software. Por isso, os

processos de depuração que ocorrem durante a codificação, depois do teste e durante a manutenção, podem ser diferenciados. A depuração durante a codificação é uma atividade complementar à de codificação. Já a depuração depois do teste é ativada pelo teste bem-sucedido, isto é, aquele que revela a presença de defeitos, podendo beneficiar-se da informação coletada nessa fase. A depuração durante a manutenção, por sua vez, ocorre por uma necessidade de manutenção no software, que pode ter sido causada, por exemplo, por um defeito revelado depois de liberado o software ou pela necessidade de acrescentar novas características a ele (DELAMARO; JINO; MALDONADO, 2013). Como ilustra a Figura 13.

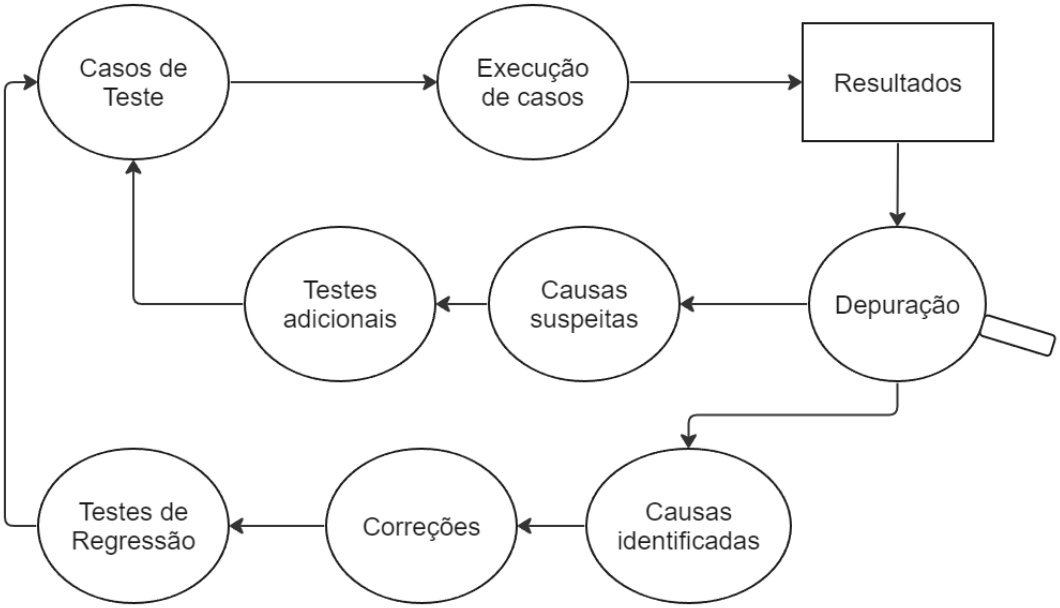


Figura 13 – Processo de Depuração.

Fonte: Autoria própria.

3 Metodologia

A presente seção descreve a metodologia adotada para o desenvolvimento e avaliação do jogo sério denominado GiroJampa. Os pontos abordados nestas seção incluem, características, arquitetura e telas do GiroJampa, bem como a caracterização do objeto de estudo, materiais e métodos usados para elaboração dos testes unitários, e mapeamento dos casos de teste.

3.1 GiroJampa

O jogo GiroJampa é um jogo sério projetado para proporcionar um ambiente de exercício e diversão para pessoas usuárias de cadeira de rodas. Os principais pontos do jogo são os seguintes:

- **Método de Tomada de Decisão**: O jogo utiliza um Sistema Baseado em Regras, onde as ações dos jogadores são definidas por regras lógicas condicionais (if-else). Isso permite que o jogo avalie o desempenho do usuário e forneça *feedback* e recompensas com base nas decisões tomadas (BRAGA et al., 2021).
- **Dados Coletados**: O jogo coleta dados das ações dos jogadores, incluindo distância percorrida, tempo de realização de objetivos e índice de esforço percebido. Esses dados são obtidos por meio de um sistema de sensorização instalado em um ergômetro para cadeira de rodas (SOUTO; SIEBRA; SILVA, 2019).
- **Níveis de Dificuldade**: O jogo possui três níveis de dificuldade: iniciante, intermediário e avançado. Os jogadores precisam atingir metas específicas de velocidade e tempo para progredir de um nível para outro.
- ***Feedback* e Recompensas**: Com base na análise dos dados coletados e nas regras de tomada de decisão, o jogo fornece *feedback* aos jogadores, permitindo-lhes avançar de nível, permanecer no mesmo nível ou retornar a um nível anterior. Além disso, recompensas, como troféus virtuais, são oferecidas aos jogadores com base no desempenho alcançado (RAMOS; MACHADO, 2021).

O nome do jogo “Girojampa” é um acrônimo da palavra “Giro”, lembrando do movimento circular das rodas de uma cadeira de rodas, e da palavra “jampa”, apelido da cidade que inspirou o cenário João Pessoa/PB. Pensando nisso, foi desenvolvido um sistema de regras que deve medir a capacidade física do jogador, a fim de realizar uma seleção do nível de dificuldade baseado nos dados de entrada do jogador (BRAGA et al., 2021).

A arquitetura do jogo foi desenvolvida para dispositivos móveis que usam o *Android* como sistema operacional. Para visualização e interação foi considerado uma visualização em primeira pessoa pelo uso de um *smartphone* acoplado a um óculos RV. A Escala de Percepção de Esforço de Borg foi utilizada para avaliar o esforço percebido por o usuário após a atividade física (CABRAL et al., 2020) e consiste em 15 pontos, graduados de 6 (sem esforço) a 20 (esforço máximo). Sua utilização no SG foi necessária para que o módulo de inteligência identifique o quanto cada tarefa requer fisicamente do usuário. Para coleta de dados, linha sensoriamento, desenvolvido por (SOUTO; SIEBRA; SILVA, 2019), onde um sensor conectado a um ergômetro sem rodas (SOUTO, 2018) capta informações em cada tarefa e as transmite para um dispositivo móvel.

A Figura 14, mostra a arquitetura descrita e suas conexões para funcionamento do GiroJampa.

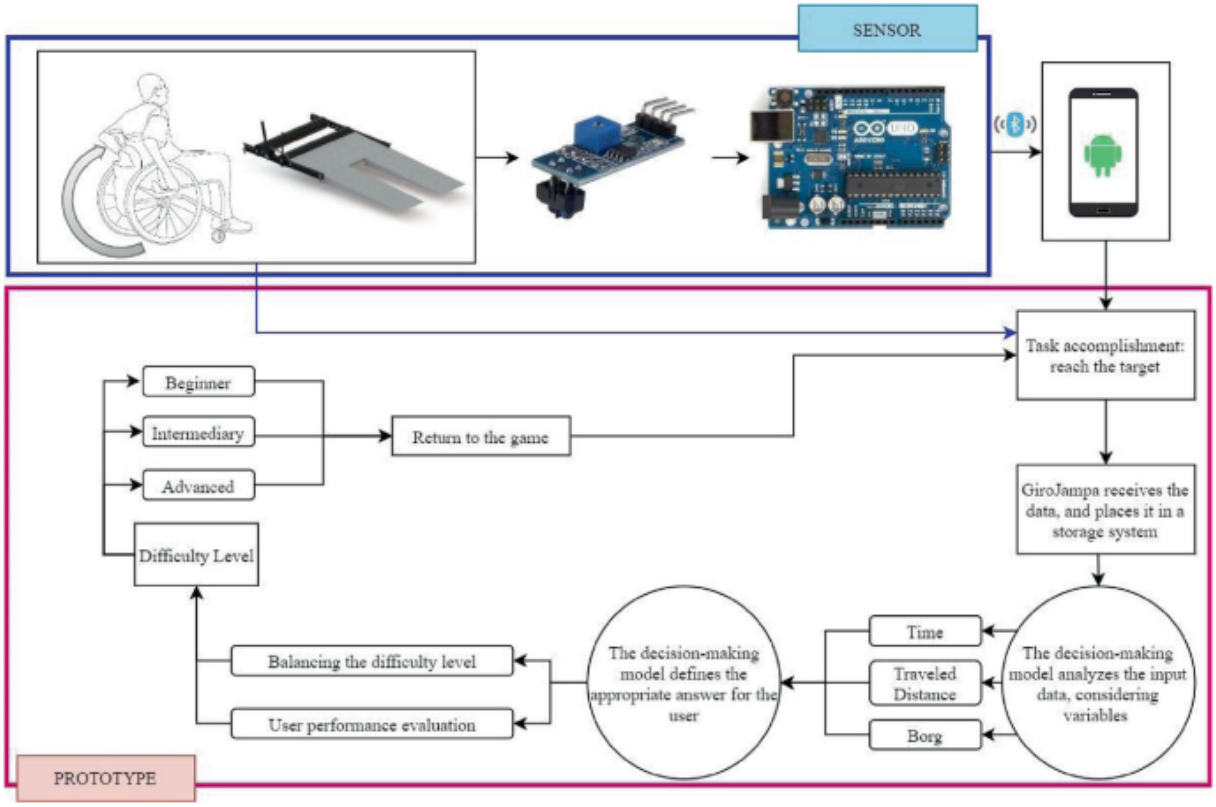


Figura 14 – Arquitetura do Jogo.

