MARCO TÚLIO CARACIOLO FERREIRA ALBUQUERQUE

Gameguts: um framework para ajudar o desenvolvedor a escolher, representar, acessar e apresentar dados DE GAMEPLAY das sessões de jogos

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

Recife
2016

MARCO TÚLIO CARACIOLO FERREIRA ALBUQUERQUE

Gameguts: um framework para ajudar o desenvolvedor a escolher, representar, acessar e apresentar dados DE GAMEPLAY das sessões de jogos

Tese de Doutorado apresentado ao Programa de Pós-Graduação em Ciências da Computação, como parte dos requisitos necessários à obtenção do título de Doutor em Ciências da Computação

Orientador: Geber Lisboa Ramalho

Recife
2016

Catalogação na fonte
Bibliotecário Jefferson Luiz Alves Nazareno CRB 4-1758

A345g Albuquerque, Marco Túlio Caraciolo Ferreira.
GameGuts: um framework para ajudar o desenvolvedor a escolher, representar, acessar e apresentar dados DE GAMEPLAY das sessões de jogos / Marco Túlio Caraciolo Ferreira Albuquerque. – 2016.
125f.: fig., tab.

Orientador: Geber Lisboa Ramalho.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação. Recife, 2016.
Inclui referências e apêndices.

1. Ciência da computação. 2. Design de jogos. 3. Telemetria. I. Ramalho, Geber Lisboa. (orientador) II. Título.

004 CDD (22. ed.) UFPE-MEI 2018-97

Marco Tulio Caraciolo Ferreira Albuquerque

GameGuts: um framework para ajudar o desenvolvedor a escolher, representar, acessar e apresentar dados DE GAMEPLAY das sessões de jogos

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de **Doutor** em Ciência da Computação.

Aprovado em: 16/02/2016.

_____

Prof. Dr. Geber Lisboa Ramalho
Orientador do Trabalho de Tese

BANCA EXAMINADORA

_____

Prof. Dr. Silvio Romero Lemos Meira
Centro de Informática / UFPE

_____

Prof. Dr. André Luiz de Medeiros Santos
Centro de Informática / UFPE

_____

Profa. Dra. Patricia Cabral de Azevedo Restelli Tedesco
Centro de Informática /UFPE

_____

Prof. Dr. André Menezes Marques das Neves
Departamento de Design/UFPE

_____

Prof. Dr. André Maurício Cunha Campos
Departamento de Informática e Matemática Aplicada/UFRN

Dedico esta tese, primeiramente a meus pais que me deram todo o apoio e incentivo possíveis, não só durante este trabalho, mas durante toda a minha vida. À minha esposa pelo companheirismo e ajuda nas horas mais complicadas. Ao Centro de Informática que frequento há quinze anos, por toda a infraestrutura e suporte oferecidos. E, por fim, ao Professor Doutor Geber Lisboa Ramalho cuja valiosa orientação tornou possível a concretização deste trabalho.

AGRADECIMENTOS

Neste momento especial gostaria de agradecer a todos aqueles que ajudaram em minha jornada até esta conquista que será um marco para a minha carreira acadêmica. Agradeço orgulhosamente aos meus pais que me apoiaram em todas as decisões, à minha esposa que é mais do que minha melhor amiga e também a todos os amigos e parentes que não só criaram laços comigo, como também foram importantes para a minha vida. Agradeço também aos meus professores e mestres que tanto me ensinaram e que, ainda hoje, por meio de suas palavras, exercem forte influência no meu jeito de pensar e agir. Dentre todos os docentes do Centro de Informática da Universidade Federal de Pernambuco - CIn-UFPE, gostaria de agradecer ao Professor Doutor Geber Lisboa Ramalho que tem me ajudado e guiado desde a graduação e que por meio de sua orientação, tornou possível a construção deste trabalho. Gostaria de agradecer também ao Professor Doutor Silvio de Barros Melo por ter me orientado durante o mestrado e também em grande parte da graduação.

“Esvazie sua mente, seja amorfo, sem forma como a água, se você coloca água em um copo, ela se torna um copo, se você a coloca em uma garrafa ela se torna uma garrafa, se você a coloca em uma chaleira, ela se torna uma chaleira, a água pode fluir, mas também destruir, seja água meu amigo”.

— Bruce Lee (tradução livre)

RESUMO

Atividades do design de jogos digitais se baseiam, cada vez mais, na análise dos dados relacionados às preferências e ao comportamento dos jogadores. Diversas ferramentas disponíveis para os desenvolvedores de jogos ajudam na aquisição, no registro e na análise de dados genéricos, principalmente relacionados à tríade Aquisição, Retenção e Monetização (ARM). Essas soluções são boas para dar indícios de onde os problemas podem ser encontrados em um jogo. No entanto, para se ter um diagnóstico mais preciso do problema, é necessário trabalhar com dados que reflitam o que aconteceu com o jogador dentro do jogo. No contexto destes dados, que chamamos de dados de gameplay, não existe atualmente um suporte consistente para ajudar os desenvolvedores na decisão de que dados capturar, de como escrever o código de tal captura, escolher a melhor representação desses dados e permitir a sua recuperação e apresentação adequadas. Este trabalho introduz GameGuts (GG), um "framework" focado em dar assistência aos desenvolvedores na hora de escolher, representar, recuperar e apresentar dados de gameplay de jogos. GG contém processos (que se instanciam segundo os tipos de jogos), "guidelines" (para ajudar os desenvolvedores na codificação dos seus projetos) e artefatos que ajudam a representar, guardar e recuperar os dados. Como estudo de caso, usamos o GG para registrar e analisar 500.000 sessões de diversos jogos com jogadores reais, além de dezenas de sessões geradas algoritmicamente para testes exploratórios e de regressão. Os resultados são encorajadores, pois foi possível encontrarmos automaticamente vários "bugs" e trapaças, além de diagnosticar com precisão problemas de design dos jogos, isso tudo com menos esforço do que se tivesse sido feito manualmente.

Palavras-chave: Design de jogos. Analíticas de jogos. Telemetria. Linguagem de domínio específico. Recuperação de informação.

ABSTRACT

Game design activities increasingly rely on the analysis of gamers' behavior and preferences data. Various tools are available for game developers to track and analyse general data concerning the ARM tryad (*acquisition, retention and monetization*) *of game commercialization. These solutions are very good at indicating where problems are in a game, but not to enable a precise diagnosis, which demands fine-grained data. In fact, we need to work with data from that reflects what really happened with the player. To deal with this kind of data, henceforth called gameplay data, there is not enough support or guidance to decide which data to capture, how to write the code to capture it, how to choose the best representation of it and how to allow an adequate retrieval and presentation of it. This work introduces GameGuts (GG), a framework devoted to assisting developers in choosing, representing, accessing and presenting game sessions fine-grained data. GG contains processes (that can be instatiated according to the type of game), guidelines (to help developers code their projects), and artifacts that help developers represent, store and retrieve captured data. As a case study, we have uses GG to record 500.000 sessions of four different games with real players and dozens of algorithmically generated sessions used for exploratory and regression tests. The results obtained are very encouraging since we were able to find several bugs and cheats, as well as to give a precise diagnosis of game design issues. All of this was accomplished faster and cheaper than if done manually or using other frameworks.*

Keywords: Game design. Game analytics. Telemetry. DSL. Information retrieval.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE PSEUDO-CÓDIGOS

LISTA DE SIGLAS E ABREVIATURAS

DAU	Usuários ativos por dia
CIn	Centro de Informática
MAU	Usuários ativos por mês
CTR	Taxa de cliques
GG	GameGuts
GOP	Game Ontology Projects
UFPE	Universidade Federal de Pernambuco
FPS	First-Person Shooter
MOBA	Mutiplayer online Battle Arena

SUMÁRIO

1. INTRODUÇÃO

O presente capítulo apresenta o problema a ser tratado neste estudo, com um breve histórico da popularização dos jogos como serviço e a consequente necessidade de registrar o comportamento dos jogadores dentro dos jogos durante a partida.

1.1 MOTIVAÇÃO E CONTEXTO

Com a popularização da banda larga, a distribuição digital se tornou a maneira mais eficaz de atingir uma vasta audiência de consumidores[1], pois elimina custos de logística e localização, tanto do produto quanto do material de marketing. A distribuição digital também facilita a disseminação de atualizações do jogo (novos níveis, mudança de variáveis, etc). Isso é possível devido à facilidade do envio de novo conteúdo dos desenvolvedores para os jogadores. Mas, o inverso também acontece. Ou seja, possibilita que o desenvolvedor receba informações do jogador facilitando, dessa maneira, o acompanhamento e posterior análise do comportamento do jogador durante uma jogada.

A Zynga foi uma das empresas pioneiras na utilização com sucesso de métricas para análise do comportamento do jogador. Na verdade, a Zynga muitas vezes é considerada uma empresa de análise de grandes bases de dados, e não de jogos, para explicitar essa cultura [2]. Após o grande sucesso comercial da Zynga [3], o modelo *freemium*, ou "*free to play*", no qual pagar é uma opção do jogador após o *download,* virou padrão na indústria de jogos [4]. De fato, com uma olhada nos *rankings* das lojas, podemos ver que esse modelo de receita é utilizado por mais de 90% dos jogos de maior receita nas lojas virtuais para dispositivos móveis. Redes sociais como o Facebook e portais de jogos como o Kongregate permitem somente jogos *freemium*. O sucesso do modelo *freemium* está intimamente ligado à volatilidade dos jogos que podem ser modificados com bastante facilidade. Nesse contexto, jogos são vistos como serviços [5], pois contrariamente aos "produtos" que, uma vez lançados no mercado não sofrem mais modificações, os jogos estão sob constante monitoramento e atualização de acordo com as necessidades, comportamentos e preferências dos jogadores.

Neste paradigma de Jogo como Serviço ou *GaaS* (do inglês, "*Game as a Service"*), entender ao máximo os comportamentos e as preferências de cada jogador com o uso da análise

de dados e métricas (analíticas), é fundamental para corrigir uma gama de problemas, de game design a problemas técnicos, visando a maximização da satisfação do usuário jogador e, consequentemente, o sucesso comercial [6].

É importante notar que o modelo de jogos, como serviço, se caracteriza pelas constantes atualizações e não pelo modelo de receita *freemium* que o tornou tão popular. Alguns jogos, como *World of Warcraft* por exemplo, utilizam o modelo de jogos como serviço atrelado ao modelo de receita de *subscription*, no qual o jogador paga uma mensalidade para jogar.

O foco atual de grande parte da indústria reside, principalmente, nas métricas relacionadas a três indicadores: Aquisição (atração de novos jogadores), Retenção (manutenção dos jogadores engajados) e Monetização (converter os jogadores em pagantes) [7] que, em conjunto, é chamado ciclo ARM. Diversas ferramentas estão disponíveis para ajudar designers e programadores a capturar, guardar e analisar dados sobre o comportamento e preferências dos jogadores. Essas ferramentas são amplamente utilizadas pela indústria de jogos e suprem, de forma decente, as necessidades relacionadas ao ciclo ARM [8].

O problema, normalmente, é o foco excessivo somente na perspectiva comercial de ARM, criando ferramentas padronizadas apenas para captura, representação e recuperação de "dados de engajamento" e "dados demográficos". Dados de engajamento estão relacionados, em geral, ao uso do jogo, como por exemplo: usuários ativos por dia, retenção diária, receita por usuário e custo por instalação. Os dados demográficos mostram segmentação por sexo, idade, país, etc. Em ambos os casos, suportam apenas o diagnóstico da existência de um problema.

Esses tipos de dados fornecem dicas de onde está o problema. Por exemplo: os jogadores desistem com frequência no terceiro dia, mas são muito genéricos e, portanto, os dados são insuficientes para um diagnóstico preciso ou então, os jogadores saem porque completam todo o jogo ou porque estão entediados. Um índice baixo de receita por usuário é tão útil para entender o que está errado no modelo de monetização do jogo quanto uma febre em um diagnóstico médico, por exemplo.

A principal hipótese de trabalho desta tese é que, para o desenvolvedor entender mais profundamente o comportamento e as preferências dos jogadores e compreender o que está por

trás dos dados de engajamento ou demográficos, é preciso incluir na análise os "dados de *gameplay"*, ou seja, aqueles relacionados diretamente a elementos do jogo. Em outras palavras, ir além de dados mais genéricos para mergulhar em dados mais detalhados, que retratem o comportamento e as escolhas do usuário ao jogar. Em um jogo em primeira pessoa, do inglês "First-Person Shooter (FPS), exemplos de dados de *gameplay* são tiros por jogador, colisões entre elementos do jogo, pontos de vida, etc. Só assim, por exemplo, é possível entender porque um grande número de jogadores desistiu no terceiro dia, e, consequentemente, prover recomendações de design [9]. Em casos extremos, é preciso reproduzir uma jogada (*replays* como nos programas esportivos da televisão) para se entender o que aconteceu, o que só é possível usando dados de *gameplay*. A abordagem de *replays* permite inclusive fazer diversas análises no mesmo conjunto de dados e já foi empregada para alinhar objetivos conflitantes de entretenimento e aprendizado em jogos educacionais [10].

Para dar um passo adiante nas métricas, é necessário adquirir e analisar dados de *gameplay* [11] relacionados à jogabilidade do jogo em questão como, por exemplo, quantas balas o jogador utilizou para matar o chefão ou que unidades o jogador criou antes de começar uma batalha ou ainda, quantas vidas o jogador tem em um dado momento do jogo, etc.

Existem diversos *analytics frameworks* (framework para análise de dados) comerciais, mas seu foco principal é na captura, análise e visualização de dados de engajamento, dados de loja e dados demográficos. Apesar de proverem formas de guardar e analisar dados de *gameplay*, esses *analytics frameworks* não dão suporte suficiente para que o desenvolvedor possa decidir quais dados de *gameplay* devem ser capturados, qual a melhor representação para esses dados e como disponibilizá-los de maneira adequada para a recuperação e uso da informação [11]. Alguns *frameworks* acadêmicos [12] se voltam para os dados de *gameplay,* porém com limitações: focam ou em apenas um tipo de jogo (o xadrez) ou em um pequeno subconjunto de dados de *gameplay* (os espaciais).

Escolher os dados de *gameplay* relevantes é uma tarefa complexa, pois depende do propósito da análise, dos *stakeholders* envolvidos, da mecânica do jogo e de diversos outros fatores. Normalmente, a escolha dos dados é feita *ad hoc* e vários aspectos não são devidamente trabalhados, como a granularidade. A falta de processo na captura de dados de *gameplay* também gera problemas no código do jogo, comprometendo, principalmente, a separação de conceitos (do inglês *separation of concerns* [13]). Por fim, a recuperação e o uso dos dados

capturados é essencial para garantir que todos os envolvidos no projeto possam usufruir dos dados.

Em outras palavras, os *analytics frameworks* atuais são extremamente úteis. No entanto, acreditamos que para tirar total vantagem do paradigma de jogos como serviço, desenvolvedores podem alcançar melhores resultados tendo mais ajuda para poder lidar com dados de *gameplay* que têm granularidade mais fina do que os dados de engajamento ou demográficos.

1.2 OBJETIVO E MÉTODO

Diante do exposto, o objetivo deste trabalho é justamente auxiliar o desenvolvedor de jogos digitais na definição, representação, captura, apresentação, armazenamento e análise dos dados de *gameplay*.

Com relação ao estado da arte, não é uma preocupação deste trabalho a maneira como os dados são guardados posteriormente, já que esses aspectos são razoavelmente resolvidos pelas ferramentas disponíveis e trabalhos acadêmicos já publicados [11]. Este trabalho apenas utiliza algumas dessas técnicas e ferramentas para resolver a persistência.

Para atingir tal objetivo, propomos *GameGuts* (ou GG), um *analytics framework* formado por recomendações (*guidelines*), processos e ferramentas, focado em ajudar desenvolvedores a definir, capturar, representar e recuperar dados de sessões de jogos.

GameGuts foi criado como uma ferramenta individual, *stand-alone*, mas que também pode ser acoplada e utilizada junto com ferramentas existentes, complementando-as. GG ajuda os desenvolvedores, desde o início do desenvolvimento, na identificação e captura dos dados de *gameplay*, na sua apresentação a *stakeholders* e como utilizá-los com algoritmos externos de análise.

Primeiro, um processo preciso e direto, baseado em ontologias, para ajudar na escolha de que dados devem ser capturados e também em como esses dados devem ser representados, usando uma linguagem específica de domínio (DSL, do inglês *Domain Specific Language*).

Finalmente, o processo é acompanhado de uma interface de programação de aplicação (API, do inglês *Application Programming Interface*) para recuperar os dados e uma linguagem de recuperação visual (*visual query languague*) para facilitar tal recuperação e o consequente uso e visualização dos dados, principalmente por *stakeholders* não-técnicos como designers e produtores.

Como um estudo de caso, o processo do GameGuts foi aplicado em quatro jogos casuais para web e dispositivos móveis. Em conjunto, foram mais de meio milhão de sessões de mais de cem mil usuários. Os resultados são encorajadores, com economia de centenas de horas de trabalho, pois ajudaram a equipe de desenvolvimento a identificar furos de segurança e também problemas de projeto do jogo.

1.3 ESTRUTURA DA TESE

A estrutura deste documento, após este capítulo introdutório, está organizada da seguinte forma:

O capítulo 2 discute a análise de dados de jogadas, com exemplos das aplicações do registro de partidas e o porquê da necessidade desse registro.

O capítulo 3 focas na aquisição e representação dos dados, mais especificamente, na importância e dificuldade dessas tarefas.

O capítulo 4 mostra as opções disponíveis para o desenvolvedor fazer o registro de suas partidas e poder analisar os dados posteriormente.

O capítulo 5 apresenta o framework resultante deste trabalho: o GameGuts.

O capítulo 6 apresenta os estudos de casos e experimentos para demonstrar a efetividade e potencial do uso de GG em jogos.

Conclusões e trabalhos futuros é um espaço dedicado à apresentação das conclusões deste trabalho, suas contribuições, limitações enfrentadas e perspectiva de trabalhos futuros referentes ao método.

2 - USOS E CATEGORIAS DE DADOS DE JOGOS COMO SERVIÇO

Este capítulo tem como objetivo apresentar os diferentes usos da análise de dados em jogos, evidenciando a necessidade do registro de dados de *gameplay* e também os benefícios da prática.

2.1 INTRODUÇÃO

Ao definir um processo apropriado à captura e análise de dados para qualquer fim ou domínio, é importante entender os objetivos de cada um dos *stakeholders* envolvidos na análise de tais dados. Na indústria de jogos, em particular, são muitos os envolvidos, como será apresentado neste capítulo. Cada um desses *stakeholders* possui interesses particulares e os dados extraídos em um jogo podem ser utilizados para esses interesses.

Este capítulo apresenta os tipos de dados existentes em jogos e discute o uso dos dados sob a ótica ARM, ou seja, como os dados são utilizados na aquisição, retenção e monetização de usuários. Em seguida, discutirá quais outros usos podem ser feitos dos dados, além da abordagem ARM, usos esses muitas vezes negligenciados por falta de uma abordagem unificada de captura e recuperação de dados de *gameplay*.

Para o nosso propósito, os diferentes usos da análise de dados apresentados nesse capítulo são primordiais, pois (1) evita retrabalhar o que já funciona bem nas ferramentas existentes; (2) permite um melhor entendimento dos propósitos do GG; (3) servem de casos de uso para a validação do GG.

2.2 STAKEHOLDERS

A criação de um jogo é um processo genuinamente multidisciplinar. As equipes são extremamente heterogêneas, envolvendo diversos papéis que não são comuns na indústria de software [14]. Entre as categorias de *stakeholders* podemos definir [14]:

- designers: responsáveis por criar uma experiência lúdica para o jogador;
- produtores: responsáveis por gerenciar a equipe que produz o jogo;
- gerenciadores de comunidades: responsáveis por fomentar a comunidade do jogo;
- provedores de dados e análises: responsáveis por prover a infraestrutura necessária para a captura e armazenamento de dados de jogo;
- programadores: responsáveis por codificar o jogo;
- QAs: responsáveis por garantir a qualidade do software criado;
- serviços de terceiros para jogadores: responsáveis por aumentar a experiência dos jogadores com plataformas (ex.: *metacritic* [15]);
- marketing: focados na distribuição e divulgação do jogo;
- entre outros.

Essa pluralidade impacta a análise dos jogos, pois cada um dos *stakeholders* envolvidos no processo se preocupa com um aspecto diferente do jogo. Durante a leitura deste capítulo, é importante atentar para esses diferentes objetivos e visões.

2.3 CATEGORIAS DE DADOS DE JOGOS

Antes de discutir os dados relevantes segundo as necessidades de cada *stakeholder*, é importante deixar mais claras as categorias de dados envolvidas na análise de comportamento e de preferências do jogador.

Não existe na literatura um consenso sobre as categorias de dados de jogos [11]. Baseado em mais de 10 anos de experiência pessoal no mercado de jogos, sendo sócio e gerente de projetos de uma empresa do ramo que faz utilização sistemática de *analytics*, entendo que existem cinco principais categorias de dados, que serão detalhados nesta seção:

- dados demográficos;
- dados de loja;

- dados de engajamento;
- dados de software; e
- dados de gameplay.

Os dados demográficos são relacionados aos jogadores. Definem que pessoas estão jogando o jogo em termos de sexo, idade, país, língua, localização, entre outros. Em geral, esses dados são segmentados para o desenvolvedor entender em termos gerais o jogador. Por exemplo, quem joga são mulheres brasileiras com mais de 50 anos de idade.

Já os dados de loja incluem quantidade de downloads, receita gerada em relação a outros jogos e aplicativos nas lojas (físicas e digitais), posicionamento em seções de destaque nas lojas, entre outros. Eles indicam ao desenvolvedor como seu jogo se posiciona no mercado em geral.

Jogos podem ser analisados como um artefato digital de acordo com sua frequência de uso e o comportamento de uso de seus usuários. Quantos usuários jogam todo dia? Ou todo mês? Quanto tempo esses usuários passam no jogo? Quanto cada jogador gasta em média? Qual a fração dos jogadores que experimentaram o conteúdo do terceiro estágio? Nós chamamos estes dados de *dados de engajamento*, pois indicam quão engajados os jogadores estão com o jogo: só fez o download, jogou por poucos dias, jogou por vários dias, fez compras, etc. São estes dados os relacionados com o ciclo ARM (aquisição, recuperação e monetização).

Entendemos que em parte da literatura de jogos, em particular na que lida com ajuste de dificuldade, o conceito de engajamento é utilizado em outra perspectiva: o quão o jogador está envolvido no jogo ao jogar [16][17]. No entanto, por falta de uma denominação mais apropriada, utilizaremos este termo no contexto desta tese.

Um jogo é um software e, portanto, inclui diversos elementos não relacionados à jogabilidade em si. Dados que indicam integração com outros sistemas como redes sociais, (por exemplo, quantos usuários estão fazendo uso da sinergia com o Facebook), falhas de software (*bugs*), tempo de *uptime*, entre outras coisas, são classificados como *dados de software* e não fornecem indicações ligadas diretamente à jogabilidade e elementos de jogo.

Por fim, há os *dados de gameplay,* que são relacionados diretamente com o que o jogador faz *dentro do jogo*, com sua relação com os elementos dos jogos. Alguns exemplos desses dados são: quantos tiros o jogador dispara por estágio de um *FPS*, quais feitiços o

jogador usou mais em um certo personagem de um *MOBA*, em que ordem os "prédios" foram construídos em um *RTS*, quanto pontos de vida tinha o jogador e que golpe recebeu para morrer em um jogo de luta, etc. Vale salientar que, enquanto os outros dados (de loja, demográficos, de engajamento e de software) são genéricos, mudando muito pouco ou nada segundo o jogo em questão, os dados de *gameplay* dependem das especificidades de cada jogo.

Importante notar que os tipos diferentes de dados podem se combinar para entregar mais valor aos *stakeholders*. É comum, por exemplo, segmentar dados de engajamento por dados demográficos: qual o gasto médio de jogadores (engajamento) homens com idade entre 24 e 30 anos (demográfico). Nas próximas seções, serão apresentados usos dos dados para análises e também como podem ser impactados pela captura deficiente.

2.4 USO DE DADOS NO CLICO ARM

Nas próximas seções, vamos detalhar dados referentes ao ciclo ARM que são, na sua maior parte, dados de engajamento. Damos ênfase a esses dados agora porque são os mais usados atualmente na indústria, assim como os que tem melhor suporte de ferramentas.

2.4.1 Uso de dados na aquisição de usuários

A aquisição de usuários é a prática utilizada pelos desenvolvedores para que mais jogadores joguem o jogo pela primeira vez. Gerentes de marketing, por exemplo, são especialmente preocupados com a aquisição e principalmente com o custo. A viabilidade comercial de jogos, como serviço, é medida pela relação de custo para aquisição de um usuário contra a receita gerada por ele. E o marketing é responsável por manter esse custo abaixo da receita gerada, ou melhor, o mais baixo possível.

A aquisição normalmente se preocupa com (1) as peças de publicidade usadas, (2) a descrição do jogo nas lojas e (3) a experiência inicial do usuário.

O custo por usuário direto é medido em termos de performance das peças publicitárias, como vídeos ou imagens promocionais. Cada um desses artefatos, chamados *creatives*, é testado com diversas opções de segmentação. Os dados capturados com maior frequência são, em geral, demográficos ou de engajamento e tentam responder perguntas como, por exemplo:

- Qual a imagem com mais cliques?
- Qual o vídeo mais assistido?
- Qual a imagem que gera mais downloads?
- Qual o vídeo com melhor performance entre homens de 25 a 34 anos?

A utilização de métricas objetivas facilita a criação de novas peças de publicidade, principalmente com a permutação de imagens e chamadas de texto, como na Figura 1. Cada uma das imagens tem sua performance medida e, aquela com melhores índices, será usada com mais frequência (maior investimento).

Figura 1 - Exemplos de imagens diferentes com a mesma chamada de texto

Fonte: Manifesto Games LTDA

Na aquisição, a composição de dados de engajamento e demográficos é essencial. É importante entender o perfil das pessoas mais impactadas por cada uma das peças. Uma peça pode ter uma performance baixa, quando utilizada com mulheres brasileiras de até vinte anos e uma performance muito boa quando utilizada com homens chineses acima dos cinquenta anos. As redes de publicidade digital, como o Google *Admob*[18] e *Chartboost*[19], proveem relatórios com esse tipo de dado e ferramentas para definição de campanhas com essa especificidade.

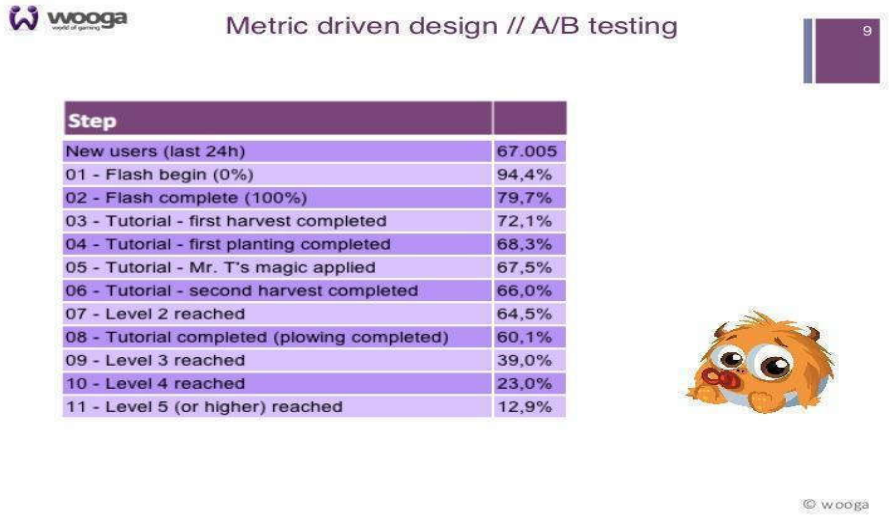
Essa medição também ajuda a entender o público do jogo em si. Dessa maneira, os designers possuem conhecimento de que pessoas estão jogando o jogo e podem fazer ajustes mais objetivos.

Em geral, ao trabalhar a aquisição, é comum utilizar dados de loja como posição no *ranking* e avaliações que impactam na facilidade de encontrar o jogo, assim como a descrição do aplicativo e os comentários dos jogadores. Novamente, ferramentas populares como *App Annie*[20] e *Appfigure*[21] proporcionam um suporte padronizado e de baixo custo para os desenvolvedores que podem acompanhar a performance do jogo nos diversos países em que foi lançado.

A descrição é utilizada nos algoritmos de indexação das lojas e tem um papel importante nas buscas. Ferramentas que simulam buscas e mostram a eficácia da descrição para cada termo estão disponíveis - em que lugar meu jogo aparece quando faço uma busca por “efeitos espetaculares” - são amplamente utilizadas para otimizações.

A experiência inicial do usuário, normalmente um tutorial mostrando como jogar o jogo, também é medida como parte da aquisição - considera-se que o jogador ainda não teve uma experiência de jogo. Normalmente, os diversos passos do tutorial são medidos em formato de funil para saber qual percentil dos usuários iniciais terminou cada etapa. Na Figura 2, uma imagem de como a desenvolvedora alemã *Wooga* mede os passos do jogo Monster World. Esses dados são considerados de engajamento pois medem, em termos gerais, se o jogador efetivamente se adaptou ao jogo ou não, sem trabalhar elementos diretos do jogo

Figura 2 - Taxa de usuários em cada uma das fases iniciais do Monster World.



Step	
New users (last 24h)	67.005
01 - Flash begin (0%)	94,4%
02 - Flash complete (100%)	79,7%
03 - Tutorial - first harvest completed	72,1%
04 - Tutorial - first planting completed	68,3%
05 - Tutorial - Mr. T's magic applied	67,5%
06 - Tutorial - second harvest completed	66,0%
07 - Level 2 reached	64,5%
08 - Tutorial completed (plowing completed)	60,1%
09 - Level 3 reached	39,0%
10 - Level 4 reached	23,0%
11 - Level 5 (or higher) reached	12,9%

Fonte: http://www.slideshare.net/wooga/2011-0720casualconnectedited

Por fim, alguns elementos do próprio jogo influenciam na aquisição, mas não são exatamente dados como, por exemplo, a disponibilidade do jogo (o jogo está disponível para Windows? E Linux? E para dispositivos móveis? Etc). A utilização de marcas conhecidas, como o personagem Mickey, por exemplo, também tem impacto claro, apesar de intangível, na aquisição de usuários.

2.4.2 Uso de dados na retenção de usuários

A retenção de usuários é a parte da operação de um jogo como serviço preocupada em evitar que o jogador pare de jogar. A retenção é, portanto, o principal indicador a ser monitorado e ajustado e de interesse particular dos *designers* envolvidos. Criar uma experiência ótima para o usuário é o caminho usado por game designers para tornar o jogo o mais divertido e viciante possível.

Normalmente, a retenção é medida com dados de engajamento como: (1) percentil dos jogadores que voltam ao jogo após um certo número de dias (retenção após 1 dia, após 7 dias, etc.), (2) tempo de sessão por jogador, (3) quantidade de sessões por jogador, (4) usuários ativos por dia, (5) usuários ativos por mês, entre outras.