Fonte: (BRAGA et al., 2021)

O *Game Design* tem em sua essência, as características necessárias para o jogador sentir-se unido ao contexto apresentado pelo jogo, podendo assim realizar escolhas e tomar decisões pertinentes para o progresso no jogo. Isto é, o *game design* deve oferecer oportunidades para os jogadores, que os levem a realizar decisões que afetarão o resultado do jogo (SATO, 2010).

3.1.1 Adaptação do nível de dificuldade a capacidade física do jogador

Os dados coletados pelos sensores do jogo, alimentam o método de tomada de decisão e a inteligência realiza a gradação da dificuldade das tarefas subsequentes. Para determinação dos níveis de dificuldade foram avaliadas três variáveis: (1) Distância percorrida, (2) Tempo, e (3) Nível de esforço percebido através de uma adaptação da escala de BORG (CABRAL et al., 2020). As duas primeiras varáveis citadas são coletadas através da leitura do sensor proposto por (SOUTO, 2018). A terceira variável Nível de esforço percebido é medida por meio de uma tela interativa, na qual o jogador pode informar a quantidade de esforço físico que realizou para concluir a meta do jogo. De acordo com a resposta fornecida pelo jogador o sistema de regras realizará a verificação se o jogador está apto a avança de de nível, caso apresente nível de esforço baixo, se o jogador permanecerá no mesmo nível, caso aponte nível de esforço moderado, ou se ele voltará um nível se declarar nível de esforço elevado, como mostra a Figura 15.

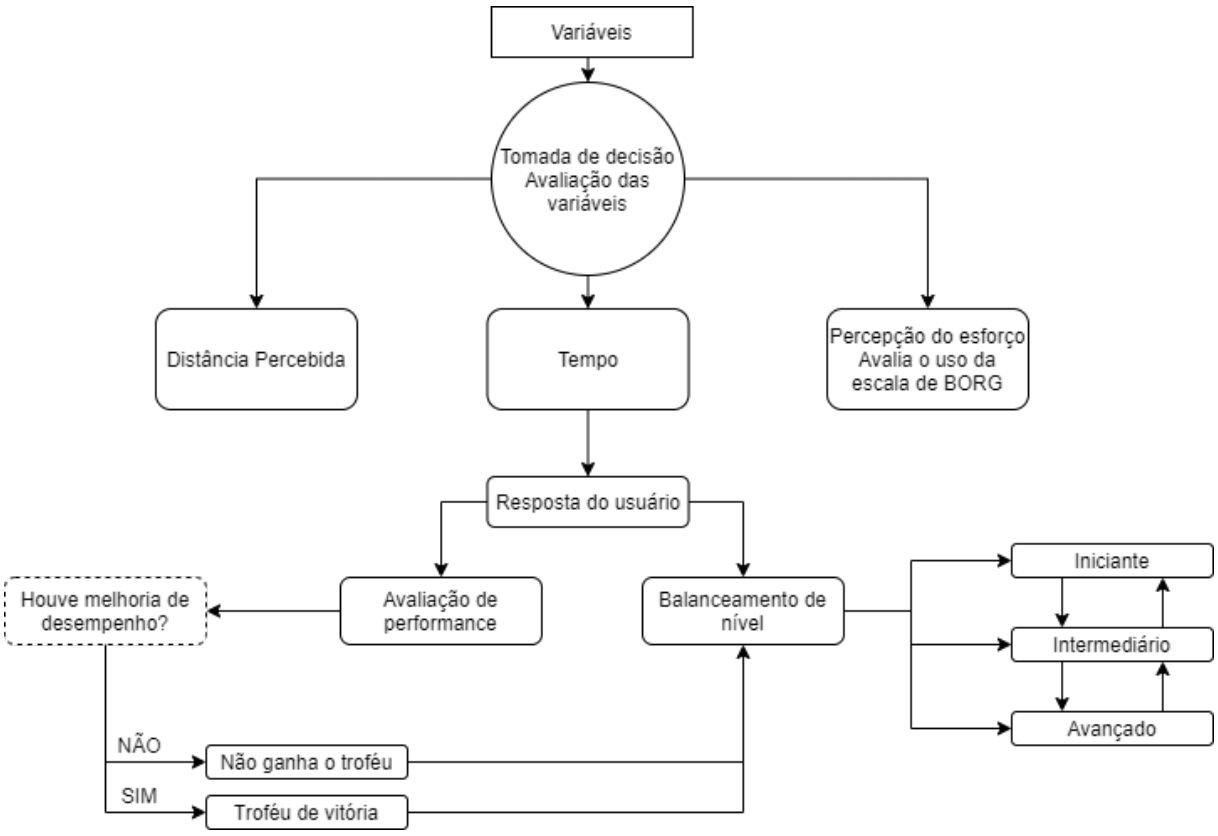


Figura 15 – Variáveis e respostas geradas pelo método de tomada de decisão da inteligência do jogo.

Fonte: (BRAGA et al., 2021)

3.1.2 Telas do Jogo

A implementação do jogo GiroJampa foi realizada na linguagem de programação C#. As cenas exibidas para o jogador contam com mensagens motivacionais para aumentar

a satisfação e engajamento (RAMOS; MACHADO, 2021), com o cenário em si, e com a parte de pontuação. O cenário é uma reconstrução da orla marítima de João Pessoa/PB, contado com pontos turísticos durante o trajeto.

Na Figura 16 é possível observar algumas das telas do jogo, em cima a esquerda tem-se a visão do jogador, ao lado a tela para o usuário informar o esforço percebido (escala de Borg), a baixo e a esquerda um exemplo da tela de calibração do óculos RV, e ao lado dela um exemplo das telas de recompensas, com o troféu de ouro.

Figura 16 – Algumas telas do GiroJampa.

Fonte: Equipe girojampa

Em resumo, o GiroJampa é um jogo sério que combina atividade física e entretenimento para pessoas usuárias de cadeira de rodas. Ele emprega um método de tomada de decisão baseado em regras para avaliar o desempenho dos jogadores, fornecendo *feedback* e recompensas com base em suas ações e alcançando objetivos. Isso cria uma experiência interativa e motivadora para os jogadores, promovendo o exercício e a diversão simultaneamente.

3.2 Caracterização do objeto de estudo

O estudo visou desenvolver testes automatizados para avaliar a eficácia da implementação das funcionalidades de serious games destinados ao condicionamento físico e reabilitação, com foco específico no jogo GiroJampa. A metodologia proposta para a realização da testagem dos componentes do jogo GiroJampa, teve enfoque nos testes de unidade, partindo do pré-suposto que os mesmo já serão suficientes para identificar falhas no jogo.

3.3 Materiais e Métodos

Como mencionado acima, foi utilizada a linguagem de programação C#, juntamente com o *framework NUint* que vem integrado à *Unity*. Este *framework* permite a realização de teste em *playmode* e *editmode*. Para os relatórios de cobertura de código foi utilizado o *Coverlet* que é uma ferramenta popular de cobertura de código para aplicativos .NET que ajuda a identificar lacunas na cobertura de código. Ele funciona com várias estruturas de teste unitário como *NUnit, xUnit e MSTest*. *Coverlet* integra-se bem com sistemas de *build* populares e pipelines de CI/CD e pode ser facilmente integrado ao seu fluxo de trabalho de desenvolvimento. Ele suporta diferentes formatos de saída, incluindo relatórios *XML, JSON e HTML*, permitindo visualizar e analisar os resultados da cobertura de maneira simplificada (HUSSAIN, 2023).

3.3.1 Mapeamento de Casos de Testes

O mapeamento dos testes se deu de acordo com a estrutura organizacional de pastas do jogo, mais especificamente dentro da pasta scripts, que é onde ficam guardadas toda parte de códigos que influenciam no jogo. A principio o GiroJampa possuía a mesma organização mostrada na 17, porém sem a pasta circulada em azul, a pasta **Tests** foi criada para guardar os testes do jogo, com isso a estrutura ficou da seguinte forma.

Figura 17 – Estrutura organizacional do GiroJampa.

Fonte: Autoria própria.

Sabendo desta estrutura, a criação do escopo de cada arquivo de teste se baseou na estrutura das pastas. Sendo assim, foram criados sete arquivos de teste que correspondem

as pastas. Ou seja, o arquivo TestesBorg por exemplo contempla os testes das classes dos arquivos da pasta Borg, o arquivo TestesMensagens os testes das classes dos arquivos da pasta Mensagens, o arquivo TestesPontuacao os testes das classes dos arquivos da pasta Score, o arquivo TestesTrocaCenas os testes das classes dos arquivos da pasta TrocaCena, o arquivo TestesContadores foram divididos em TesteContadoresDistancia e TestesContadoresTempo, pois dentro da pasta Contadores os *scripts* estavam organizados desta forma, estes testam respectivamente as classes dos ContadoresDistancia e ContadoresTempo, por fim o arquivo TestesGerais os testes das classes dos arquivos soltos na pasta *scripts*, vide Figura 18.