Cada um desses dados de engajamento indica sintomas superficiais de problemas. O designer precisa usar sua experiência para o diagnóstico correto e, principalmente, para remediar a situação. Em jogos com retenção baixa no sétimo dia, no qual a maioria dos usuários abandona o jogo após uma semana, o designer lança mão de medidas padrão da indústria, tais como mais conteúdo (novos níveis), ou aumentar seu *replay value* criando recompensas externas ao jogo (como *daily rewards*, amplamente utilizado em jogos para dispositivos móveis) para tentar atacar o problema. No entanto, essas medidas são arbitrárias e baseadas em padrão da indústria. A real motivação da saída dos usuários não é indicada e tais soluções nem sempre são relevantes [22].

Dessa maneira, designers precisam de dados de retenção relacionados ao *gameplay* que indiquem problemas na experiência do usuário e/ou na mecânica de jogo. Saber um jogador sai no terceiro dia de jogo é menos relevante do que entender que ele sai no terceiro dia por que o jogo é muito fácil, ou muito difícil ou frustrante ou chato, etc. Nesses casos, a métrica objetiva (como quantos dias, em média, um jogador joga o jogo), deve ser trocada por métricas mais subjetivas, (como o jogador está frustrado com a experiência, se ele se divertiu, se o nível de dificuldade é adequado), que influenciam na retenção. Dados de *gameplay* podem ser utilizados para medir a experiência do jogador como um todo, com relação a sentimentos como frustração, desafio, entre outros [3].

O funil de conteúdo é normalmente medido por índices que indicam a parcela de usuários que atingiu aquele conteúdo - dados de *gameplay*. Esse índice deve ser monitorado para (1) identificar pontos com altas taxa de *dropouts* (jogadores abandonam um jogo ao atingirem certo conteúdo) ou (2) medir a parcela de usuários que já acessou a maior parte dos conteúdos indicando a necessidade de conteúdos novos. No entanto, comumente é usado de forma superficial e pode indicar problemas (*dropouts* altos ou falta de conteúdo) ou até pontos positivos (baixíssima taxa de *dropout* vinculada a alto *replay value* de um certo conteúdo, por exemplo), embora não indiquem que parte da experiência daquele conteúdo o faz ser bom ou ruim.

Quando os jogadores abandonam o jogo por ter completado todo o conteúdo, o funil pode indicar corretamente o problema. Mas, indicará, nesse caso, a necessidade de criação rápida de conteúdo. É necessário criar níveis suficientes para que o jogador jogue o jogo mais vezes e em uma velocidade maior do que o jogador pode consumir. No entanto, a criação não

pode ser feita de qualquer maneira. Dados de *gameplay* podem ajudar na criação automática de níveis (mais rápida, menos custos), que utiliza o comportamento de milhares de jogadores nos níveis [23] e a percepção de dificuldades [24] para garantir a qualidade. Ou seja, os dados de *gameplay* são essenciais para garantir que a métrica de retenção seja melhorada.

Em alguns jogos, como em *PinUp Heroines* [25] da Figura 3, os níveis são na verdade os oponentes do jogador. Nesses casos é comum o game designer focar na criação de experiência de luta entre os jogadores, para que disputem entre si aumentando o *replay value*, em teoria, infinitamente. Dessa maneira, não é necessária a criação de diversos níveis de dificuldades para se adequar a cada jogador. No entanto, com dados de *gameplay* é possível escolher entre níveis de dificuldade pré-determinados [16] ou utilizar aprendizagem de máquina na criação online de personagens, baseada na avaliação de emoções [3]. Criar personagens automáticos ajuda tanto no (1) aumento do funil com cada novo personagem (2), bem como no aumento do *replay value* de cada um dos oponentes, que aprendem com o jogador e podem propiciar um nível de desafio adequado [17].

Figura 3 Tela do jogo Pinup Heroines

Fonte: Manifesto Games LTDA

Como mostrado nesta seção, os problemas de retenção podem ser melhor identificados e tratados com o uso de dados de *gameplay*. Além disso, a retenção é também um problema crucial pois está intimamente ligada à monetização, ou seja, ao sucesso comercial do jogo (discutido na próxima seção). O Quadro 1 representa os números do portal Kongregate.

Quadro 1- Monetização de acordo com a quantidade de sessões por usuário no portal de jogos Kongregate

Top 10 Jogos Multijogadores	% Compradores	Receita por usuário pagante	Média por transação (compra)	Quantidade média de transações (compras)	% de jogadores	% da receita total do jogo
1 jogada	0,03%	$ 6,98	$ 5,02	1,39	45,5%	0,1%
2 a 10	0,40%	$ 11,01	$ 6,63	1,66	40,3%	0,9%
11 a 50	4,93%	$ 19,82	$ 7,92	2,50	7,7%	3,9%
51 a 100	11,14%	$ 33,14	$ 8,97	3,70	2,3%	4,3%
101 a 250	17,11%	$ 63,12	$ 11,11	5,68	2,2%	12,0%
251 a 500	26,94%	$ 123,92	$ 14,09	8,80	1,0%	16,9%
500+	39,39%	$ 270,58	$ 19,03	14,22	1,1%	62,0%
Total	1,89%	$ 102,74	$ 15,03	6,84	100,0%	100,0%

Fonte: http://www.slideshare.net/Kongregate/f2p-design-crash-course-casual-connect-kyiv-2013-27819236

2.4.3 Uso de dados na monetização de usuários

Monetização é o nome utilizado para as práticas que os desenvolvedores utilizam na criação de receita a partir do jogo. Em geral, a análise da monetização se preocupa em entender os comportamentos de compra ao (1) analisar os perfis dos compradores (para quem vender), (2) entender as necessidades de compra (o que vender?) e (3) a disposição para gastar (por quanto compra?).

Ao analisar os jogadores, com dados de engajamento e demográficos, a conclusão da indústria é que a grande maioria da receita é gerada por alguns poucos usuários[26] denominados *whales* (baleias). As baleias representam, em média, menos de 1% do público e mais de 40% da receita [27]. Não se sabe ao certo se esse comportamento é realmente inerente ao modelo de receitas *freemium* ou somente da maneira como os dados são capturados para trabalhar a monetização. Ou seja, uma miopia provocada pelo uso de poucos tipos de dados.

Além disso, coloca o modelo de receita em um dilema ético amplamente divulgado pela mídia e que denigre a imagem dos desenvolvedores [28].

Essa mesma miopia pode afetar também a análise dos itens. Dados de engajamento como que item vende mais, são muito comuns, e alguns poucos itens são, em geral, responsáveis pela maior parte da receita. No entanto, também não é fácil concluir se é um problema inerente ao modelo de receita *freemium* ou àmiopia dos dados.

É importante entender o que vende mais e por qual valor. Mas também, é necessário saber o porquê. Em que contexto um item é comprado? Que aspectos do jogo um usuário pagante vivencia e que não está sendo vivenciado por não pagantes?

Figura 4 - Vários aspectos da monetização representam menos de 15% da receita enquanto que a Magic Wand tem 35%.

Fonte: http://www.slideshare.net/wooga/2011-0720casualconnectedited

2.5 USO DOS DADOS PARA OUTROS FINS

Além do ciclo ARM, outros fatores estão ligados ao desenvolvimento de um jogo. Esta seção discute três principais fatores. O primeiro deles é um dos melhores efeitos colaterais da captura correta de dados de *gameplay* para registro de jogadas, a possibilidade de reprise de partidas possibilitando, entre outras coisas, novas análises a partir do mesmo conjunto de dados. Outros dois aspectos discutidos estão ligados ao desenvolvimento de software em geral, sendo eles (1) bugs e (2) falhas de segurança. Cada um desses fatores é discutido com mais detalhes nas próximas seções.

2.5.1 Uso dos registros para reprise de jogadas

Um dos usos dos dados de *gameplay* é a possibilidade de visualização da jogada. Esse ponto é trado de maneira específica pois, em uma análise feita por humanos, a possibilidade de poder assistir à jogada facilita o entendimento do contexto e, consequentemente, o diagnóstico do problema.

Em jogos competitivos, a visualização de partidas é de grande valia para os jogadores. Assistir partidas de jogadores mais experientes, como em um esporte, acelera o processo de aprendizado. Nesse caso específico, é claro que o uso de vídeos também pode ser utilizado. No entanto, o uso de vídeos impede a paralisação da partida a qualquer momento para que um jogador em treinamento continue a partir daquele momento para tentar outras possibilidades, como é comum em partidas de xadrez, por exemplo.

A análise de partidas por algoritmos melhora a qualidade do planejamento das ações dos personagens virtuais, melhorando assim a qualidade do jogo [29]. Algoritmos também são capazes de classificar estratégias e indicar a performance de cada uma delas[30]. Esse registro precisa ser feito automaticamente, o registro manual mostra-se não confiável [31] para a utilização nesse tipo de tarefa.

A visualização de jogadas também facilita no entendimento de problemas de design de jogo dos desenvolvedores. Um problema muito comum é a identificação de estratégias dominantes, no qual o jogador pode seguir um certo conjunto de comandos e sempre se sair vencedor. A partir da reprise de uma jogada, na qual tal estratégia foi usada, é possível entender

como modificar o jogo para eliminá-la. Esse registro pode, inclusive, ser reprisado quantas vezes forem necessárias a cada mudança no jogo, para garantir que a estratégia dominante foi eliminada.

Em diversos jogos, os desenvolvedores tornam os dados públicos ou possíveis de extrair pelo jogador. Dessa maneira, os próprios jogadores podem utilizar os dados para alterar a experiência de jogo [32]. Na verdade, existem várias comunidades virtuais focadas em analisar registros de jogos e ferramentas apropriadas para isso e estão amplamente disponíveis na internet, como a *World of Logs*[33] utilizadas por jogadores do jogo *World of Warcraft*. No entanto, por falta de padronização e ferramentas para captura de dados de *gameplay*, essa prática não é adotada na maioria dos jogos.

Testes em grupos focados para avaliar a qualidade da experiência do usuário e a usabilidade das entradas do jogo também podem ser imensamente facilitados e barateados com o uso de reprises. Os testes para a usabilidade na localização de jogos, por exemplo, precisam que o teste seja feito em diferentes línguas por usuários nativos daquela língua e que, em muitos casos, são de difícil acesso. Fazer o teste remotamente, com o uso de reprises, evitaria a necessidade de uma infraestrutura adicional em diversos países para o teste. Além disso, a automação do processo possibilita o teste com centenas, milhares de usuários, diminuindo o viés do grupo de teste [34].

Por fim, a reprise pode ser usada para geração de dados que não foram inicialmente registrados, em casos no qual um certo tipo de análise não foi originalmente pensado e, consequentemente, os dados necessários para tal não foram registrados. Fazer a reprise da jogada é o suficiente para gerar esses dados. Uma outra possibilidade é a utilização de simulações simplificadas do jogo (não necessariamente no próprio jogo) para testes diversos, como mostraremos mais adiante neste trabalho, com a utilização dos dados para reprisar a jogada em um sistema baseado em regras para testes de segurança.

2.5.2 Uso do registro na qualidade do software

Programadores e testadores estão muito interessados nos aspectos de software do jogo como *bugs*, o que pode causar diversos problemas e perdas [35][36]. Alguns jogos

implementam um sistema, para reportar *bugs* remotamente, que envia o estado da aplicação para um servidor quando um erro crítico "quebra" o software [37], enquanto o usuário final o está usando. Esses dados podem ser utilizados para entender o erro e consertar. Esse formato, no entanto, não é ideal pois pede a ajuda de um jogador que teve sua experiência prejudicada por um problema que foge ao seu alcance. A frustração já foi criada e pode, nesse momento, ser irremediável. Dessa maneira, é preferível remover o máximo possível de problemas de software antes do lançamento do jogo (idealmente lançar sem nenhum erro).

Simular jogadores para provocar bugs automaticamente pode ajudar [8] mas, a única maneira confiável de minimizar o problema é utilizando processos formais de teste [38]. No entanto, os relatórios de *bugs* são extremamente complexos e envolvem uma complicada documentação para reproduzi-lo. Somente a partir da reprodução do *bug* o programador é capaz de corrigi-lo. A geração de registros das sessões produz um relatório detalhado para a reprodução. A facilidade de reprodução de um *bug*, reduzindo os custos para manutenção do código, é mais um dos benefícios dos dados de *gameplay*. Esse registro, e a possibilidade de reprise da sessão, também reduzem drasticamente os custos com testes de regressão, uma vez que podem ser feitos de maneira automática.

Uma abordagem para testes bastante popular é a do *Test Drive Development* [39] no qual o desenvolvedor cria os testes antes de efetivamente codificar uma funcionalidade. Dado que existe a possibilidade de usar um registro de jogo para reprisar, o programador pode criar os testes diretamente em um registro de jogo. E assim, "executar" a nova funcionalidade para saber se funcionará corretamente quantas vezes desejar.

2.5.3 Uso dos registros para melhoria de segurança

Programadores também são responsáveis por garantir a segurança e evitar problemas de "*cheat*" (ou burlar o jogo), um notável fenômeno de jogos na Internet. Burlar o jogo não só é possível devido a falhas de software como é fácil de ser feito [40].

Existem pelo menos quinze maneiras diferentes de burlar um jogo [40]. No entanto, o *cheating by exploiting misplaced trust* é um dos mais comuns e também combatível com o uso de dados das sessões de jogo. Esse tipo de "roubo" é possível pelo excesso de confiança no

cliente de jogo (o executável que fica na máquina do jogador). Como o cliente fica na máquina do jogador, o jogador pode modificar o executável (ou até trocar por completo) e usar isso a seu favor. Esse tipo de ataque é conhecido como ataque ativo e tem, principalmente, quatro formas: *spoofing*, *replay*, *modification of message content* and *denial of service* [41].

A modificação de conteúdo de mensagem, por exemplo, é muito comum em jogos *web* (jogados no navegador). O *ranking* do jogo é feito a partir das pontuações dos jogadores. A pontuação é normalmente enviada, ao final da partida para o servidor. O jogador pode, dessa maneira, interceptar a mensagem a ser enviada e modificar o valor da sua pontuação manualmente. Com dados de *gameplay*, a sessão inteira pode ser verificada para garantir que aquela pontuação é factível e se foi realmente a obtida pelo jogador.

2.6 RESUMO

Esse capítulo mostra as diversas situações em que os dados de engajamento e demografia, por exemplo, são de extrema utilidade e como ajudam os desenvolvedores de jogos. No entanto, também fica clara a deficiência para alguns casos: o desenvolvimento da indústria depende de análises de dados de sessões de jogos, para possibilitar a criação de produtos ainda mais engajantes e inovadores. Mas, a captura de tais dados de *gameplay* não é simples pois, envolve diversos desafios que são discutidos no próximo capítulo.

3. DESAFIOS NA AQUISIÇÃO E REPRESENTAÇAO DE DADOS DE *GAMEPLAY*

Este capítulo apresenta os desafios na aquisição de dados de *gameplay* e, dessa maneira, reforçando a necessidade de ajuda aos desenvolvedores durante o registro de dados em um jogo.

3.1 INTRODUÇÃO

A captura de dados baseados no ciclo ARM é relativamente simples e feita com bastante autoridade pelas ferramentas disponíveis - principalmente dos dados de *gameplay* e demografia. Desta maneira, o foco deste trabalho é nos desafios envolvidos na aquisição de dados de *gameplay*. Esta seção busca discutir e responder às perguntas: Que dados capturar, como representar tais dados de uma maneira mais formal, como codificar o jogo para capturar tais dados e os requisitos para um sistema de recuperação dos dados que facilite a análise. No caso ideal, tratamos do problema do "*replay*". Ou seja, que dados de *gameplay* são suficientes, sem redundâncias, para que a jogada possa ser refeita a partir de um registro.