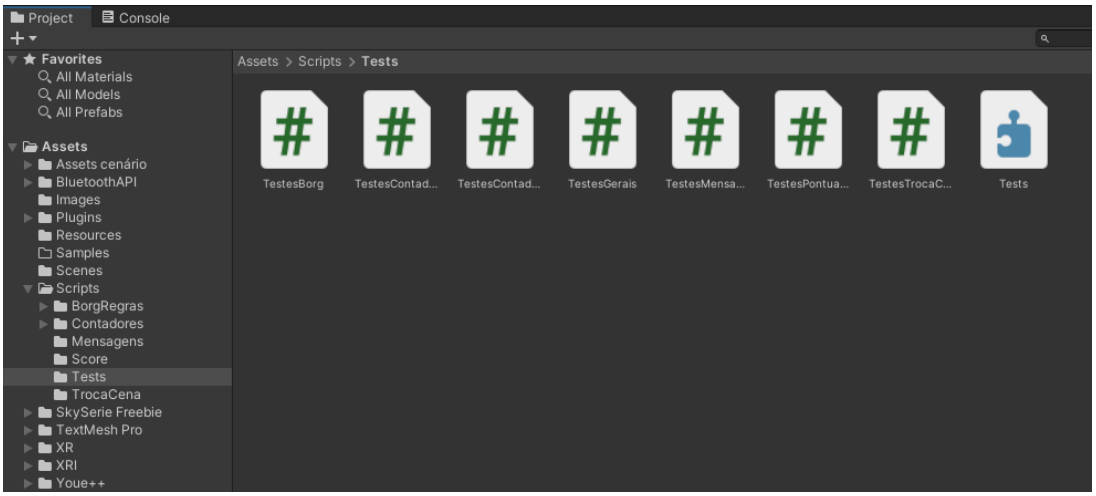


Figura 18 – Estrutura organizacional dos Testes.

Fonte: Autoria própria.

Com isso, o mapeamento dos casos de testes se fez pensando em testar unitariamente cada classe dos *scripts* citados acima, tendo em vista que, conforme mencionado na revisão bibliográfica, cada classe implementa uma funcionalidade do jogo.

3.3.2 Refatoração das classes

Refatoração é o processo de alterar um sistema de software de tal forma que não altere o comportamento externo do código, mas melhore sua estrutura interna (FOWLER; BECK, 1997).

A implementação dos testes unitários para os métodos das classes presentes no código do GiroJampa só foi possível após uma refatoração que isolou a lógica do jogo em métodos que poderiam ser executados pelos *scripts* de teste. A classe `ChegandoLa`, por exemplo, precisou ter a lógica de exibição da mensagem extraída para um método, pois antes estava presente completamente no interior da função `Update`, controlada pela *Unity* e que sua execução para fins de testes unitários não é possível.

Para ilustrar, o trabalho de refatoração realizado no código do jogo. Será utilizada para fins de exemplo, a classe `ChegandoLa`. No *Listing* 3.1, pode-se observar que toda a lógica de verificações da classe `ChegandoLa` está dentro da função `Update`, que é executado a cada quadro um vez. Logo, sendo inviável os testes unitários, fazendo-se necessária a refatoração.

Listing 3.1 – Código do GameObject ChegandoLa, antes da refatoração.

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class ChegandoLa : MonoBehaviour
6 {
7     public GameObject CanvasChegandoLa;
8     private float Countdown_time = 11.0f;
9
10     void Start()
11     {
12         CanvasChegandoLa.SetActive (false);
13     }
14
15     // Update is called once per frame
16     void Update()
17     {
18         if (Contador_DistanciaAvan.distancia >= 800.0)
19         {
20             CanvasChegandoLa.SetActive (true);
21             Countdown_time -= Time.deltaTime;
22
23             if(Countdown_time <= 0.0f)
24             {
25                 CanvasChegandoLa.SetActive (false);
26             }
27         }
28     }
29 }
```

Sendo assim, foram criadas funções testáveis, nesse caso de exemplo da classe como mostra o Listing 3.2. Porém, vale ressaltar que o mesmo foi feito para todos os arquivos que não contavam com classes testáveis, apenas os arquivos testados em TestesBorg, TestesGerais e TestesTrocaCenas não precisaram de refatorações consideráveis, alguns contaram com pequenas alterações. Outro ponto, a ser citado, é que alguns objetos e variáveis precisaram se tornar públicos para serem *mockados* nos testes.

Listing 3.2 – Código do GameObject ChegandoLa, após da refatoração.

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class ChegandoLa : MonoBehaviour
6 {
7     public GameObject CanvasChegandoLa;
8     public float Countdown_time = 11.0f;
9
10     void Start()
11     {
12         CanvasChegandoLa.SetActive (false);
13     }
14
15     // Update is called once per frame
16     void Update()
17     {
18         bool ativarCanvas = ActivateCanvasChegandoLa(
               Contador_DistanciaAvan.distancia);
19         CanvasChegandoLa.SetActive(ativarCanvas);
20     }
21
22     public bool ActivateCanvasChegandoLa(float distanciaAvan)
23     {
24         if (distanciaAvan >= 800.0)
25         {
26             Countdown_time -= Time.deltaTime;
27
28             if (Countdown_time <= 0.0f)
29             {
30                 return false;
31             }
32             return true;
33         }
34         return false;
35     }
36 }
```

3.3.3 Desenvolvimento dos Testes Unitários Automatizados

Os testes foram implementados em `scripts` a parte, instanciando uma classe para cada componente a ser testado e métodos de testes. Para isso foi utilizado o ambiente de desenvolvimento da *Unity* 3D, a IDE Visual Studio, a linguagem de programação C# com auxílio do *framework* de testes *NUnit*.

Foram implementados testes unitários automatizados, capazes de garantir que o código possa receber manutenção e passar por refatorações sem que perca o seu funcionamento base. As seções seguintes abordam as nuances dos resultados obtidos com a implementação dos testes unitários para o GiroJampa.

Para ilustrar como foram feitos os *asserts* de teste, o *Listing* 3.3 apresenta o código parcial de uma classe de testes, nesse casso a de mensagens.

Pode-se observar também em *Listing* 3.3, o uso do chamado *decorator* antes de cada classe de teste (`[Test]`), isso indica que o teste é feito em *editmode*, que quer dizer que são executados apenas no Editor do *Unity* e têm acesso ao código do Editor, além do código do jogo. Este modo de teste, também pode controlar a entrada e saída do modo de reprodução no teste. Os testes realizados para o GiroJampa, foram apenas em *editmode*.

Existe também a possibilidade do teste ser feito em *playmode* que permite testar de forma independente em um *Player* ou dentro do Editor. Os testes do modo de jogo permitem que o jogo seja "jogado" durante a testagem, pois os testes são executados como corrotinas se marcados com o atributo *UnityTest* (Unity Technologies, 2022).

Em *Listing* 3.3, ainda é possível notar que é passado o valor esperado e o calculado na classe que está sendo testada, aplicando o conceito de *Assert*, visto na seção 2.5. O conceito *mocking* abordado na seção 2.4.3, também é utilizado, com o intuito de simular e examinar as interações do jogo, ao se criar dados que forçam a checagem, como por exemplo o `Countdown_time`, que é forçadamente atribuído 0 para testar a segunda condicional (`if`) de *Listing* 3.2, simulando esta condição.

No teste `TestesMensagensVoceConsegue` vide *Listing* 3.3, por exemplo, que é o teste da classe `VoceConsegue` da pasta Mensagens, vide seção 3.3.1. Primeiro é criado o objeto do jogo e atribuído a ele a classe a ser testada (linhas 13 e 14). Então é feito o primeiro *assert*, simulando duas distancias onde a mensagem não deve ser ativada, por isso é *assert* é *IsFalse*, ou seja, se a condição não for verdadeira o teste vai passar (linha 16). Em seguida, é testada a primeira condicional e seu contador, simulando as distancias que ativam a mensagem para um número que satisfaz a condicional e checando se o contador `CountDown_time` é maior igual a zero, pois o *assert False* passa se a condição for falsa (linhas 18 e 19). Por fim, nesse teste é *mockado* o valor `CountDown_time` para forçar entrar na segunda condicional e agora é testado o *True* no *assert*, que por sua vez passa quando a condição é verdadeira (linhas 21, 22, e 23).

Dessa forma, foram feitos todos os outros casos de teste buscando checar todos os casos possíveis de condições do jogo. Evidentemente, as demais classes das outras pastas do jogo possuíam lógicas diferentes, e necessitaram de outros tipos de *asserts*, que correspondessem a suas finalidades.

Listing 3.3 – Código de TestesMensagens.