3.2 QUAIS DADOS CAPTURAR?

A maioria dos jogos modernos tem o potencial de gerar quantidades enormes de dados a partir de uma única sessão de jogo. Somado ao fato de que jogos de sucesso possuem milhões de usuários ativos todos os dias, guardar todos os dados referentes a uma sessão de jogo não é uma opção viável na maioria dos casos. Além da banda de comunicação necessária e o espaço em disco, é importante a atenção na tratabilidade dos dados, para evitar a "*curse of dimensionality*" [42].

Escolher os dados de *gameplay* relevantes a serem capturados durante uma sessão é difícil, pois requer muito conhecimento prévio dos envolvidos no projeto. Algumas decisões a serem tomadas, com relação a que dados capturar, envolvem sérios *trade-offs* e compromissos por parte do desenvolvedor.

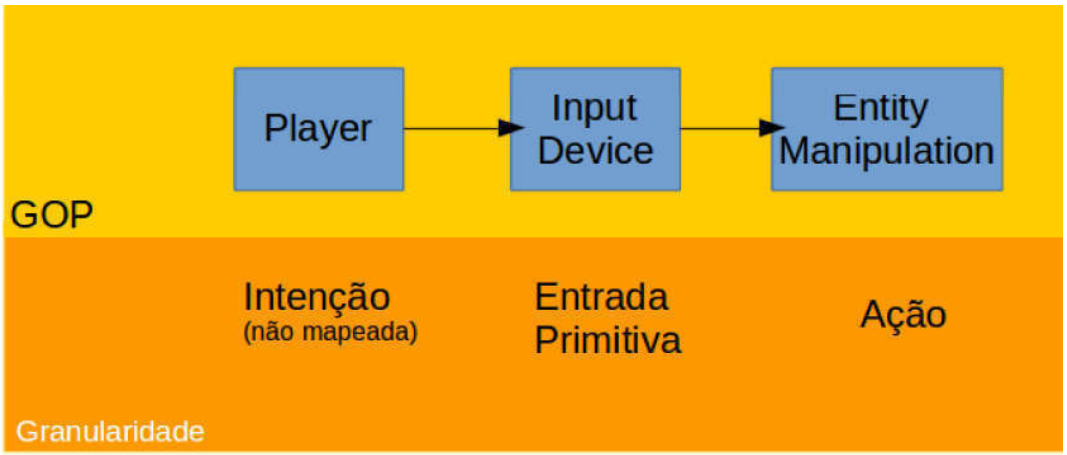
Um dos principais *trade-offs*, quando falamos de dados de *gameplay*, é a definição do nível de granularidade que será utilizado. Podendo ser (1) entrada primitiva (teclas pressionadas, cliques do *mouse*, etc.), (2) ações (ir para a esquerda, pular, correr, etc.) ou (3) intenções (fugir de um inimigo, evitar um obstáculo, etc.).

Normalmente, entradas primitivas são muito verborrágicas e geram uma grande quantidade de dados intratáveis ou, no melhor dos casos, a necessidade de simular todas as regras do jogo novamente para criar dados relevantes. Ou seja, muitas vezes, requer que as regras do jogo sejam implementadas tanto no jogo em si quanto no sistema de análise dos dados. Em alguns jogos, a lógica pode ajudar e facilitar essa implementação dupla (como é o caso de vários jogos casuais de Facebook).

Intenções, por outro lado, são muito abstratas e em certos casos podem omitir dados importantes. Por exemplo, "evitar um obstáculo" pode ser feito de várias maneiras diferentes e em uma análise mais profunda pode ser necessário entender como o jogador executou tal intenção. Sem os dados de menor granularidade é impossível descobrir como certa ação foi executada. Por outro lado, é possível inferir intenções a partir das granularidades mais baixas.

Ações possuem o melhor custo benefício apesar de ainda serem insuficientes em certos casos como, por exemplo: os usuários preferem utilizar o mouse ou o teclado para disparar a arma?

Figura 5 - As diferentes granularidades da representação dos dados e seu mapeamento no Game Ontology Project (GOP)[43].



Fonte: o autor

Ainda é possível a utilização de uma abordagem híbrida misturando dados de diferentes granularidades para casos específicos. Por exemplo, para saber quais usuários preferem mouse ou teclado, essa informação pode ser também registrada, juntamente com vários outros aspectos, utilizando-se uma granularidade de ação.

A escolha errônea dos dados pode acarretar problemas de grande escala pois, ao perceber algum dado necessário que não está sendo adquirido nas sessões, é necessário criar uma atualização do jogo para que o dado passe a ficar disponível. A atualização terá que ser redistribuída para um grande número de jogadores. Esses jogadores terão que jogar várias vezes para que o novo dado possa ser capturado [8]. Em alguns casos, o distribuidor pode levar vários dias para aprovar e começar a distribuir a nova versão.

3.3 COMO REPRESENTAR OS DADOS?

Representação de conhecimento é um campo vasto e complexo da Ciência da Computação. No campo da análise de jogos, no entanto, o foco é voltado para alguns poucos formalismos: lógica atributo-valor, lógica de primeira ordem e ontologias. Cada um dos formalismos possui prós e contras relacionados à expressividade e tratabilidade do conhecimento guardado.

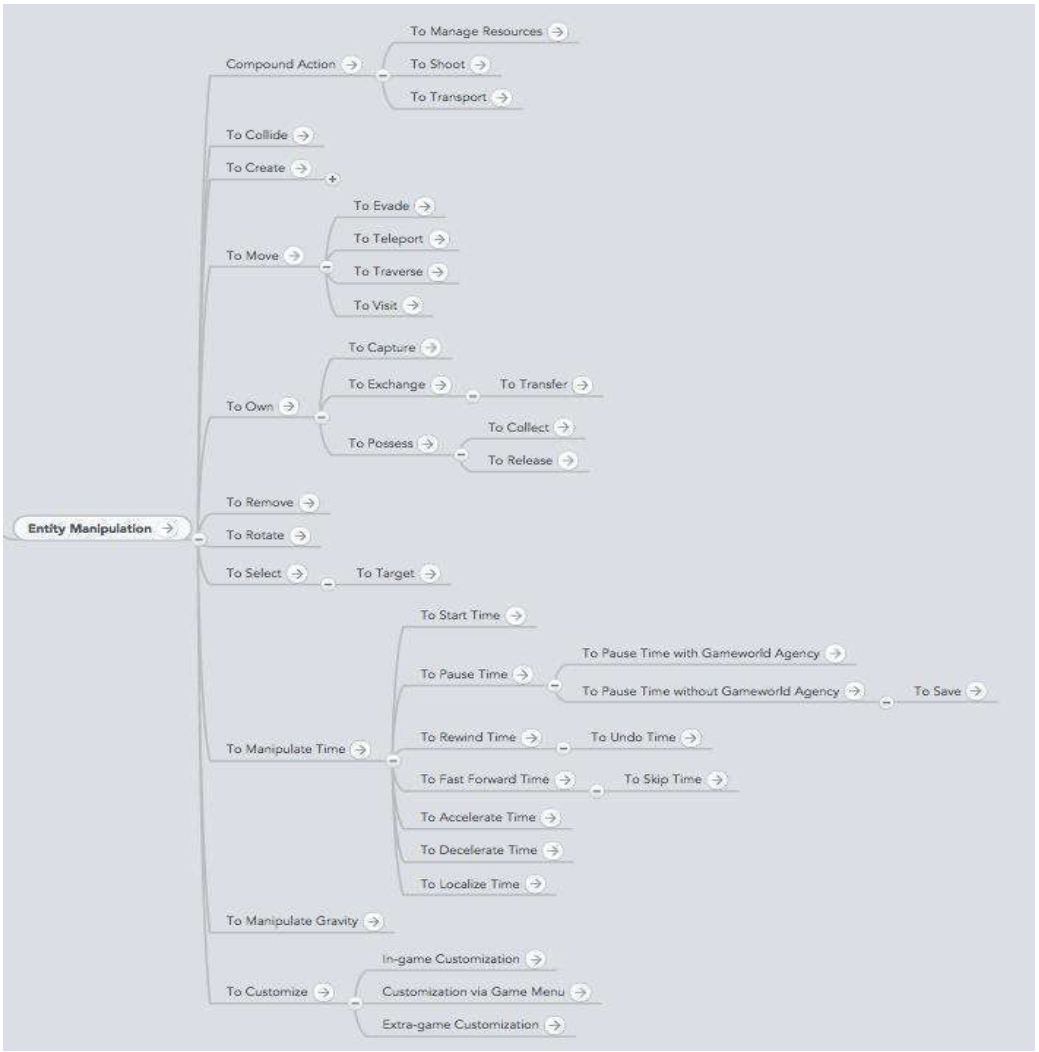
A lógica de atributo-valor é simples e tratável, mas não possui a expressividade necessária em muitos casos. Por exemplo, descrições simples de eventos como, *Above* (X,Y) ∧ *Left*(X,Z) ∧ *Behind* (X,G), para representar a posição de um objeto em relação a outros, não pode ser feita de maneira concisa para muitos objetos usando a representação da lógica atributo-valor [44]. Quantificadores universais e existenciais são um outro exemplo de elementos expressivos muito úteis que não estão disponíveis na lógica de atributo-valor. É difícil, por exemplo, criar uma busca nos dados para responder perguntas como: Existe entre os jogadores algum caso no qual o chefão de um estágio foi morto com uma habilidade especial? Essa busca requer o uso de lógica de primeira ordem, com uma representação como: $\exists x,y,z$ *Player*(x) ∧ *Boss(*y) ∧ *EspecialAbility*(z) ∧ *Killed* (x, y) ∧ ¬ *Used* (x, z).

É o caso do *Game Ontology Project* (GOP) [43] que oferece um vocabulário capaz de representar sessões de jogo, tanto no nível de granularidade das entradas primitivas (sub-

conjunto chamado de Input Method), quanto na granularidade das ações (usando a parte de Entity Manipulation mostrada na figura 6).

Ontologias, como o GOP, também foram bastante usadas para representar dados de jogos. Em termos de tratabilidade e expressividade, ontologias ficam no meio do caminho entre a lógica atributo-valor e a de primeira ordem. O maior trunfo das ontologias é ir além da sintaxe puramente e propor um novo vocabulário para o conhecimento a ser representado.

Figura 6 - Estrutura “Entity Manipulation” do GOP.



Fonte: [43]

A utilização de ontologias para representar o conhecimento baseado em casos e não em regras, assim como a busca por casos em ontologias indexadas, são comuns na literatura

[45][46]. Dessa forma se facilita a identificação de casos em que o jogador usou uma habilidade especial para vencer um “chefão” de estágio, como exemplificado acima.

Outro ponto positivo, é a possibilidade de usar a ontologia como ponto de partida, na fase de mapeamento de domínio [47], na criação de uma Linguagem Específica de Domínio (DSL). A criação de uma DSL pode ajudar no tratamento dos dados devido às diversas ferramentas disponíveis para tal. A vantagem na utilizaçao de uma DSL é trabalhar no nível correto de abstração sem se preocupar com construções sintáticas e semânticas de linguagens ad-hoc[48]. A utilização de DSL para a programação de jogos, principalmente no que chamamos de linguagens de script, é muito popular e presente em frameworks de criação, como a Unreal Engine [49] ou na criação de jogos para dispositivos móveis em HTML5 [50]. No entanto, a utilização de DSL [51] para representação de jogadas ainda é incipiente. Apesar disto, há algumas iniciativas neste sentido, como é o caso da representaçao de ações do jogador feita pela Unreal Engine, e mostrada no exemplo abaixo.

Pseudo-código 1 - DSL para criação de máquina estado de comportamento de personagens na Unreal Engine.

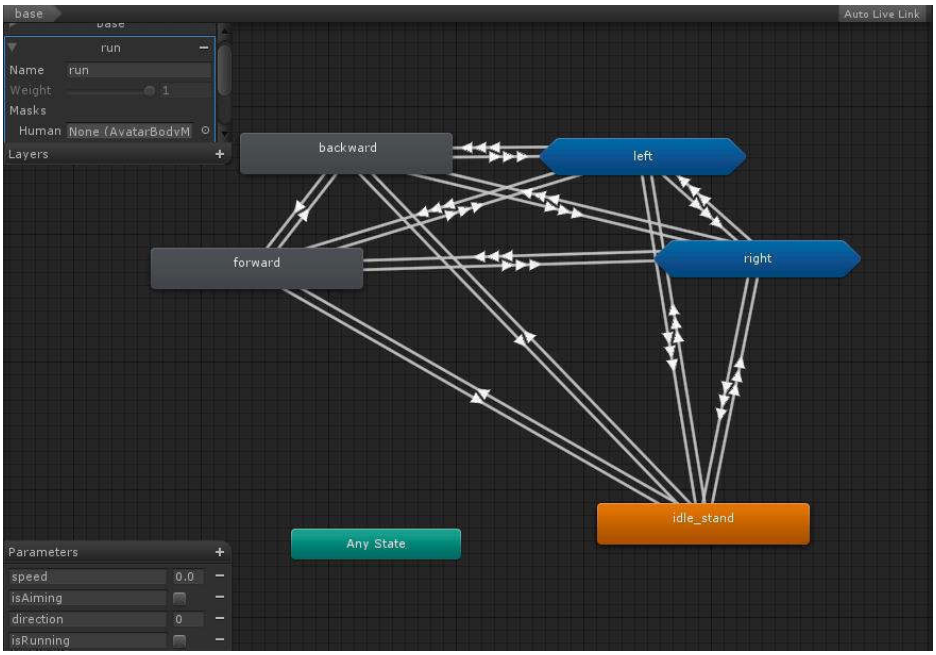
```
// Base Attacking state.
state Attacking {
// Stick base functions here...
}

// Attacking up-close.
state MeleeAttacking expands Attacking {
// Stick specialized functions here...
}

// Attacking from a distance.
state RangeAttacking expands Attacking {
// Stick specialized functions here...
}
```

Fonte: o autor

Além da utilização de *DSL*s textuais, as *DSLs* visuais [52] estão cada vez mais populares. Pois, além de trabalhar no nível correto de abstração e granularidade, facilitam o uso dos dados por stakeholders do projeto que não podem programar em uma linguagem de programação. A própria Unreal Engine, aleḿ da Unity [53] mostrada na Figura 7, possui

ambientes de programação com *DSLs* visuais para programar o algoritmo de escolha e transição entre animações de um personagem dado o estado em que este se encontra.

Figura 7 Programação do "Animation picking" usando uma DSL visual do framework de programação de jogos Unity.



Fonte: https://docs.unity3d.com/Manual/index.html

Por fim, uma linguagem específica para o domínio de uma sessão de jogo pode ser vista como uma maneira de usar o jogo como uma plataforma programável e a DSL seria, então, uma maneira de programar tal plataforma. Ou seja, os jogos gravados podem ser posteriormente "executados" na "plataforma jogo".

3.4 COMO MODIFICAR O CÓDIGO DO JOGO PARA CAPTURAR OS DADOS?

Para capturar os dados é necessário modificar o código do jogo. No entanto, essa tarefa que parece simples, esconde sérios problemas que dificultam o "separation of concerns". Em outras palavras, a captura de dados é um "*cross-cutting concern*" pois afeta todas as partes do

código e pode, potencialmente, gerar códigos duplicados e dependência excessiva aumentando a complexidade do código e, consequentemente, o custo de manutenção e extensão[54].

O código abaixo, por exemplo, mostra uma chamada para o código de dados inserido em um código da camada de apresentação. Essa função faz parte de uma tela de final de estágio onde o jogador pode ver sua performance. Basicamente, as informações de um estágio do jogo recentemente completado são passadas para os painéis, o *callback* de um botão é definido e uma animação é iniciada. No meio de tudo isso, a ação de estágio concluído é logada.