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using NUnit.Framework;
4 using UnityEngine;
5 using UnityEngine.TestTools;
6
7 public class TestesMensagens
8 {
9     // A Test behaves as an ordinary method
10     [Test]
11     public void TestesMensagensVoceConsegue()
12     {
13         var go = new GameObject();
14         var a = go.AddComponent<VoceConsegue>();
15
16         Assert.IsFalse(a.ActivateCanvasVoceConsegue(99, 99));
17
18         a.ActivateCanvasVoceConsegue(101, 101);
19         Assert.False(a.Countdown_time <= 0);
20
21         a.Countdown_time = 0;
22         a.ActivateCanvasVoceConsegue(101, 101);
23         Assert.True(a.Countdown_time <= 0);
24     }
25     [Test]
26     public void TestesMensagensVoceEstaIndoBem()
27     {
28         var go = new GameObject();
29         var a = go.AddComponent<VoceEstaIndoBem>();
30
31         Assert.IsFalse(a.ActivateCanvasVoceEstaIndoBem(299, 299));
32
33         a.ActivateCanvasVoceEstaIndoBem(301, 301);
34         Assert.False(a.Countdown_time <= 0);
35
36         a.Countdown_time = 0;
37         a.ActivateCanvasVoceEstaIndoBem(301, 301);
38         Assert.True(a.Countdown_time <= 0);
39     }
40 }
```

4 Resultados e Discussões

Na presente seção, são abordadas as maneiras de mensurar os testes unitários, proporcionando uma explanação mais detalhada de conceitos fundamentais, como a cobertura de código. Além disso, buscaremos quantificar os valores de cobertura obtidos durante a execução dos testes no jogo, com o intuito de compreender a significância desses valores para o GiroJampa.

4.1 Resultados Computacionais

Um dos desafios nos testes de software é avaliar a qualidade dos casos de testes (HEMMATI, 2015). Tais medidas de qualidade são chamadas de Critérios de Adequação de teste, abordada na seção 2.4. O tipo de critério de adequação de teste mais usado comumente é a Cobertura de código. Desta forma, é importante diferenciar a Cobertura de código da Cobertura de testes, pois os dois temas são facilmente confundidos:

- **Cobertura de código (*code coverage*)**: Visa medir quanto (%) do software é coberto/exercitado ao executar um determinado conjunto de casos de testes (CANDIDO, 2019). Sendo diretamente afetada por qualquer novo código inserido (BELLODI, 2021);
- **Cobertura de testes (*test coverage*)**: Visa medir a eficácia dos testes perante os requisitos testados, determinando se estes estão cobertos (CANDIDO, 2019). Além disso, a cobertura de testes e sofre mudanças apenas com a adição ou exclusão de testes (BELLODI, 2021).

A conta para descobrir a porcentagem de cobertura de código é dada pela fórmula:

$$\frac{LinhasCobertas}{LinhasDeCódigo} = \%\ de\ cobertura \tag{4.1}$$

Existe um fator a mais a ser considerado nessa fórmula, que é negligenciado muitas vezes, nem todo código é testável, seja por demandar muito esforço ou custo, ou por simplesmente por não representar utilidade funcional, por esse motivo a fórmula adequada para contagem da cobertura seria:

$$\frac{LinhasCobertas}{LinhasDeCódigoTestáveis} = \%\ de\ cobertura \tag{4.2}$$

Com base nas fórmulas apresentadas é possível calcular a cobertura de código do GiroJampa. Entretanto, como o jogo possui uma base de código extensa, separada em

muitos arquivos, essa tarefa se tornaria inviável de ser realizada manualmente. Por isso, como mencionado na seção 3.3 foi utilizada a ferramenta *Coverlet* de cobertura de código que conta com relatórios em HTML que já realizam o cálculo de cobertura e devolvem diversas informações que podem ser utilizadas para melhorar a qualidade do código.

A informação que mais iremos explorar nestes trabalho será a de cobertura de código de quase 30% no GiroJampa (este valor é verificado no campo *Line coverage*, que representa o percentual de linhas cobertas ao executar os testes), onde 606 linhas do jogo estão cobertas por testes (observado no campo *Covered lines*), como visto na figura 19.

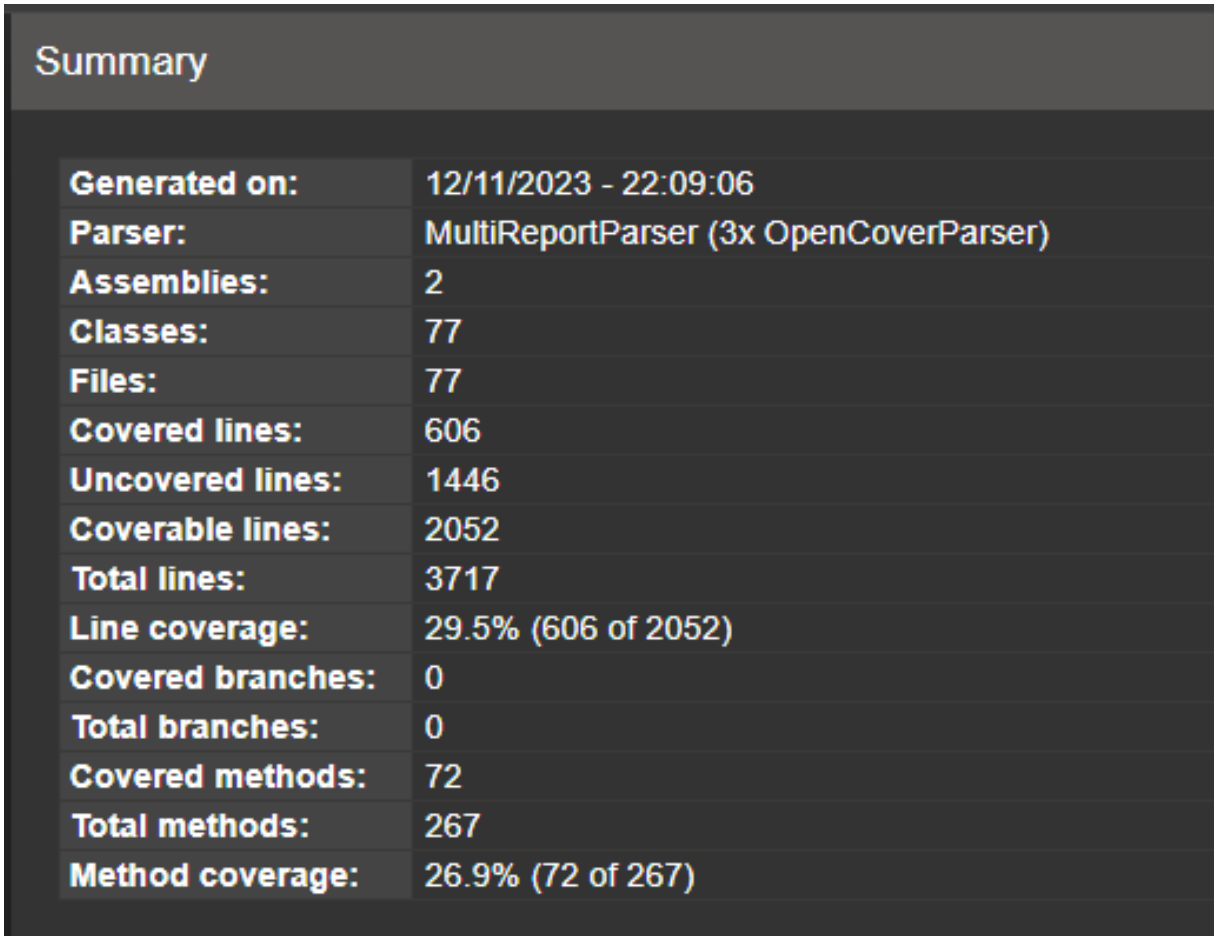


Figura 19 – Captura de tela da ferramenta *Coverlet* contendo em resumo o quanto do projeto os testes implementados cobre.

Dentro dos 30% de cobertura, foram testados os métodos das classes que continham regras de negócio do jogo, ativações de componentes do jogo, verificações, cálculos, definições, e armazenamento de valores.

Esse número de cobertura só foi possível devido a refatoração realizada no código do GiroJampa, para que o mesmo se tornasse testável, assunto anteriormente abordado na seção 3.3.2. O jogo GiroJampa conta com 70 (setenta) arquivos, além disso foram escritos sete arquivos de testes divididos de acordo com a estrutura do jogo, vide seção 3.3.1, totalizando 77 arquivos testáveis.

É importante frisar que existe uma correspondência entre o total de arquivos e o total de classes presentes no jogo. Sendo assim, como existem 77 (setenta e sete) arquivos testáveis, também existe 77 classes. Como visto na Figura 19 nos campos *Classes* e *Files*, o número de Classes e Arquivos, respectivamente, que é 77.

Para ter uma ideia de quanto do jogo precisou ser refatorado, deve-se excluir do cálculos 7 arquivos criados para testes, restando 70 arquivos. Como sabe-se que o número de classes é igual ao número de arquivos, logo, temos 70 classes, e dessas 70 classes 22 (vinte e duas) foram refatoradas.

Assim, a proporção de classes refatoradas, pode ser expressa por:

$$x = \frac{22 * 100}{70} \approx 31,43\% \tag{4.3}$$

Logo, um pouco mais que 31% de todo código do jogo sofreu alguma refatoração. A Tabela 1, evidencia que as classes refatoradas possuíram um percentual mais alto de cobertura de código. Pois, ao refatorar foi possível desacoplar as partes com as lógicas e regras de negócios testáveis, permitindo a simulação de possíveis ações do jogador no GiroJampa.

Tabela 1 – Mapeamento da classes refatoradas no GiroJampa.

Nome da classe	**Foi Refatorada?**	**Cobertura de testes**
AcrediteEmVoce	Sim	61.1%
BLmove	Não	0%
BorgRegraAvan	Sim	66.6%
BorgRegraInt	Sim	66.6%
BorgRegras	Sim	66.6%
BotoesBorgRegras	Não	0%
CameraController	Não	0%
ChamaCenaCalibra	Não	0%
ChamaCenaIniciante	Não	0%
ChamaCenaIntermediaria	Não	0%
Chamaobj1	Não	0%
ChamaObjinter	Não	0%
ChamaObjinter	Não	0%
ChamaObjavan	Não	0%
ChegandoLa	Sim	61.1%
Contador_Distancia	Sim	50%
Contador_DistanciaAvan	Sim	50%
Contador_DistanciaInter	Sim	50%
Contador_Tempo	Não	0%
Contador_TempoAvan	Sim	39.1%
Contador_TempoIni	Sim	39.1%
Contador_TempoInter	Sim	39.1%
ContinuaIniciante	Não	0%
GirarRoda	Não	0%
MenuInicial	Não	0%