Pseudo-código 2 - Código para executar a mudança de placar e registrar a completude de um estágio

```
public void SetData(MetaLevel metaLevel, int score, Action onPlayClick)
{
        base.SetSocial(metaLevel);
        int level = metaLevel.LevelIndex;
        transform.Find("Panel/Title/Text").GetComponent<Text>().text =
        string.Format("popup.level.title".Localize(),level);
        Facade.Instance.ExecuteActionLevelComplete(level, score);
        StartCoroutine(ScoreAnimation(score));
        this.onPlayClick = onPlayClick;
}
```

Fonte: o autor

A mudança da tela de final de estágio (para um novo layout, por exemplo), por um desenvolvedor desavisado, pode acarretar a remoção da chamada do código de registro dos dados. Dessa maneira, o dado não será mais armazenado pelo jogo. Se o código for movido da camada de apresentação para a camada de aplicação, por exemplo, existe o risco de duplicação de código devido às diversas condições de final de estágio de um jogo (por exemplo, vencer o estágio, morrer em um buraco, morrer por um inimigo, entre outros). O código abaixo, por exemplo, mostra o evento sendo gravado quando o jogador perde uma vida.

Pseudo-código 3 - Código do registro de um evento do jogador perdendo uma vida

```
public Dictionary<string,object> ExecuteActionLevelComplete(int level, int score)
{
        LifeSystem.Singleton.RemoveDebt();
        Analytics.CompleteLevel(level,score);
        return serverController.ExecuteActionLevelComplete(level, score);
}
```

Fonte: o autor

Podemos identificar dois problemas em potencial nesta abordagem. Primeiro, o evento está sendo gravado em outras condições de final de estágio? Segundo, quando novas condições de final de estágio são incorporadas ao jogo, onde deve ser incluído o código de registro dos dados?

Em um jogo onde cada nível tem um limite de tempo para ser concluído, um limite de colisões a serem feitas e também a condição de vitória, podemos ter no primeiro caso, o código de ativação da métrica *"CompleteLevel*" sendo chamado em apenas uma ou duas condições. No segundo cenário, mesmo que o código seja incluído nas três possibilidades, será triplicado? Se sim, quando da mudança de código, existirá documentação para que o desenvolvedor ajuste nos três lugares? Se não, que artifícios serão usados para resolver o problema.

Para diminuir os problemas de *"cross-cutting"* é necessário prover ao desenvolvedor o entendimento necessário sobre as necessidades de capturas de dados o mais cedo possível. Se as necessidades de captura estão claras no começo do ciclo de desenvolvimento, o programador tem mais facilidade e capacidade para criação de um código com arquitetura e design suficientes para atender a tais requisitos [54]. Esse entendimento também ajudará na documentação adequada dessas decisões de projeto, que são cruciais para evitar os problemas de expostos nesta seção.

3.5. COMO RECUPERAR E USAR OS DADOS?

Alguns dos interessados no projeto, principalmente as equipes de negócios e criativa, requerem uma maneira intuitiva e interessante de representação e busca dos dados do jogo. Os formatos usuais de representação, como lógica de primeira ordem, são normalmente longos textos com sintaxe desinteressante e complicada, de difícil interpretação, para a maioria das pessoas. Por outro lado, programadores preferem formatos que facilitem o tratamento dos dados por código. Dessa maneira, é possível criar algoritmos de aprendizagem de máquina e mineração de dados, com mais facilidade.

A apresentação, por exemplo, pode ser visual [8;55]. É mais fácil identificar os caminhos e decisões tomados pelo jogador durante uma partida e reconhecer problemas no

design do jogo, a partir de uma representação visual dos dados. Dados espaciais de jogos são comumente representados visualmente [56].

Por outro lado, a representação focada em computadores precisa ser a mais formal e padronizada possível, para ser utilizada por diferentes algoritmos para diferentes análises. A falta de padronização leva a limites na interoperabilidade e, consequentemente, dificulta o reuso de código [12].

Em uma sessão de jogos, os desafios relacionados à descrição de dados de *gameplay* são vários, como explicitados nesse capítulo. Não só a descrição é desafiadora, como também a coleta dos dados e a recuperação e análise. Dessa maneira, algumas características são desejáveis para as potenciais soluções deste problema. No próximo capítulo, tais características são apresentadas e o estado da arte em *frameworks* para manipulação de dados em jogos é analisado.

3.6 COMO FAZER REPRISE DE PARTIDAS?

A reprise (ou "*replay"*) de um software é um problema complexo e vem sendo estudado há muito tempo nos mais diferentes contextos [57;58]. A execução de um "*replay"* requer registros detalhados do que foi executado (o que chamamos de registros com dados de *gameplay*). Registros de alta fidelidade já foram usados nos primeiros sistemas para reprises da execução completa de uma máquina, com o intuito de depurar sistemas operacionais ou "*cache coherence*" em sistemas com memória compartilha distribuída [59;60]. Esses registros, no entanto, necessitavam de um hardware próprio, um *kernel* modificado ou uma máquina virtual especial, o que limitava o uso. Além disso, a produção intensa de dados nesses registros tornava a persistência de toda a execução impossível. Normalmente, só os últimos segundos de execução antes de um erro eram reportados.

Em níveis mais altos de abstração, em uma linguagem de programação por exemplo, a quantidade de dados gerados pode ser diminuída drasticamente e várias ferramentas - como *Mugshot* [61], *DejaVu* [62] e *liblog* [63] - existem para o registro da execução de um software. No entanto, são insuficientes para a execução no nível da aplicação. No caso de erro de hardware, ou de sistema operacional, não é possível entender o que aconteceu. Essas ferramentas, se baseiam no mecanismo de reflexão ("*reflection*") dessas linguagens e tentam

criar os registros de maneira transparente para o desenvolvedor. Ao utilizar a reflexão da própria linguagem, o ambiente de execução do registro já existe, então os *frameworks* não precisam se preocupar em criar um novo. No entanto, ao fazer isso, limita as opções do desenvolvedor que precisa utilizar uma linguagem de programação ou *framework* específico.

Para a reprise de sessões de jogos, os *frameworks* baseados em reflexão, mostram-se deficientes também pela enorme quantidade de informação desnecessária gerada. Dessa maneira, o "*overhead*" para a análise de dados específicos de jogo é muito grande. O *Mugshot*, por exemplo, utiliza as entradas do jogo na granularidade de *input*. Dessa maneira, a busca por eventos específicos do jogo necessitaria de uma nova simulação para cada registro, para chegarmos ao nível de granularidade de ação e possibilitar a busca por eventos como: $\exists x,y,z$ *Player(x)* $\wedge$ *Boss(y)* $\wedge$ *EspecialAbility(z)* $\wedge$ *Killed (x, y)* $\wedge \neg$ *Used (x, z)*. Ou seja, apesar de resolver o problema da reprise, não seria suficiente para o uso por todos os *stakeholders* envolvidos no projeto e ainda assim, seria necessária a criação de uma linguagem de representação das partidas para o registro na granularidade de ação.

Dessa maneira, é necessário o uso de uma linguagem no nível de ação que sejaexpressiva o suficiente para a execução de reprises. Na criação de um novo formato de registro, o ambiente de execução precisa ser criado. No caso de jogos digitais, podemos adaptar o jogo para execução de reprises, utilizando os registros e não os comandos do jogador como entrada. É importante salientar que não é uma troca simples pois os registros estão no nível das ações enquanto que a entrada do jogador no nível de *input*.

Quadro 2 - Mapeamento dos eventos de entrada no Mugshot.

	Tipo de Evento	Exemplos	Capturado por Mugshot
Eventos DOM $ 3.1.3	Mouse Tecla Carregamento Formulários (tag <form>) Corpo do HTML (tag <body>)	Clique, Passar o mouse Soltar tecla, apertar tecla Carregarmento Focar, desfocar, selecionar, mudar Rolagem, redimensionar	Sim Sim Sim Sim Sim
Interrupções $ 3.1.2	Temporizadores AJAX	setTimeout(f, 50) req.onreadystatechange = f	Sim Sim
Funções não determinísticas $ 3.1.1	Recuperar data Gerar número aleatório	(new Data()).getTime() Math.random()	Sim Sim
Seleção de Texto $ 3.1.8	Firefox: window.getSelection() IE: document.selection	Selecionar texto com o mouse Selecionar texto com o mouse	Sim Parcialmente
Objetos de navegação externos $ 3.1	Vídeo em formato *Flash* *Applet* em formato Java	Usuário pausa o vídeo *Applet* Java atualiza a tela	Não Não

Fonte: [61]

Além dessa assimetria no nível de granularidade, existem problemas inerentes à execução de reprises. O principal deles é o registro de eventos não determinísticos. Esses eventos, por diversos motivos, podem acontecer de diferentes maneiras em diferentes execuções de um mesmo software. E no caso específico de jogos, esse problema é agravado pela quantidade de eventos desse tipo. É muito comum, por exemplo, que diversos elementos de um jogo sejam decididos com o uso intensivo de funções para geração de números aleatórios - a posição inicial das peças na tela de um jogo de quebra-cabeça, por exemplo. Em todos os eventos desse tipo, o resultado precisa ser devidamente registrado e tratado de maneira apropriada na plataforma de execução.

3.7 RESUMO

Esse capítulo faz um apanhado dos diversos desafios envolvidos na criação de registros de uma partida. De uma maneira geral, vários desses desafios estão sendo tratados de maneira isolada. Essa abordagem, apesar de funcional, cria problemas diversos para os desenvolvedores,

uma vez que resulta em diferentes formatos para diferentes registros pensados para diferentes objetivos - um registro para as métricas de retenção e outro para a reprise, por exemplo. Dessa maneira, impacta ainda mais na complexidade de código com problemas de duplicação de código e *"cross-cutting*". O próximo capítulo analisa diversas dessas abordagens, olhando para exemplos de soluções e discutindo como cada uma delas trabalha os desafios descritos.

4. ESTADO DA ARTE: SOLUÇÕES PARA O REGISTRO DE SESSÕES DE JOGOS

Neste capítulo são definidas as propriedades essenciais para soluções de registro de sessões (*frameworks* de analítica) e discute sobre opções existentes em relação a essas propriedades. Esse debate irá facilitar o posicionamento do *GameGuts* nos capítulos seguintes.

4.1 INTRODUÇÃO

Esta seção define propriedades de *analytics frameworks* que precisam ser consideradas durante a captura, representação, armazenamento, busca e apresentação de dados de *gameplay* de jogos. Em seguida, *frameworks*, tanto comerciais quanto acadêmicos, são analisados e discutidos sob a ótica das propriedades definidas.

Cada uma das propriedades descritas abaixo foi definida com base nos desafios ligados a dados de *gameplay* de seções de jogos, descritos no capítulo anterior:

Conhecimento prévio necessário: A maioria dos frameworks assume que o desenvolvedor do jogo entende as necessidades de captura de dados de *gameplay*. Entendemos que o desenvolvedor deve ser ajudado ao máximo nesse ponto. Essa característica está relacionada ao desafio de quais dados capturar.

Expressividade: a linguagem de representação dos dados do jogo deve ser suficientemente expressiva para garantir a representação de todos os tipos de dados de uma sessão de jogo. Essa característica está relacionada ao desafio de como representar os dados e, principalmente, a possibilidade de reprises.

Separação de conceitos (do inglês, *separation of concerns*): Codificar o registro dos dados em um jogo e escrever sua documentação deve ser uma tarefa simples ou, ao menos, feita de forma direta pelo desenvolvedor. O código e documentação resultantes devem ser fáceis de manter por um longo período de tempo (sob a perspectiva de GaaS). Essa característica está relacionada ao desafio de modificação do código de jogo.

Interoperabilidade, ou Formato "*Computer-friendly*": é importante que o formato de representação seja facilmente guardado, recuperado e processado por algoritmos. Dessa

maneira, permitimos com facilidade a mineração de dados, geração automática de relatórios, aprendizagem de máquina, entre outras coisas. Essa característica está relacionada ao desafio de recuperação e uso dos dados.

Formato amigável ao usuário: A apresentação dos dados deve ser de fácil entendimento e interpretação por um humano, especialmente as equipes não-técnicas como design e negócios. Essa característica está relacionada ao desafio de uso dos dados capturados.

Extensibilidade: O *framework* deve ser extensível para facilitar a adiçao de novas métricas à medida em que o jogo evolui. Essa característica está relacionada ao desafio da escolha de qual representação dos dados, bem como da modificação do código de jogo.

4.2 *FLURRY* E OUTRAS SOLUÇÕES COMERCIAIS

Pelo menos uma dezena de opções comerciais proveem soluções para o registro de partidas, como *Flurry*, *gamenalytics*, *google analytics*, *deltadna*, entre outros. No entanto, todas utilizam a mesma abordagem com foco em ser amigável para o usuário e pequenas diferenças nas métricas de dados de engajamento. Também são fornecidos mecanismos para o suporte de dados de *gameplay* com algumas limitações. Esta seção discute como cada um dos desafios do registro de jogadas é abordado por essas ferramentas e a habilidade de cada uma delas com relação às propriedades descritas na seção inicial deste capítulo.

Por ser o *framework* mais popular na indústria de jogos, com mais de 150 bilhões de sessões de mais de 170 mil desenvolvedores, o *Flurry* [64] será analisado nesta seção.

A popularidade de *Flurry* se deve, em parte, à facilidade de integração do cliente. Apenas a inicialização do *framework,* com uma simples chamada do cliente no jogo, já garante ao desenvolvedor diversos dados de demografia e engajamento - como tempo de sessão, sessões por dia, usuários ativos por dia e mês - métricas essas fora do escopo deste trabalho. Dessa maneira, para dados de engajamento e demografia, *Flurry* não requer nenhum conhecimento prévio do que capturar e lida com o problema de manutenção do código para o desenvolvedor.

Por outro lado, para dados de *gameplay*, *Flurry* utiliza eventos customizáveis (do inglês, "*custom events*") que consiste de uma tabela "*hash"* na qual o desenvolvedor guarda pares de

atributo e valor. Essa estrutura de dados fornece expressividade suficiente para qualquer jogo e normalmente é utilizada para capturar dados de *gameplay*. Um exemplo simples para captura de um evento para representar que o jogador lançou uma habilidade especial seria:

Pseudo-código 4 - Código Flurry para uso de uma habilidade especial pelo jogador

```
Map<String, String> articleParams = new HashMap<String, String>();
articleParams.put("Character", "Player");
articleParams.put("AbilityType", "Special");
articleParams.put("Timestamp", "109");
FlurryAgent.logEvent("UseAbility", articleParams);
```

Fonte: o autor

A abordagem que permite grande expressividade, requer bastante conhecimento prévio do desenvolvedor sobre o que registrar pois, não fornece processo ou mecanismos de ajuda, nos desafios do que guardar e como representar, discutidos anteriormente. O exemplo acima, pode ser feito de outras maneiras, como por exemplo:

Pseudo-código 5 - Outra maneira de registrar o uso de uma habilidade especial pelo jogador

```
Map<String, String> articleParams = new HashMap<String, String>();
articleParams.put("Character", "Player");
articleParams.put("EventType", "SpecialAbility");
articleParams.put("Timestamp", "109");
FlurryAgent.logEvent("GameEvent", articleParams);
```

Fonte: o autor

Como podemos ver nos exemplos de código acima, a facilidade de integração para dados de engajamento e demografia não é a mesma para dados de *gameplay*. Não existe mecanismo ou processo para auxiliar os desenvolvedores com problemas de *separation of concerns*, dificultando o processo de criação do código necessário para capturar as métricas no jogo.

Em termos de interoperabilidade, existe um conjunto de *APIs REST* para recuperar informações capturadas anteriormente. No entanto, só a *API EventMetrics* [65] possibilita a recuperação de dados de eventos customizados. Apesar de prover a resposta nos formatos XML e JSON, somente informações combinadas de um certo evento estão disponíveis. No código abaixo, um exemplo de retorno mostra que para cada evento só são fornecidas informações superficiais como o total de vezes em que o evento ocorreu. Não é possível, por exemplo,

segmentar os eventos por parâmetros (se temos o evento andar, não podemos recuperar somente andar para a esquerda).