MenuInicialChamaOrientacoes	Não	0%
MovEsfera	Não	2.9%
MovEsferaAvan	Não	2.9%
MovEsferaInter	Não	2.9%
Permaneceavan	Não	0%
PersistaEstaProximo	Sim	61.1%
QuaseLa	Sim	61.1%
ResetRotation	Não	0%
RodarRoda	Não	0%
Rotation	Não	0%
SairDoJogo	Não	0%
ScoreDistancia	Sim	55.5%
ScoreTempo	Sim	50%
SelecaoNivel	Não	0%
UmPouco	Sim	61.1%
ValePena	Sim	61.1%
VoceConsegue	Sim	61.1%
VoceEstaIndoBem	Sim	61.1%
VoltaIniciante	Não	0%

Nome da classe	**Foi Refatorada?**	**Cobertura de testes**
VrModeController	Não	0%
VrModeCountdown	Não	1.5%
avancaavan	Não	0%
avancainter	Não	0%
calibrar	Sim	33.8%
camera	Não	0%
chamaOrientacoes	Não	0%
chamaOrientacoesSCRIPT	Não	0%
continuainit	Não	0%
distbronze	Não	0%
distbronzeavan	Não	0%
distprata	Não	0%
imagemfinal	Não	0%
mensagens	Sim	77.5%
menu	Não	56.5%
menuinichamainic	Não	0%
pause	Não	0%
recarregainiciante	Não	0%
regrideavaparainter	Não	0%
trofeuniveliniciante	Não	0%
trofeuouroavan	Não	0%
trofeuourocont	Não	0%
trofeuourodist	Não	0%
trofeuourodistavan	Não	0%
trofeupratanivel	Não	0%
vinteminutos	Sim	69.6%

Total de classes refatoradas:	22/70

Na Figura 20, é possível checar os percentuais de cobertura de cada classe exposto no relatório da ferramenta *Coverlet*. Verificando de forma visual a progressão de cada classe, também pode-se usar o *Coverlet* para comparar por datas e filtrar as classes (campos encontrados na imagem no topo a direita, *Compare with* e *filter*, respectivamente), informação útil para acompanhar a evolução do percentual de cobertura e realizar comparações entre classes, por exemplo.

Dessa forma, a ferramenta *Coverlet* se mostrou eficaz para identificar possíveis melhorias, devido ao seu relatório de fácil leitura e funcionalidades que proporcionam simplificação de análise principalmente em projetos que apresentam uma base de código extensa como GiroJampa.

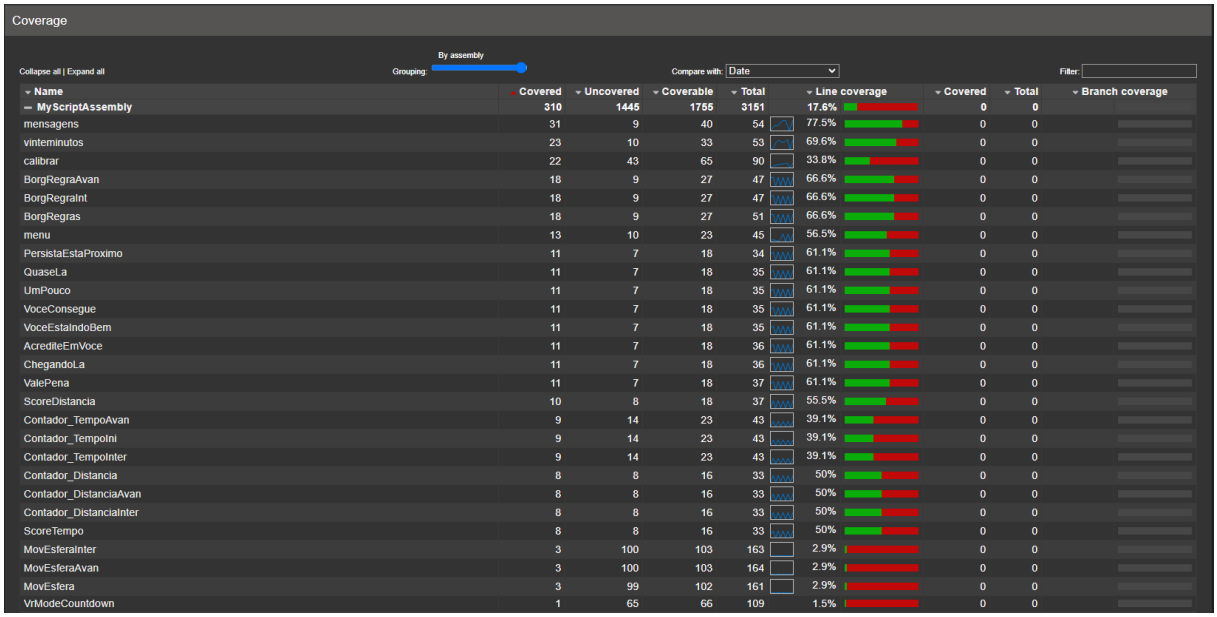
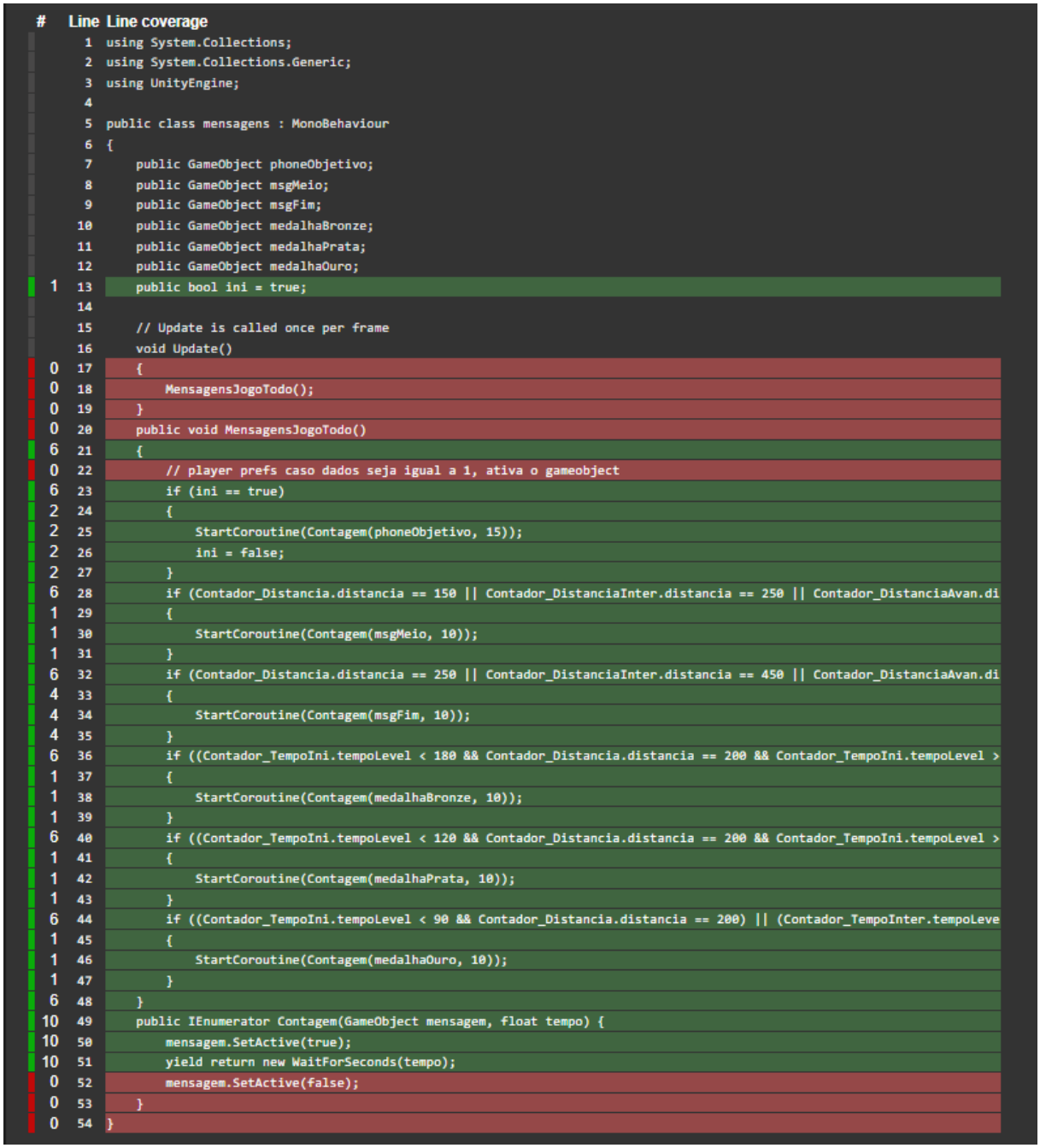


Figura 20 – Captura de tela da ferramenta *Coverlet* com a lista dos arquivos que possuem maior cobertura de testes.

A ferramenta *Coverlet* ainda conta com um relatório detalhado de cada classe, dessa forma, é possível avaliar os trechos de código mais cobertos por testes e os menos cobertos. Como mostra a figura 21, onde a cor **vermelha** indica linhas não cobertas e em **verde** linhas cobertas por testes. Essa indicação visual possibilitou que, alguns casos de teste que a principio não haviam sido pensados pudessem ser revistos e cobertos.



#	Line	Line coverage
	1	using System.Collections;
	2	using System.Collections.Generic;
	3	using UnityEngine;
	4	
	5	public class mensagens : MonoBehaviour
	6	{
	7	public GameObject phoneObjetivo;
	8	public GameObject msgMeio;
	9	public GameObject msgFim;
	10	public GameObject medalhaBronze;
	11	public GameObject medalhaPrata;
	12	public GameObject medalhaOuro;
1	13	public bool ini = true;
	14	
	15	// Update is called once per frame
	16	void Update()
0	17	{
0	18	MensagensJogoTodo();
0	19	}
0	20	public void MensagensJogoTodo()
6	21	{
0	22	// player prefs caso dados seja igual a 1, ativa o gameobject
6	23	if (ini == true)
2	24	{
2	25	StartCoroutine(Contagem(phoneObjetivo, 15));
2	26	ini = false;
2	27	}
6	28	if (Contador_Distancia.distancia == 150 || Contador_DistanciaInter.distancia == 250 || Contador_DistanciaAvan.di
1	29	{
1	30	StartCoroutine(Contagem(msgMeio, 10));
1	31	}
6	32	if (Contador_Distancia.distancia == 250 || Contador_DistanciaInter.distancia == 450 || Contador_DistanciaAvan.di
4	33	{
4	34	StartCoroutine(Contagem(msgFim, 10));
4	35	}
6	36	if ((Contador_TempoIni.tempoLevel < 180 && Contador_Distancia.distancia == 200 && Contador_TempoIni.tempoLevel >
1	37	{
1	38	StartCoroutine(Contagem(medalhaBronze, 10));
1	39	}
6	40	if ((Contador_TempoIni.tempoLevel < 120 && Contador_Distancia.distancia == 200 && Contador_TempoIni.tempoLevel >
1	41	{
1	42	StartCoroutine(Contagem(medalhaPrata, 10));
1	43	}
6	44	if ((Contador_TempoIni.tempoLevel < 90 && Contador_Distancia.distancia == 200) || (Contador_TempoInter.tempoLeve
1	45	{
1	46	StartCoroutine(Contagem(medalhaOuro, 10));
1	47	}
6	48	}
10	49	public IEnumerator Contagem(GameObject mensagem, float tempo) {
10	50	mensagem.SetActive(true);
10	51	yield return new WaitForSeconds(tempo);
0	52	mensagem.SetActive(false);
0	53	}
0	54	}

Figura 21 – Exemplo visual da cobertura de testes de um arquivo referente a uma classe do código.

4.2 Discussões sobre o tema

Diante das informações apresentadas nesta seção de Resultados, foi possível chegar a alguns pontos que trazem discussões relevantes sobre a aplicação dos testes automatizados unitários ao jogo Girojampa.

4.2.1 Cobertura de código de 100% é o ideal?

Um ponto que é discutido na literatura de desenvolvimento de testes, é se existe um valor de cobertura ideal para atestar a qualidade do *software*. Em boa parte dos casos, a ideia dos desenvolvedores está totalmente voltada na mecânica de obter 100% de cobertura nos testes. Entretanto, hoje em dia sabe-se que atingir 100% de cobertura de testes não é garantia de qualidade. Oliveira (2021) afirma que, o importante não é obter 100% de cobertura do código fonte. O importante deve ser cobrir todos os cenários que possuam uma lógica que, se alterada, vai afetar o núcleo do *software* do sistema.

Apesar de poderem ser definidos valores como padrão esperado pelo sistema, por exemplo, uma empresa pode definir 80% como sendo o valor mínimo de cobertura que os desenvolvedores devem atingir ao programar um sistema. Não há garantia que o código será livre de defeito por ter sido atingida uma cobertura de código considerada alta. Isso porque, o desenvolvedor pode testar até métodos e classes sem importância, apenas para aumentar o percentual de cobertura de código. Esse tipo de pensamento pode causar danos aos sistemas.

Com isso, pode-se afirmar que, a cobertura de código pode ser útil, já que ela informa sobre áreas da aplicação que não estão sendo cobertas por nenhum teste assertivo. E isto é um risco e funciona como um convite para investigar essas áreas **não** testadas. Portanto, as porcentagens podem não importar, mas as informações sobre qual código não está sendo executado representam uma heurística útil (ASHBY, 2019).

Sendo assim, é possível concluir que a cobertura de código de quase 30% apresentada no GiroJampa indica que tais partes do código estão mais verificadas e possuem testes que checam algumas de suas condições. Dessa forma, sendo possível identificar pontos de revisão, acompanhar a evolução do software (se o código novo não está afetando o antigo), entender melhor o funcionamento da aplicação, identificar falha, entre outros.

Porém, não é verdade que todas as condições do jogo estão testadas, além de não atingir cobertura total do código, um dos desafios discutidos na literatura para testes no desenvolvimento de jogos diz respeito a incluir algo que simule as ações do jogador. Além de que, como visto na Tabela 1, apenas algumas classes foram atingidas pelos testes. Pois, apenas algumas classes sofreram a refatoração, devido ao grande volume de *scripts* e ao esforço de se codificar testes de um software que teve seu desenvolvimento concluído.

4.2.2 Como o fluxo de desenvolvimento pode afetar o percentual de cobertura de código?

Ciclo de vida típico de desenvolvimento de software ou *Software development life cycle* (SDLC), utilizado no GiroJampa, é uma estratégia de desenvolvimento bastante comum na produção de jogos, como cita (RAMADAN; WIDYANI, 2013). As fases típicas

do SDLC são mostradas na Figura 22.

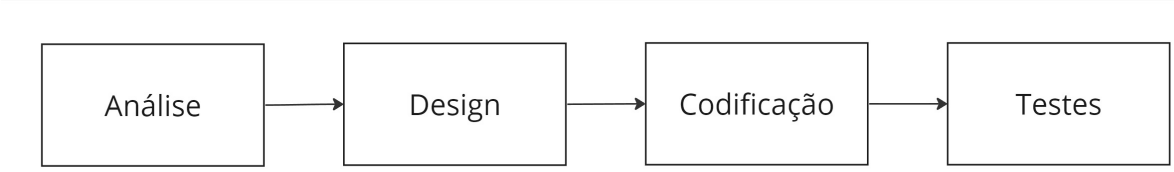


Figura 22 – Ciclo de vida típico de desenvolvimento de software ou *Software development life cycle* (SDLC)

A análise está relacionada à coleta e medição dos requisitos do usuário para criar as especificações de requisitos de software. Na fase de design, esses requisitos são traduzidos em modelos mais detalhados e representação de módulos de software. Durante a geração ou implementação do código, os modelos são traduzidos em código-fonte e aplicativo executável. Finalmente, são realizados testes para garantir que todos os elementos funcionam corretamente e atendem às especificações.

Como pode-se notar nesse modelo, os testes citados são realizados apenas a posteriori da codificação completa, apenas destinados a testar a usabilidade e jogabilidade do jogo, ou seja, testes voltados a checagem de trechos de código, como é o caso de testes unitários, muitas vezes são ignorados durante a produção de jogos, como destaca os autores (MURPHY-HILL; ZIMMERMANN; NAGAPPAN, 2014).

Entretanto, com o avanço dos métodos ágeis e da constante busca pela excelência na qualidade do software, como aborta as seções 2.2 e 2.3, o processo de desenvolvimento deve andar entrelaçado com o de testes, numa espécie de ciclo até uma entrega final. Dessa forma, sendo possível testar o jogo a medida que o mesmo vai evoluindo, testando *features* novas com as já existentes no projeto, até chegar numa entrega e pode-se testar a usabilidade completa do jogo. Como mostrar a Figura 23.

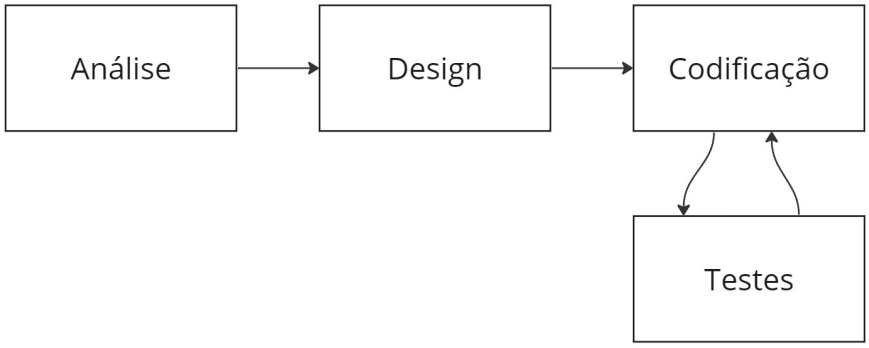


Figura 23 – SDLC modificado

Diferentemente dos testes citados na Figura 22 onde fase de testes contempla apenas os testes de usabilidade e de jogabilidade, na Figura 23 a fase de testes é bem mais ampla,

e contempla tanto a codificação de testes (unitários, de integração de sistema), quanto os testes de usabilidade e jogabilidade.

Porém, o como na implementação do GiroJampa foi adotado o modelo SDLC, para criação dos testes unitários automatizados após o jogo concluído, como mencionado anteriormente foi necessária a refatoração para a elaboração dos testes, ou seja, foi incluída uma nova fase ao ciclo, o que implica em maior tempo e maior esforço até a conclusão do processo.

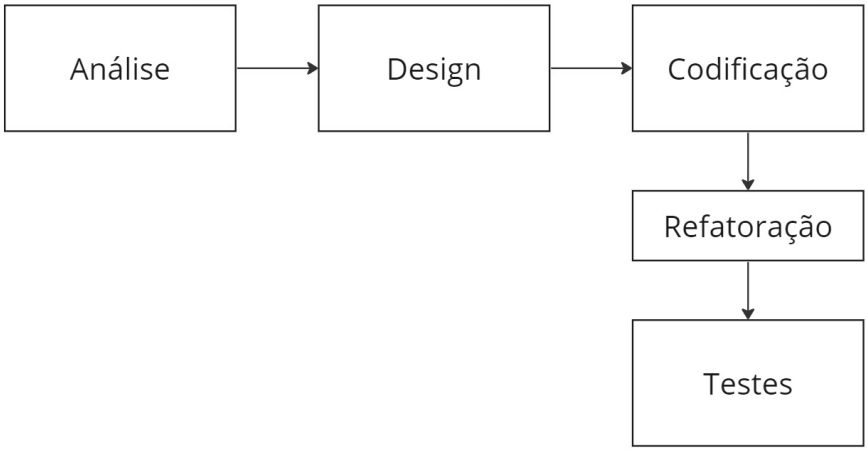


Figura 24 – Fluxo de desenvolvimento do GiroJampa

Por se tratar de um jogo concluído, perceba que na Figura 24 não é necessário realizar o ciclo de volta.

4.2.3 Fatores que impactaram negativamente o percentual de cobertura

Após a realização da refatoração do código do jogo, e da criação dos testes automatizados unitários, onde se conseguiu a cobertura de código de quase 30%. Foi possível notar que, algumas pastas do GiroJampa foram inteiramente testadas, foram elas: Mensagens, Contadores, Pontuação e Borg, que representam a maior parte das pastas. Porém, quatro pontos que impactaram significativamente o não aumento no percentual de cobertura do código do jogo, foram eles: Alguns fatores impactaram significativamente nesse resultado, foram eles:

1. **Os testes unitários serem implementados só após a finalização do jogo**: Pois, como visto na seção 2, um dos pontos apresentados como forma de agilidade é integrar os testes ao desenvolvimento. Integrar os testes ao desenvolvimento das classes garantiria maior cobertura, uma vez que cada classe seria implementada em conjunto com seus testes, vide Seção 4.2.2;

2. **Classes não estarem preparadas para serem testadas**: A implementação das regras de negócio do jogo estava fortemente acoplada a funcionalidades da *Unity*, o que dificultou a implementação dos testes unitários. Dessa forma, foi exigido maior esforço na refatoração e tempo que não foram aplicados exclusivamente na elaboração de testes, vide Seção 3.3.2;

3. **Scripts não testáveis**: a como a maioria dos *scripts* estavam na pasta TrocaCenas ou avulsos, que possuem lógicas entrelaçadas a *Unity*, ou seja, que são acionadas por elementos não testáveis, algumas classes foram inevitavelmente não testadas. A tabela 1, mostra que algumas classes obtiveram o percentual de 0% de cobertura. A falta de documentação sobre como testar cenas também teve impacto direto também nos resultados, pois mais da metade dos testes possuía chamada de cenas. E como não foi realizado um estudo prévio das funcionalidades, não foi possível prevê-lo;

4. ***Coverlet* contabiliza linhas que não são executáveis como não testadas**: a ferramenta detectar linhas que não são executáveis, como mostra a Figura 21. Onde, pode-se observar (linha 11), que a ferramenta *Coverlet* detecta um comentário, informação não executada e o indica em vermelho como linha não coberta. A documentação não explica o porque deste erro.

4.2.4 Pontos positivos da implementação dos testes unitários automatizados

Diante do exposto, a realização de testes automatizados unitários aplicados no jogo GiroJampa, se mostrou útil diante dos seguintes pontos:

- **Agilizando a identificação e correção de problemas**: Pois, ao criar testes automatizados unitários são isoladas as lógicas das classes podendo testa-las individualmente, como mostra a Figura 20 e 21. O que proporcionou uma rapidez para verificar os fluxos do jogo, sem ter que executar o jogo manualmente, onde uma partida dura em média 20 minutos;

- **Rastreando problemas de codificação**: com a criação dos testes automatizados unitários e do auxilio da ferramenta *Coverlet* que forneceu os relatórios de cobertura de código do GiroJampa, foi possível checar os trechos de códigos não cobertos por teste e revê-los e/ou reescreve-los caso não fizessem sentido ou caso houvesse redundâncias;