Pseudo-código 6 - Exemplo de retorno a uma chamada a API EventMetrics de Flurry

```
{"@type":"Summary",
"@version":"1.0",
"@generatedDate":"3/1/10 2:10 PM",
"@startDate":"2010-03-01",
"@endDate":"2010-04-15",
"@versionName":"1.0",
"event":[
 {
   "@eventName":"Logout",
   "@usersLastDay":"123",
   "@usersLastWeek":"123",
   "@usersLastMonth":"123",
   "@avgUsersLastDay":"123",
   "@avgUsersLastWeek":"123",
   "@avgUsersLastMonth":"123"
   "@totalCount":"123",
   "@totalSessions":"123",
   "@date":"2010-03-01"
 },
 {
   "@eventName":"OnClick",
   "@usersLastDay":"123",
   "@usersLastWeek":"123",
   "@usersLastMonth":"123",
   "@avgUsersLastDay":"123",
   "@avgUsersLastWeek":"123",
   "@avgUsersLastMonth":"123"
   "@totalCount":"123",
   "@totalSessions":"123",
   "@date":"2010-03-02"
 },
 ...
 ]
}
```

Fonte: o autor

Além da opção de API, *Flurry* permite o download dos dados de seção em formato csv. Esses dados podem então ser utilizados para análises diversas como as discutidas no capítulo 2. No entanto, a falta de uma API compromete o grau de interoperabilidade pois precisa que uma pessoa faça o download do arquivo. É importante salientar que essa é uma decisão de projeto válida, dada a pressão comercial por performance. *Flurry*, e outras soluções comerciais,

guardam *terabytes* de dados e prover acesso direto via API comprometeria a banda dos servidores, corrompendo a experiência de uso do restante do sistema.

Pseudo-código 6 - Trecho de arquivo no formato csv de um jogo capturado com *Flurry*

Timestamp, Session Index, Event, Description, Version, Platform, Device, User ID, Params,

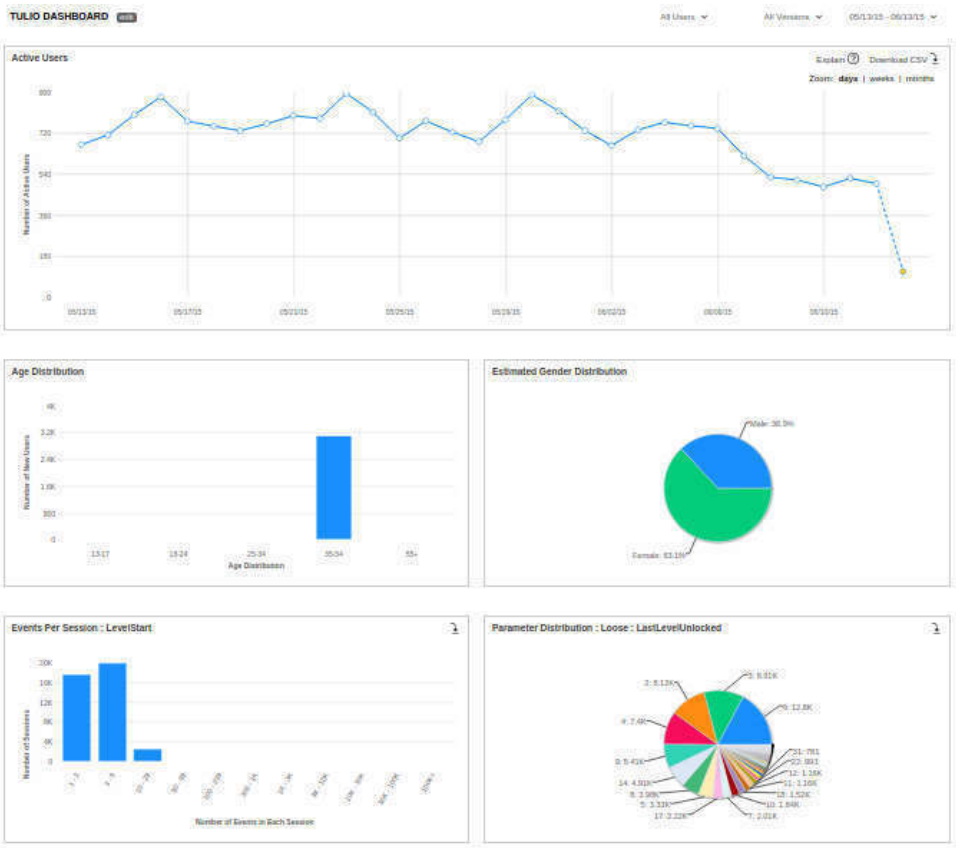
"Nov 01, 2015 06:57 AM",1, LevelStart,,1.2.5,Android,CCE Motion Hold TR72,,"{ UserType : Free; CurrentScreen : Game; LastLevelUnlocked : 1; DayIntallGap : 0; Upgrades : {""Bomb"":0, ""Potions"":0, ""TimeMachine"":0}; Level : 1; CoinAmount : 0; Facebook : 0}",
"Nov 01, 2015 06:57 AM",2,LevelComplete1,,1.2.5,Android,CCE Motion Hold TR72,,"{ UserType : Free; CurrentScreen : Win; TimeSeconds : 12; Cake : 3; LastLevelUnlocked : 2; DayIntallGap : 0; Upgrades : {""Bomb"":0, ""Potions"":0, ""TimeMachine"":0}; CoinAmount : 0; Facebook : 0; Moves : 12}",
"Nov 01, 2015 06:50 AM",1,LevelStart,,1.2.5,Android,CCE Motion Plus Sk351,,"{ UserType : Free; CurrentScreen : Game; LastLevelUnlocked : 2; DayIntallGap : 2; Upgrades : {""Bomb"":0, ""Potions"":0, ""TimeMachine"":0}; Level : 1; CoinAmount : 0; Facebook : 0}",
"Nov 01, 2015 06:50 AM",2,LevelComplete1,,1.2.5,Android,CCE Motion Plus Sk351,,"{ UserType : Free; CurrentScreen : Win; TimeSeconds : 36; Cake : 3; LastLevelUnlocked : 2; DayIntallGap : 2; Upgrades : {""Bomb"":0, ""Potions"":0, ""TimeMachine"":0}; CoinAmount : 0; Facebook : 0; Moves : 16}",

Fonte: o autor

Flurry, como as outras opções comerciais, não fornecem mecanismos para extensão de suas funcionalidades. Desta maneira, o trabalho do desenvolvedor é dificultado, ao querer incluir novas análises no jogo em questão, exceto quando as análises já são as existentes como o *Funnel.*

Apesar das limitações de interoperabilidade e extensão, o *Flurry* utiliza uma abordagem visual para recuperação de informação, que é rápida e eficiente para todos os tipos de usuário, por ter um foco em ser amigável aos humanos. A Figura 8 (4) é um exemplo de relatório do *Flurry* que mostra a quantidade de usuários ativos no jogo a cada dia entre 13 de maio e 13 de junho de 2015 no topo. Em seguida, a segmentação por idade (onde todos os usuários estão entre 35 e 54 anos) e por sexo. Além de dados de *gameplay* - do lado esquerdo - com quantas seções com 1 ou 2 eventos, 3 a 9 e assim por diante e - do lado direito - a distribuição do evento *Loose* de acordo com o parâmetro *LastLevelUnlocked* que, nesse jogo, podemos ler como a quantidade de partidas perdidas segmentadas por nível do jogo.

Figura 8 - Relatório Flurry com dados de engajamento e demográficos como usuários ativos e níveis iniciados e perdidos por sessão.



Fonte: Flurry Analytics - dev.flurry.com/home.do

Relatórios mais complexos com os eventos customizados podem ser criados também utilizando uma ferramenta visual chamada *Funnels*, mostrada na Figura 9. No *Funnel*, o desenvolvedor pode montar diversos contextos de utilização de seu jogo e acompanhar como os jogadores se comportam.

Figura 9 - Exemplo de um *"funnel"* montado no Flurry.

Fonte: Flurry Analytics - dev.flurry.com/home.do

Em termos gerais, *Flurry* e outras soluções comerciais são adaptações diretas das ferramentas existentes para o registro de uso de artefatos digitais, onde o usuário é apenas consumidor da mídia (como websites). A visualização de informação por gráficos e outros formatos visuais (com foco em ser amigável ao usuário) é crucial para o uso amplo e a popularização da ferramenta - principalmente por *stakeholders* que não programam como *game designers*. No entanto, essas ferramentas deixam o desenvolvedor desamparado com relação ao conhecimento prévio e àinteroperabilidade, por exemplo. Na próxima seção, serão discutidas abordagens mais interessadas nesses outros aspectos.

4.3 SOLUÇOES BASEADAS EM ONTOLOGIAS

Do lado acadêmico, abordagens focadas em interoperabilidade, ou "*computer-friendly", se* mostram promissoras. Os registros de *gameplay* utilizam o mesmo esquema baseado em eventos. No entanto, o fazem de maneira mais formal, por uso de ontologias e lógica de primeira ordem. Existe a preocupação na definição prévia do formato de representação, para que se possa utilizar diferentes algoritmos, melhorando a interoperabilidade. Esta seção discute diferentes trabalhos onde a interoperabilidade e o conhecimento prévio são tratados com mais cuidado para o registro de sessões de jogos e também os *trade-offs* feitos para possibilitar esse foco.

A principal diferença entre a abordagem de *Flurr*y e as abordagens baseadas em ontologias, está na padronização do formato de representação de uma partida. A pradonização traz diversas vantagens. Entre elas, está a existência de métodos para o uso de uma ontologia como base para a criação de outra ontologia [66], ou a personalização da ontologia para instâncias específicas de problemas [67]. Esses métodos diminuem a necessidade de conhecimento prévio por parte do desenvolvedor pois o domínio está, pelo menos parcialmente, mapeado em ontologias já existentes, como a *Game Ontology Proj*ect (ou GOP) podendo esse conhecimento ser utilizado como ponto de partida. Outro ponto positivo é a possibilidade de transformar ontologias em linguagens de domínio específico (DSLs) [47] - mais adequadas a análise por algoritmos do que um simples arquivo em formato *csv*.

Mepham [68] propõe um *framework* simples que verifica a validade dos eventos com Jess [69] a partir de uma base de regras do jogo definida em OWL [55]. Diferentemente das opções comerciais, que procuram ser genéricas o suficiente para serem usadas na maior quantidade de jogos possíveis, Mepham opta por uma abordagem específica para jogos de tabuleiro, como xadrez. Dessa maneira, opta por conhecimento prévio e interoperabilidade em detrimento de expressividade (só é possível expressar regras para o jogo de xadrez).

Pseudo-código 7 - Regras utilizadas para movimentação de peças no xadrez. Nesse caso, deslocando a peça x para a posição l.

```
go:Piece(?x)
^ go:Location(?l)
^ go:movesPiece(?x, ?l)
^ swrlx:makeOWLIndividual(?z, ?x)
→ go:MovePieceEvent(?z)
^ go:piece(?x)
^ go:destination(?y, ?l)
```

Fonte: o autor

Não existe nenhuma preocupação explícita com relação a *separation of concern especificamente* no código do cliente. No entanto, por usar um paradigma de programação

declarativa em Jess, a ambiguidade na codificação dos eventos é diminuída e a separação mais clara.

Uma base de regras também facilita extensibilidade, pois dispara regras a partir dos eventos que entram como fatos na base. Dessa maneira, para criar uma nova análise, basta criar uma nova regra. No entanto, as restrições em criar somente conjunções (a ^ b) e não disjunções (a v b) nem negações limita as possibilidades de análise.

Pseudo-código 8 - Regra disparada quando uma peça é movimentada que impede que duas peças ocupem o mesmo *lugar.go:MovePieceEvent(?y)*

^ *go:Location(?sg:l)*

^ *go:destination(?y, ?sg:l)*

^ *go:isOccupied(?l, true)*

→ *go:InvalidEvent(?y)*

Fonte: o autor

Outra iniciativa, a de Chan [12], tem seu foco na criação de uma base de conhecimento sobre jogos - tanto sobre sessões quanto sobre a produção. Os dados não são capturados dentro do jogo, mas sim de outras fontes (vídeos, *websites*, etc.). Dessa maneira, não trabalha problemas relacionados ao *separation of concerns*. Por outro lado, utiliza a *class entity manipulation* do *Game Ontology Project* (GOP) [70] para representar os dados de sessões. Reforçando a possibilidade de uso de ontologias, previamente criadas para ajudar na representação de novos domínios do conhecimento e, consequentemente, diminuir a necessidade de conhecimento prédio do desenvolvedor.

Os dados são também representados utilizando uma base de regras, nesse caso SWRL (extensão do OWL). Por trabalhar com dados imprecisos, a ontologia resultante possui diferentes níveis de granularidade, provendo um nível de expressividade ainda maior. Essa característica pode ser importante em contextos onde a velocidade de internet ou o espaço para armazenamento não são satisfatórios.

Um projeto adicional, chamado DBpedia, é apresentado como uma maneira de prover os dados de maneira estruturada. Esses dados podem ser acessados por meio de uma *API Protégé*[71]. Dessa maneira, os dados podem ser acessados diretamente por uma ferramenta focada na manipulação de ontologias e alguns tipos de análise podem ser feitas de maneira simples e rápida.

Assim como nas outras soluções em que é usado um sistema baseado em regras, a extensibilidade é facilitada pois, para cada fato adicionado à base, novas regras podem ser criadas sem maiores problemas.

Por trabalhar com a classificação de jogos e não em sua melhoria, a solução não se preocupa com a visualização dos dados de uma maneira amigável ao usuário.

O uso de ontologias traz grandes benefícios para a representação de partidas de jogos, diminuindo a necessidade de conhecimento prévio do desenvolvedor, provendo expressividade e interoperabilidade suficientes para a captura de dados de *gameplay* e tem sido extensivamente usado para análise de partidas [45]. No entanto, esses trabalhos não dão a devida importância à necessidade de recuperação e visualização dos dados por um observador humano e, dessa maneira, encontram resistência no uso em jogos comerciais.

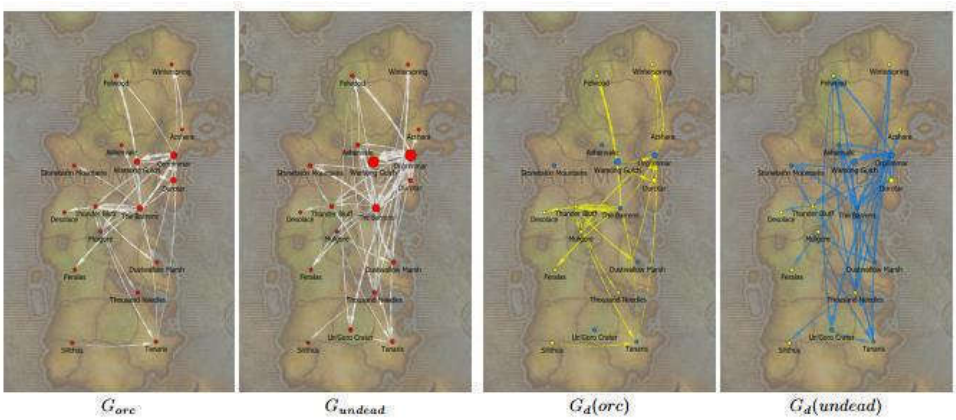
4.4 SOLUÇÕES BASEADAS EM ESTADOS

As duas alternativas apresentadas neste capítulo representam uma sessão de jogo como uma sequência de eventos. Existem, no entanto, alternativas que modelam o jogo com uma máquina de estados finita, como o *Play-Graph* [72]. Essas abordagens são, em geral, focadas na recuperação dos dados com resposta visual, amigável aos humanos, para jogos onde o *gameplay* pode ser representado com relação ao ambiente de jogo (jogos de corrida, por exemplo).