- **Simulando fluxos reais que podem ser tonados pelo jogador durante o jogo**: Com o uso dos *Asserts* e dos *mocking*, abordado na seção 2.4.3 e visto no exemplo apresentado em, *Listing* 3.3. Dessa forma, simulando o comportamento de um jogador, e através do relatório de cobertura conferindo os fluxos se estavam

cobertos por testes ou não, vide Figura 21, melhorando os testes com base no que não estava coberto e era uma ação possível do jogador;

- **Fornecendo documentação**: Tendo em vista que, os testes de uma forma geral servem como um "documento vivo" da aplicação tendo em vista que qualquer alteração no código deve ser refletida nos seus testes. Para o GiroJampa não foi diferente, os testes automatizados unitários criados descrevem o funcionamento de cada classe, parâmetros aceitos por elas, e retornos esperados, o que facilita o entendimento do sistema de forma mais resumida do que a leitura doe todos os *scripts* que formam o jogo;
- **Aumentando a confiança na manutenção e na entrega de novas versões do jogo**: Uma vez que a equipe de desenvolvimento pode ter uma garantia mais sólida de que as alterações não quebrarão funcionalidades já existentes. Portanto, a adição de novas funcionalidades deveram respeitar os critérios já existentes na aplicação, para que os testes passem.

5 Considerações finais

Os achados deste estudo reforçam os benefícios substanciais derivados da implementação estratégica de testes automatizados unitários no desenvolvimento do jogo GiroJampa, utilizando o *framework NUnit*. A introdução tardia, mas abrangente, desses testes não apenas fortaleceu a estabilidade do sistema, mas também emergiu como uma ferramenta vital na identificação e mitigação eficaz de potenciais problemas. A cobertura de código fornecida pela ferramenta *Coverlet* dos testes proporcionou uma visão crítica sobre os pontos de melhorias de código e de testes do jogo, levantando um ponto de atenção para que mais partes da aplicação futuramente possam ser testadas.

Além disso, a implementação dos testes unitários facilitou refatorações contínuas, possibilitando modificações seguras e propostas de melhorias no fluxo de desenvolvimento para jogos como o GiroJampa. Esse fluxo discutido de codificação e testes andando juntos, não apenas acelera o processo de desenvolvimento, mas também promove uma abordagem adaptativa, permitindo que a equipe de desenvolvimento responda eficazmente às mudanças nos requisitos e às demandas do jogo. Assim, a introdução deliberada dos testes automatizados unitários não é apenas uma prática de garantia de qualidade; ela se traduz em uma vantagem estratégica, que fornece medidas que ajudam no aprimoramento do código, previsão e detecção de problemas.

5.1 Considerações futuras

À medida que olhamos para o futuro, esta pesquisa destaca caminhos promissores para pesquisas subsequentes. Uma análise mais profunda das métricas de cobertura de testes, além da cobertura de código, que foi amplamente discutida nesse trabalho, pode fornecer *insights* específicos sobre áreas críticas do código que beneficiariam de maior atenção e validação. Explorar técnicas avançadas de simulação e a integração de ferramentas automatizadas representam áreas inexploradas que podem enriquecer significativamente as práticas de teste no ambiente *Unity*.

Além disso, ao considerar a evolução futura do GiroJampa, a reflexão sobre a extensibilidade dos testes automatizados unitários é fundamental, pois como visto nos Resultados (seção 4), a aplicação dos testes unitários proporcionaram melhorias significativas para o jogo. A capacidade de escalabilidade dos testes e a otimização do desempenho são fatores determinantes para garantir que o GiroJampa continue a atender aos padrões de qualidade esperados e a evoluir de maneira sustentável em resposta às crescentes demandas e inovações no cenário dos jogos.

Referências

ALÉSSIO, S.; SABADIN, N.; ZANCHETT, P. Processos de software. Santa Catarina: UNIASSELVI, 2017.

AMORIM, J. L. F. Recruta social: a concepção e o desenvolvimento de um serious game para a promoção do controle social. Universidade Federal da Paraíba, 2019.

ANICHE, M. *Testes automatizados de software: Um guia prático*. [S.l.]: Editora Casa do Código, 2015.

ASHBY, D. *Code Coverage vs Test Coverage: Subjectivity and Usefulness*. 2019. Disponível em: <https://danashby.co.uk/2019/02/14/code-coverage-vs-test-coverage/>.

BELLODI, R. C. *Cobertura de Código: Uma Métrica não tão Importante*. 2021. Disponível em: <https://www.linkedin.com/pulse/cobertura-de-c%C3%B3digo-uma-m%C3%A9trica-n%C3%A3o-t%C3%A3o-importante-costa-bell%C3%B3di/>.

BERARD, E. V. *Essays on object-oriented software engineering (vol. 1)*. [S.l.]: Prentice-Hall, Inc., 1993.

BERNARDO, P. C. *Padrões de testes automatizados*. Tese (Doutorado) — Universidade de São Paulo, 2011.

BINDER, R. V. Object-oriented software testing. *Communications of the ACM*, Association for Computing Machinery, Inc., v. 37, n. 9, p. 28–30, 1994.

BOEHM, B. et al. Using the winwin spiral model: a case study. *Computer*, IEEE, v. 31, n. 7, p. 33–44, 1998.

BOEHM, B.; TURNER, R. Using risk to balance agile and plan-driven methods. *Computer*, IEEE, v. 36, n. 6, p. 57–66, 2003.

BRAGA, M. A. et al. Giro jampa: proposição do uso de serious games para reabilitação de pacientes com lesão medular. Universidade Federal da Paraíba, 2021.

CABRAL, L. L. et al. Initial validity and reliability of the portuguese borg rating of perceived exertion 6-20 scale. *Measurement in Physical Education and Exercise Science*, Taylor & Francis, v. 24, n. 2, p. 103–114, 2020.

CANDIDO, A. *Um Pouco sobre Cobertura de Código e Cobertura de Testes*. 2019. Disponível em: <https://medium.com/liferay-engineering-brazil/um-pouco-sobre-cobertura-de-c%C3%B3digo-e-cobertura-de-testes-4fd062e91007>.

COHN, M. *Succeeding with agile: software development using Scrum*. [S.l.]: Pearson Education, 2010.

CRAIG, R. D.; JASKIEL, S. P. *Systematic software testing*. [S.l.]: Artech House, 2002.

CUNHA, S. *Projeto de Teste*. 2010. Disponível em: <https://testwarequality.blogspot.com/p/projeto-de-teste.html>. Acesso em: 12 de Setembro de 2023.

DELAMARO, M.; JINO, M.; MALDONADO, J. *Introdução ao teste de software*. [S.l.]: Elsevier Brasil, 2013.

ENGINEERS, I. of R. Ieee standards. In: INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. [S.l.], 1990.

FERNANDES, R. Engenharia de software. *Curitiba: Fael*, v. 15, p. 16–18, 2017.

FOWLER, M.; BECK, K. Refactoring: Improving the design of existing code. In: *11th European Conference. Jyväskylä, Finland*. [S.l.: s.n.], 1997.

GOKULNATH, B. *Top 9 Benefits of Automation Testing*. 2023. Disponível em: <https://www.hurix.com/advantages-of-automation-testing/>. Acesso em: 10 de Setembro de 2023.

HEMMATI, H. How effective are code coverage criteria? In: IEEE. *2015 IEEE International Conference on Software Quality, Reliability and Security*. [S.l.], 2015. p. 151–156.

HUSSAIN, S. *How to check percentage of code covered in Unit Tests in your Project?* 2023. Disponível em: <https://www.linkedin.com/pulse/how-check-percentage-unit-tests-code-covered-using-coverlet-hussain>. Acesso em: 25 de Setembro de 2023.

JACOBSON, I. *The unified software development process*. [S.l.]: Pearson Education India, 1999.

JAMIL, M. A. et al. Software testing techniques: A literature review. In: IEEE. *2016 6th international conference on information and communication technology for the Muslim world (ICT4M)*. [S.l.], 2016. p. 177–182.

JORDAN, M. *Fundamentos Do Teste De Software - Falha, Defeito E Erro*. 2018. Disponível em: <https://backefront.com.br/fundamentos-teste-software/>. Acesso em: 12 de Setembro de 2023.

KHORIKOV, V. *Unit Testing Principles, Practices, and Patterns*. [S.l.]: Simon and Schuster, 2020.

KRIGER, D. *O que é teste de integração e quais são os tipos de teste?* 2021. Disponível em: <https://kenzie.com.br/blog/teste-de-integracao/>. Acesso em: 20 de Setembro de 2023.

LIMA, D. P. de C.; LEAL, E. D.; MESQUITA, L. J. da S. Benefícios da utilização da pirâmide de teste em uma aplicação mobile. *Revista Interface Tecnológica*, v. 20, n. 1, p. 63–75, 2023.

MATTIOLI, F. E. et al. Uma proposta para o desenvolvimento ágil de ambientes virtuais. *SBC. Anais do WRVA*, 2009.

MURPHY-HILL, E.; ZIMMERMANN, T.; NAGAPPAN, N. 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*. [S.l.: s.n.], 2014. p. 1–11.

NETO, A. Introdução a teste de software. *Engenharia de Software Magazine*, v. 1, p. 22, 2007.

O'BRIEN, H. L.; TOMS, E. G. What is user engagement? a conceptual framework for defining user engagement with technology. *Journal of the American society for Information Science and Technology*, Wiley Online Library, v. 59, n. 6, p. 938–955, 2008.

OLIVEIRA, R. *Cobertura de Testes 100% pode não ser um bom sinal*. 2021. Disponível em: <https://www.linkedin.com/pulse/cobertura-de-testes-100-pode-n%C3%A3o-ser-um-bom-sinal-rodrigo-oliveira/>.

PAIVA, J. *Níveis de Teste, Tipos de Teste e Execução de Testes – Descubra as Diferenças*. 2021. Disponível em: <https://olisipo.pt/blog/niveis-de-teste-tipos-de-teste-e-execucao-de-testes-descubra-as-diferencas/>. Acesso em: 20 de Setembro de 2023.

PERUCCI, C. C.; CAMPOS, F. C. de. Técnicas de qualidade aplicadas em software: um estudo bibliométrico. *Revista de Ciência & Tecnologia*, v. 19, n. 38, p. 5–15, 2016.

PMI. *Success rates rise*. 2017. Disponível em: <https://www.pmi.org/-/media/pmi/documents/public/pdf/learning/thought-leadership/pulse/pulse-of-the-profession-2017.pdf>. Acesso em: 20 de Setembro de 2023.

PRESSMAN, R.; MAXIM, B. Engenharia de software: uma abordagem profissional-tradução: João eduardo nóbrega tortello. *Revisão técnica: Reginaldo Arakaki, Julio Arakaki, Renato Manzan de Andrade.–8 ed. Porto Alegre: Amgh Editora*, 2016.

RAFI, D. M. et al. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: IEEE. *2012 7th International Workshop on Automation of Software Test (AST)*. [S.l.], 2012. p. 36–42.

RAMADAN, R.; WIDYANI, Y. Game development life cycle guidelines. In: IEEE. *2013 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*. [S.l.], 2013. p. 95–100.

RAMOS, R. R. B.; MACHADO, L. S. Elementos de design com foco na satisfação para serious games de reabilitação e condicionamento físico aplicados no jogo girojampa. In: SBC. *Anais Estendidos do XXI Simpósio Brasileiro de Computação Aplicada à Saúde*. [S.l.], 2021. p. 157–162.

RITTERFELD, U.; CODY, M.; VORDERER, P. *Serious games: Mechanisms and effects*. [S.l.]: Routledge, 2009.

ROCHA, A. R. C. da. *Qualidade de software: teoria e prática*. [S.l.]: Prentice Hall, 2001.

SAKUDA, L. O.; FORTIM, I. Ii censo da indústria brasileira de jogos digitais. *Ministério da Cultura: Brasília*, 2018.

SAMBO, J. L. Garantia de qualidade de software. 2018.

SANTOS, L. D. V.; OLIVEIRA, C. V. de S. *Introdução à garantia de qualidade de software*. [S.l.]: Cia do eBook, 2017.

SATO, A. K. O. Game design e prototipagem: conceitos e aplicações ao longo do processo projetual. *Proceedings do SBGames 2010*, p. 74–84, 2010.

SILVA, L. et al. Análise das atividades de teste de software quanto a sua aplicabilidade e importância em empresas privadas. Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte, 2022.

SILVA, R. O. da; ABREU, J. de; AGUIAR, J. J. P. de. Visão introdutória sobre a funcionalidade do teste de softwre no desenvolvimento de sistemas. *TECNOLOGIAS EM PROJEÇÃO*, v. 8, n. 1, p. 51–59, 2017.

SOMMERVILLE, I. Software engineering (ed.). *America: Pearson Education Inc*, 2011.

SOMMERVILLE, I. *Engenharia de software/lan Sommerville; tradução Luiz Claudio Queiroz; revisão técnica Fábio Levy Siqueira.-*. [S.l.]: São Paulo: Pearson Education do Brasil, 2018.

SOUTO, E. *Prática de atividade física para indivíduos com lesão medular atendidos no Sistema Único de Saúde de João Pessoa/PB–Fatores intervenientes e influência na aptidão física e saúde. Londrina, Paraná, Brasil*. Tese (Doutorado) — Tese [Doutorado em Educação Física]-Universidade Estadual de Londrina, 2018.

SOUTO, E.; SIEBRA, C.; SILVA, L. Avaliação de um sistema de captura e visualização de parâmetros físicos em um ergômetro para cadeirantes. In: *XI Congresso Brasileiro de Ati-vidade Motora Monitorada, Alagoas, Brasil*. [S.l.: s.n.], 2019.

SPADINI, D.; ANICHE, M.; BACCHELLI, A. 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*. [S.l.: s.n.], 2018. p. 908–911.

THOMAS, D.; HUNT, A. *The pragmatic programmer*. [S.l.]: Addison-Wesley Professional, 2019.

Unity Technologies. *Edit Mode vs Play Mode Tests*. [S.l.], 2022. Disponível em: <https://docs.unity3d.com/Packages/com.unity.test-framework@2.0/manual/edit-mode-vs-play-mode-tests.html>.

VALENTE, M. T. Engenharia de software moderna. *Princípios e Práticas para Desenvolvimento de Software com Produtividade*, v. 1, p. 24, 2020.