Essa abordagem tem como ponto principal a facilidade de visualização pelos usuários. Toda a representação de dados é feita para que os desenvolvedores consigam entender o comportamento de uso do jogo pelos jogadores. E, de fato, é extremamente popular entre empresas desenvolvedoras de jogos. Um exemplo está nas imagens da Figura 11, que mostram

a movimentação dos jogadores no jogo *World of Warcraft*, evidenciando as rotas mais frequentes (a informação pode ser usada para aumentar, diminuir ou melhorar os desafios no caminho, por exemplo).

Figura 10 - Deslocamento dos orcs – Gorc e Undeads – Gunded no continente de Kalimdor em World of Warcraft.



Fonte: [72]

A representação dos estados e das ações de transição entre um estado e outro é de inteira responsabilidade do desenvolvedor. Dessa maneira, exige um conhecimento prévio do que deve ser capturado. Uma dificuldade comum é a escolha entre a representação de um único estado com vários personagens (em que o estado pode facilmente ser composto por algumas centenas de variáveis) ou várias máquinas de estado, cada uma com um personagem diferente (dificultando o cruzamento dos dados).

Por outro lado, a representação pode ser formalmente descrita como um grafo e, dessa maneira, facilitar a análise por algoritmos (maior interoperabilidade apesar da falta de *APIs*). De fato, isto é um ponto muito positivo poisas abordagens para análise e representação de grafos são utilizadas ostensivamente para facilitar a visualização dos dados [72].

No entanto, esta representação restringe os tipos de análise possíveis, devido à sua expressividade restrita - tiros errando o alvo não modificam estados, não aparecem na representação visual e podem levar a um erro de análise, por exemplo.

Separation of concerns é com certeza um problema para captura desses dados no jogo. Apesar disso, este aspecto não é tratado pelos trabalhos nessa linha. A extensibilidade das análises é possível devido à natureza da representação. No entanto, também não é um aspecto abordado nesses trabalhos.

4.5 RESUMO

Como discutido neste capítulo, existem diferentes abordagens para a captura de dados mais complexos sobre seções de jogos. Cada uma das abordagens ajuda o desenvolvedor de alguma maneira.

As soluções mais comerciais focam em auxiliar todos os *stakeholders* na análise de dados de engajamento e demografia, sendo extremamente populares pois criam uma interface amigável com os usuários e são expressivas. No entanto, apresentam problemas de interoperabilidade na separação de conceitos, exigindo bastante conhecimento prévio do desenvolvedor.

Por outro lado, existem trabalhos acadêmicos com foco na interoperabilidade, expressividade e até na separação de conceitos, mas que pecam pela falta de interface com o usuário e também pela pouca extensibilidade.

Por fim, existem algumas soluções baseadas na visualização dos estados do jogo que são mais amigáveis para os usuários e que utilizam teorias de grafos, com bastante sucesso, para facilitar essa visualização. No entanto, pecam também no conhecimento prévio necessário para o desenvolvedor e também na extensibilidade.

O quadro 3 faz um resumo dos prós e contras de cada uma das soluções. O objetivo é somente uma visão geral das soluções para facilitar a comparação e não uma análise extensiva de cada uma delas (o que já foi apresentado anteriomente).

No próximo capítulo, vamos mostrar o GameGuts e como essa solução, focando em ajudar o desenvolvedor na captura, representação e recuperação de dados de gameplay, pode ajudar o desenvolvedor com o conhecimento prévio, sem perda significativa na expressividade e subindo um nível na interface amigável aos humanos com o uso de ferramentas visuais, não só para visualização das análises, mas para busca de dado.

Quadro 3 - Resumo com os prós e contras de cada um dos frameworks para tratamento de dados

Abordagem	Conhecimento prévio	Expressividade	*Separation of concerns*	Interoperabilidade	Amigável ao usuário	Extensibilidade
Flurry	-	+	-	+-	+	-
Memphan	+	+-	+-	+-	-	+-
Chan	+	+	-	+	-	+-
Play-Graph	-	+-	-	+-	+	-

Fonte: o autor

5. O FRAMEWORK GAMEGUTS

Este capítulo apresenta uma nova abordagem para o registro de partidas de jogos digitais *online* chamado *GameGuts*.

5.1 INTRODUÇÃO

As soluções disponíveis para captura, representação e recuperação de dados em jogos, apresentadas no capítulo 4, suportam dados de *gameplay* mas possuem diferentes objetivos:

- Auxiliar *stakeholders* na análise de dados de engajamento e *gameplay* para os *frameworks* comerciais;
- Análise de dados de *gameplay* espaciais nas soluções baseadas em estado; e
- Representação de conhecimento sobre partidas e análise de regras nas soluções baseadas em ontologia.

Este capítulo apresenta *GameGuts Framework* (GG), solução com um foco diferente: facilitar o tratamento dos dados de *gameplay* pelo desenvolvedor.

O foco no desenvolvedor justifica decisões de projeto, diferentes das outras soluções, que serão discutidas sob a perspectiva das propriedades apresentadas no capítulo anterior: conhecimento prévio, expressividade, separação de conceitos, interoperabilidade, apresentação amigável para o usuário e extensibilidade. Essas decisões formam a base para uma série de processos, ferramentas e *guidelines* que são mencionados e detalhados nas seções seguintes.

Durante esse capítulo, em conjunto com a apresentação do GG, utilizamos o jogo *PAC-MAN* (escolhido devido à sua popularidade), como exemplo para a clareza da apresentação.

5.2 ABORDAGEM DE DESENVOLVIMENTO DO *FRAMEWORK* GG

A criação de um novo *framework* para captura, representação e recuperação de dados de sessões de jogos envolve todas as dificuldades discutidas no capítulo 2. Uma abordagem tradicional baseada no modelo *Waterfall* [73] demandaria um tempo vasto de modelagem e documentação do *GG* e inviabilizaria o uso em um ambiente comercial conforme demonstrado

pelos estudos de caso apresentados no próximo capítulo. Dessa maneira, *GG* utiliza uma abordagem de desenvolvimento de software ágil, baseada nos conceitos de *Lean Startup* [74]. Ou seja, um produto mínimo viável (do inglês *Minimum Viable Product* ou MVP) com as funcionalidades essenciais do sistema foi criado para os jogos, validado e evoluído. A versão aqui apresentada é o resultado da quarta iteração.

De acordo com o *Lean Startup*, existem duas vantagens no projeto, implementação e evolução de um MVP no lugar da criação de um projeto "completo/perfeito": (1) testes podem ser feitos bem no começo do desenvolvimento, facilitando ajustes ou mudanças significativas antes do desenvolvedor se comprometer demais com o sistema e (2) o foco do projeto é maior em funcionalidades que efetivamente criam valor para o usuário, sendo assim mais produtivo.

A criação e utilização do MVP em um primeiro jogo, possibilitou evoluir a solução para os jogos seguintes. As diversas iterações feitas por GG levaram ao entendimento de que era necessário ajudar ainda mais o desenvolvedor em alguns pontos. Mais detalhes desse processo estão descritos no próximo capítulo junto com os estudos de caso que guiaram todo o desenvolvimento.

Cabe salientar que, ao seguir o processo sugerido pelo *Lean Startup*, *GG* evoluiu para ajudar o desenvolvedor e não ao contrário. As iterações beneficiaram decisões de projeto com foco em conhecimento prévio, expressividade e interoperabilidade, em detrimento da separação de conceitos e de ser "amigável ao usuário". A extensibilidade foi tratada como um híbrido de soluções comerciais que utilizam *APIs* e das soluções baseadas em sistemas de produção.

Quadro 4 - Pontos fortes e fracos de cada um dos frameworks

Abordagem	**Conhecimento prévio**	**Expressividade**	***Separation of concerns***	**Interoperabilidade**	**Amigável ao usuário**	**Extensibilidade**
Flurry	-	+	-	+-	+	-
Memphan	+	+-	+-	+-	-	+-
Chan	+	+	-	+	-	+-
Play-Graph	-	+-	-	+-	+	-
GG	+	+	-	+	+-	+-

Fonte: o autor

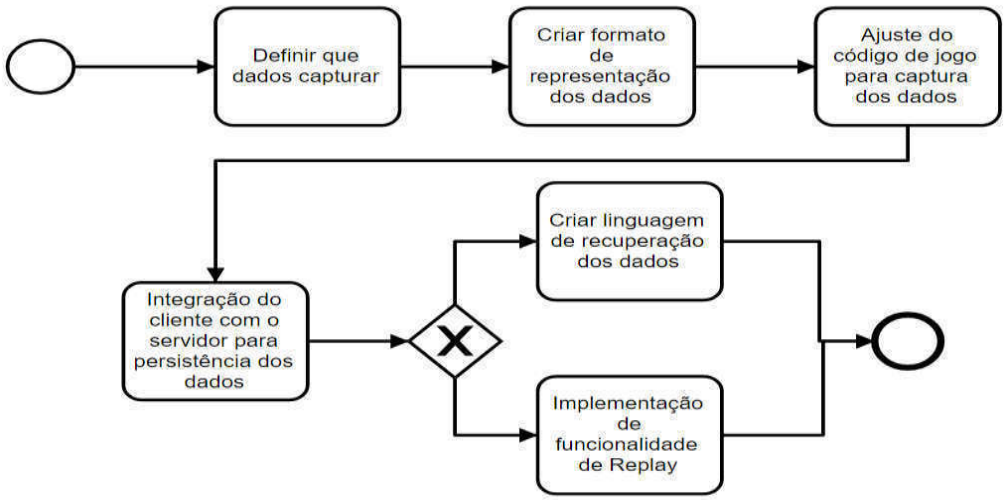
5.3 ADOTANDO O *GAMEGUTS*

Para a adoção de *GameGuts* em um novo jogo, é necessária a execução de seis passos macro. Cada um desses passos possui processos próprios para a criação de ferramentas, *SDKs* e componentes de código para o jogo em questão. São eles:

- Definir que dados capturar
- Criar formato de representação dos dados
- Ajustar o código do jogo para captura de dados
- Integrar o cliente com o servidor para a persistência dos dados
- Criar linguagem de recuperação dos dados
- Implementar a funcionalidade de Replay

O processo deve ser feito de forma linear nos quatro primeiros passos. Os últimos dois passos são independentes e podem ser feitos em paralelo. Uma representação em *BMPN* é mostrada na figura abaixo.

Figura 11 - Processo utilizado para a adoção de GG para um novo jogo em desenvolvimento



Fonte: o autor

As seções a seguir detalham cada um dos passos, bem como suas decisões de projeto e explicam o diferente posicionamento do *framework* GG em relação aos *frameworks* existentes naquilo que é disponibilizado para os desenvolvedores.

5.4 COMPONENTES *GAMEGUTS*: GG INSTANCE

GG Instance é o componente de *GG* responsável pelos primeiros dois passos na adoção de GG:

(1). Definir que dados capturar; e

(2). Criar formato de representação dos dados.

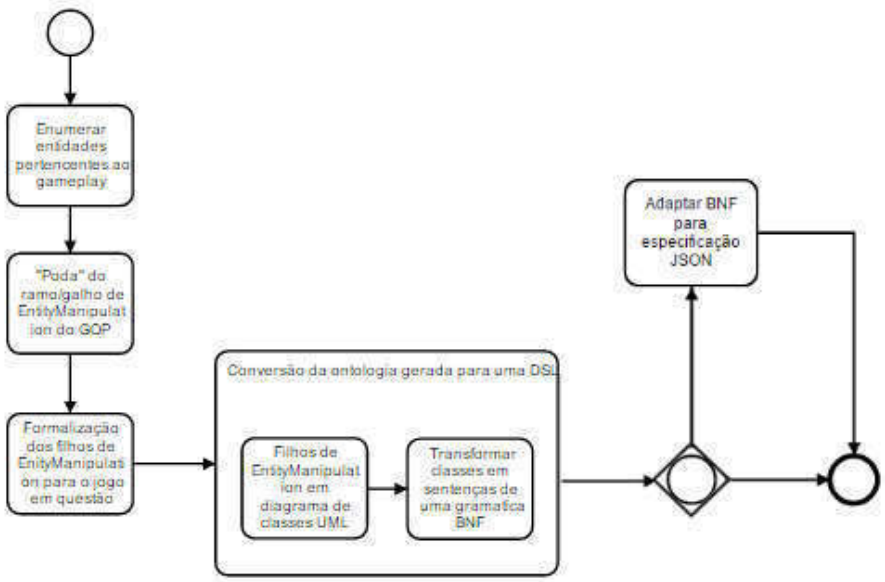
Esses dois passos são feitos em um único processo, apresentado na próxima seção, que visa diminuir a necessidade de conhecimento prévio e provê expressividade na representação dos dados de *gameplay*. As seções a seguir apresentam o processo e discutem como o primeiro passo trata a dependência de conhecimento prévio por parte dos *stakeholders* e como o segundo passo trabalha a expressividade e interoperabilidade.

5.4.1 Conhecimento prévio necessário: definindo que dados capturar

Sempre que um dado essencial não é capturado devido a um erro de projeto do analista de dados, em geral, pela falta de experiência do mesmo, isso resulta em custos para criar uma nova versão do jogo, redistribuir para milhares de jogadores e esperar uma quantidade relevante de dados para poder refazer a análise. Para reduzir as chances de erro durante essa etapa, diminuindo a necessidade de conhecimento prévio no passo 1 (Definir que dados capturar) da adoção *GG,* propõe um subprocesso, passo a passo, chamado *GG Instance,* que formaliza a definição de "o que" capturar baseado em uma ontologia, o *Game Ontology Project* (GOP).

O *GG Instance*, diagrama *BPMN* na Figura 12, faz uso de ontologias para padronizar a definição de que dados capturar, amparando o desenvolvedor e, consequentemente, diminuindo a necessidade de conhecimento prévio.

Figura 12 - Processo para geração de formato de representação de partidas de um dado jogo com GG no formato BPMN



Fonte: o autor

Ontologias diminuem a necessidade de conhecimento prévio pois representam conceitos envolvidos em um domínio de conhecimento [75]. O *Game Ontology Project*, ou GOP, pode ser utilizado como ponto de partida de novas representações, uma vez que contém conceitos para a representação de vários tipos de dados de jogos.

O GOP é composto de várias ramificações que tratam dos diferentes tipos de dados de jogos. A parte de *EntityManipulation* é de particular importância para o GG pois se preocupa com a definição dos conceitos de sessão de jogo, ou seja, dados de *gameplay*.

Sendo assim, os dois primeiros passos para definição do que capturar são: Enumeração das entidades pertencentes ao gameplay e "poda" do ramo/galho de *EntityManipulation* do GOP.

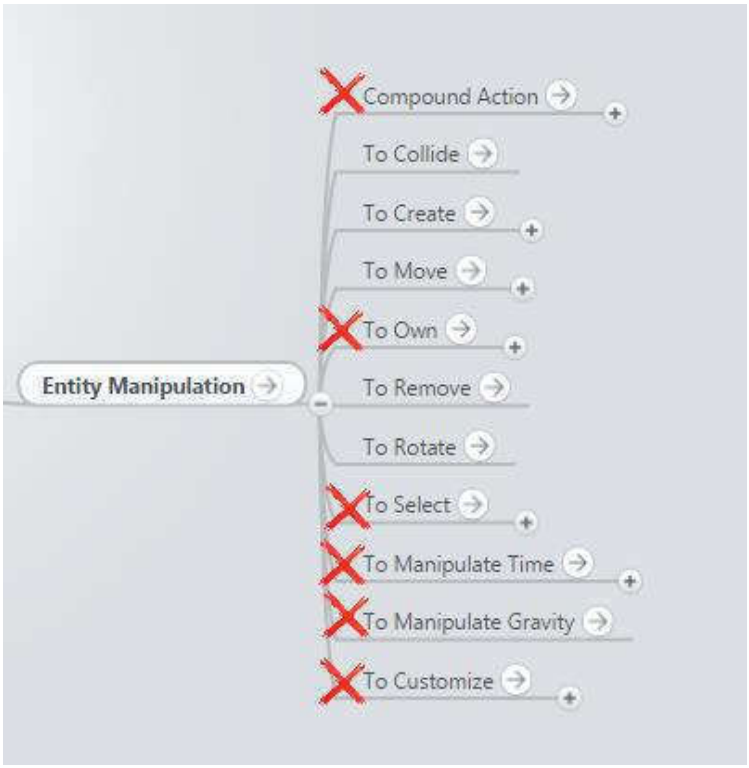
a) Enumerar todas as entidades envolvidas no jogo, assim como as variáveis de estado e as propriedades de cada uma dessas entidades. Usualmente, o documento de *game design* contém essas informações em detalhes suficientes. No jogo *PAC-MAN*, por exemplo, as entidades são o próprio *PAC-MAN*, os *pellets* (pequenos pontos no mapa que o *PAC-MAN* come no caminho), as frutas que lhe

dão pontos, os *pellets* maiores que lhe dão a habilidade de comer os fantasmas conhecidos como: *Inky*, *Blinky*, *Pinky* e *Clyde*. Essas entidades possuem atributos. O *PAC-MAN, por exemplo,* tem como variáveis de estado a sua posição e se está sob ação de um "*pellet* grande" ou não e como propriedades tem sua velocidade.

b) Identificar que partes do subconjunto de *Entity Manipulation* do GOP estão relacionadas com cada uma das entidades. A *Entity Manipulation* é a parte do GOP que representa as ações executadas por, em ou com uma dada unidade. Cada uma das manipulações é avaliada se pertence ao conjunto de ações de gameplay de um cada um dos elementos da lista criada no passo 1. O personagem *PAC-MAN*, por exemplo, pode se mover (*ToMove*) e colidir com outras entidades (*ToCollide*). Uma colisão entre o *PAC-MAN* e o *Blinky* pode ser representada como *ToCollide*(*PAC-MAN, Blinky*) ou *ToCollide*(*Pinky*, *PAC-MAN*). *ToMove* também é usada nos fantasmas, pois os mesmos se movem. *Pellets* e frutas, por outro lado, não se movem mas são criadas (*ToCreate*) e também colidem com outras entidades (*ToCollide)*.

A poda determina as partes do GOP relacionadas ao jogo em questão. Em *PAC-MAN* não é necessário representar a manipulação de entidade *ToManipulateGravity*, por exemplo. E como o *ToManipulateGravity* existem vários outros. Esse passo deve garantir que só as manipulações necessárias são usadas, criando o balanço entre expressividade e interoperabilidade. A figura 13 mostra a poda executada para o jogo *PAC-MAN*.

Figura 13 - Pod a de filhos de EntityManipulation para o PAC-MAN



Fonte: o autor

5.4.2 Expressividade e interoperabilidade: Criando o formato de representação de dados

O suporte nas ferramentas comercias, como *Flurry*, entrega ao desenvolvedor um grande potencial de expressividade. Na prática, no entanto, falta padronização (formalização), comprometendo esse potencial. Como discutido no capítulo anterior na seção de Flurry, a ação de coletar uma fruta, por exemplo, pode ser chamada de *ToCollect* ou apenas *Collect*. Ou pior, em diferentes partes do jogo a ação pode ter denominações diferentes.

O GOP inclui diversos conceitos, pois tenta representar todos os jogos e não dá atenção especial à formalização dos mesmos - criando potenciais problemas de interoperabilidade. Como exemplificação, no capítulo de estado da arte em *PAC-MAN*, a manipulação de entidade *ToCollect*, só é usada entre as entidades *PAC-MAN* e os *pellets* ou as frutas (o *PAC-MAN* não coleta fantasmas). Ou seja, é necessário qualificar a manipulação com parâmetros. Os parâmetros indicam a cardinalidade da manipulação e também os tipos envolvidos. No exemplo

anterior seria sempre *PAC-MAN* e *pellets*. Ou seja, teriamos algo como *ToCollect*(x: *PAC-MAN*, y : *Pellet*).

A movimentação do personagem no jogo é outro exemplo. O *PAC-MAN* pode se movimentar pelo cenário. Essa ação pode ser representada pelo termo *ToMove* do GOP que é definida como:

> *Entidades possuem a habilidade de mudar de posição. Jogadores podem enviar um comando de movimento de diversas maneiras diferentes. Normalmente, esse comando é mapeado diretamente em um dispositivo de entrada no qual existe uma relação um para um entre a entrada do jogador e como a entidade contralada se move. Outra maneira comum de entrada de movimentos é através de uma interface point-and-click. Nessa interface, o jogador especifica onde a entidade deve se mover. A entidade, por sua vez, utiliza de técnicas de busca em caminhos para achar a melhor maneira de chegar no local desejado. Entidades controlados por inteligência artificial normalmente usamos técnicas similares de busca para mover para uma localização desejada.* [76] (tradução do autor)

No entanto, novamente, por ter como objetivo representar qualquer conhecimento relacionado a todos os jogos, não padroniza detalhes e especificidades. Para o registro de um movimento, com o *ToMove*, existem diversas possibilidades de representação que devem ser levadas em conta: usar um vetor de velocidade, usar a direção da movimentação ou usar a nova posição do personagem, são alguns exemplos. Esses detalhes não são especificados em GOP e essa padronização se faz necessária para evitar problemas de interoperabilidade como descrito no capítulo anterior.

Para contornar esse problema, *GG Instance* sugere duas ações para adaptação além da poda: formalização e transformação. Esses passos resultam em um formato de representação que favorece a interoperabilidade, um formato *computer-friendly*. Essa parte do *GG Instance* compreende o segundo passo da instanciação de *GG*: criar formato de representação de dados e que é feito de acordo com os passos a seguir, após a poda do *EntityManipulation*.

c) Adaptação de cada um dos elementos de manipulação de entidade na ontologia, para prover uma representação mais precisa em termos de cardinalidade e domínio dos parâmetros e variáveis.

No elemento de manipulação de entidade que representa uma colisão *ToCollide*, por exemplo, informações importantes como tempo e ponto de colisão não são definidas. Dessa maneira, para o *ToCollide*, novas propriedades são necessárias para uma análise mais precisa e rica dos dados. Em geral, a posição e o tempo em que a colisão ocorreu são de extremautilidade.

Um registro de colisão poderia então ser registrado como *ToCollide* (*PAC-MAN*, *Blinky*, (27,32), 5674 onde (27,32) são as coordenadas na tela (em pixel ou tiles, por exemplo) de onde a colisão ocorreu e 5764 é um *timestamp* do momento da colisão que pode ser representado em segundos ou *loops* de jogo (para a criação de *replays* de uma partida, por exemplo, a representação em *loops* é mais apropriada).

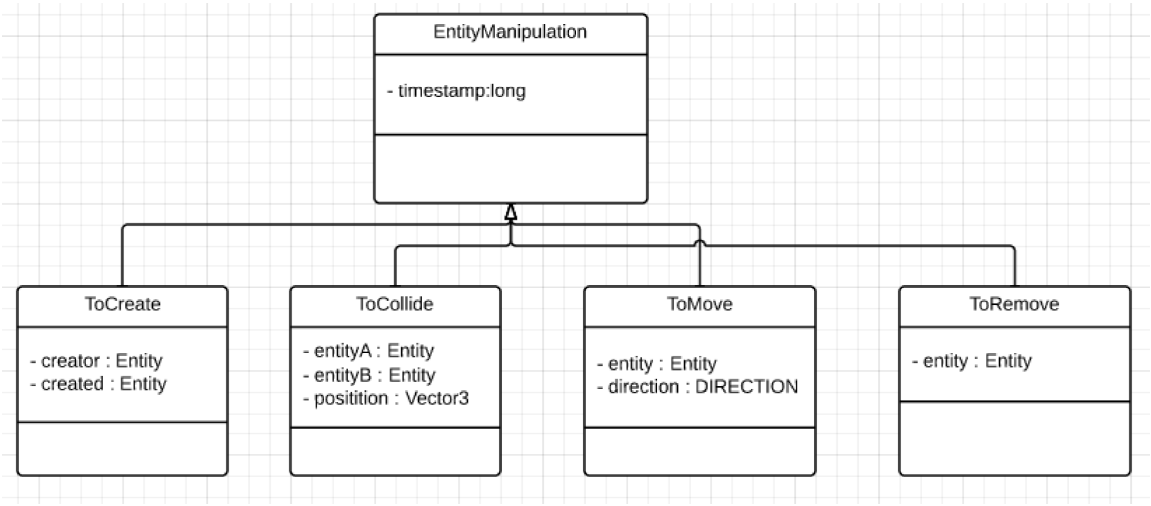
Para o *PAC-MAN*, por simplicidade, direções funcionam bem para a movimentação e ir para a esquerda do fantasma *Clyde*, por exemplo, pode ser representada como *ToMove* (*Clyde*, *LEFT*, 676) onde, 676 é novamente o *timestamp*.

d) Conversão da ontologia para uma *DSL* usando o método de Taira [47].

- O método requer, como primeiro passo, a modelagem do domínio como um diagrama de classes. É um processo extenso e tedioso, mas que terá benefícios para documentação do projeto e preocupações de *separation of concerns* no código do jogo. A figura 15 mostra o diagrama de classes do domínio para o *PAC-MAN*.

- Transformar o diagrama de classes do domínio em uma gramatica na forma de Backus-Naur [77], no inglês Backus-Naur *Form* ou BNF. A gramática na forma BNF para o jogo do *PAC-MAN* discutido nessa seção tem as definições finais abaixo:

Figura 14 - Mapeamento da manipulação de entidados do GOP no jogo clássico Pac-Man.



Fonte: o autor

Pseudo-código 9 - BNF para representação de jogadas no Pac-Man

ToCreate ::= ToCreate(<entity>, <entity>, <position>, <timestamp>)
ToCollide ::= ToCollide(<entity>, <entity>, <position>, <timestamp>)
ToMove ::= ToMove(<entity>, <direction>, <timestamp>)
ToRemove ::= ToRemove(<entity>, <timestamp>)
<direction> ::= left | right | up | down
<position> :== (<number>, <number>,<number>)
<number> ::= <digit> | <number> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<timestamp> ::= <number>
EntityManipulation ::= ToCreate | ToMove | ToOwn | ToShoot | ToCollide | ToRemove <timestamp>
<entity> ::= MAZE | PACMAN | BLINKY | PINK | ...

Fonte: o autor

e) Refatoração da *DSL* em *BNF* para uma sintaxe *JSON*. A escolha do *JSON* se deve ao fato de ser um formato de troca de dados "popular" que adiciona pouco ou nenhum código de suporte, facilitando o entendimento dos arquivos [78]. A utilização de uma linguagem totalmente nova envolve a implementação de novos parsers e analisadores sintáticos e semânticos. Ou seja, a utilização de um formato *JSON* facilitará na criação e leitura de arquivos no formato. Esse passo é opcional, mas extremamente recomendado. Um trecho de uma partida de *PAC-MAN* gravada em um arquivo *JSON* baseado na *BNF* mostrada no passo anterior seria algo como:

Pseudo-código 10 - Exemplo de registro de jogada do Pac-Man com a linguagem de representação criada.

```
{ "entities":[
    {"id": 0, "type": "game.pacman.enemies.Inky"},
    {"id": 1, "type":"game.pacman.PacMan"},
    {"id": 2, "type":"game.pacman.Pellet"},
    {"id": 3, "type": "game.pacman.Pellet"},
    {"id": 4, "type": "game.pacman.Pellet"},
    {"id": 5, "type": "game.pacman.Maze"},
    {"id": 6, "type": "game.pacman.Fruit"}, ...],
  "EntityManipulations":{
    "ToCreate":{ "creator":5, "created":1, "timestamp":403},
    "ToMove":{ "moving":1, "direction":"left", "timestamp":1055},
    "ToMove":{ " moving ":1, "direction":"left", "timestamp":1146},
    "ToCreate":{"creator":5, "created":0, "timestamp":1507 }, ...
    "ToCollide":{"entity":1, "entity":6, "timestamp":7200},
    "ToMove":{ " moving ":1, "direction":"left", "timestamp":7202},
    "ToMove":{ " moving ":1, "direction":"left", "timestamp":7450},
    "ToCollide":{"entity": 0, "entity": 1, "timestamp":7580},
    "ToRemove":{"entity":0, "timestamp":7803}
  }}
```

Fonte: o autor

5.5 COMPONENTES *GAMEGUTS: GG SERVER*

Os passos seguintes na adoção de *GG*, (Ajuste do código de jogo para captura dos dados e Integração do cliente com o servidor para persistência dos dados) são tratados por um componente chamado *GG Server* que utiliza uma arquitetura cliente-servidor inspirada em ferramentas como o *Flurry,* incorporando novos componentes baseado em sistemas de regra de produção, como *triggers*.

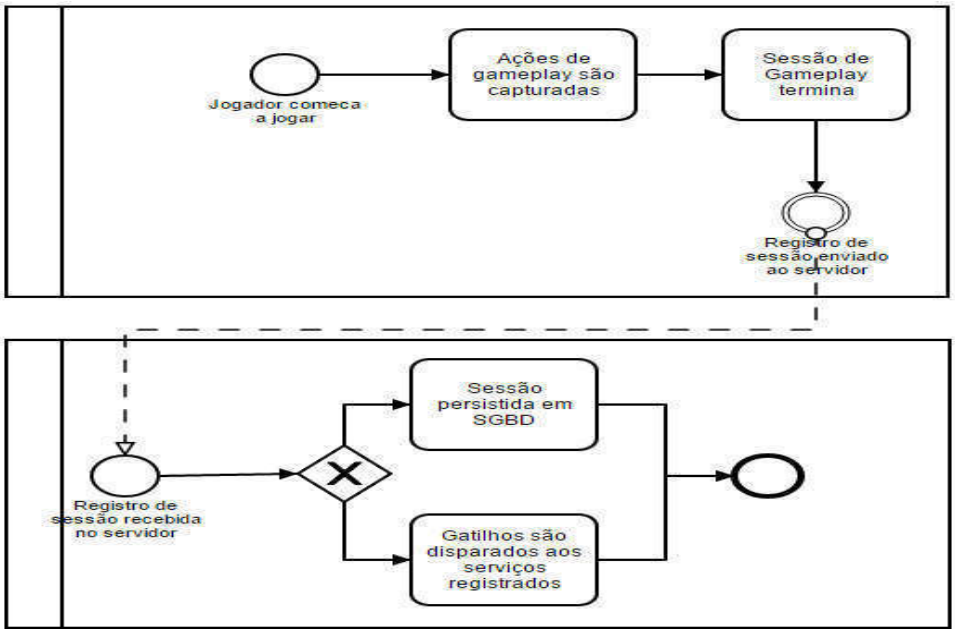
Para facilitar o entendimento da arquitetura, é importante entender como normalmente são feitas a captura, a representação e a posterior análise dos dados. Comumente, o fluxo dos dados acontece da seguinte forma, diagramado na figura 16, (mais detalhes nas seções posteriores):

- O jogador abre o jogo em seu dispositivo e começa a jogar.
- O cliente do jogo captura as ações do jogador localmente.
- Ao final da sessão, a sessão inteira é enviada ao servidor pelo cliente.

- O servidor dispara os *triggers* registrados e envia a sessão, ou pedaços dela, para as extensões e sistemas que criaram os *triggers*
- A sessão é guardada no banco de dados.
- A sessão, a partir de agora, fica disponível para acesso por outros sistemas.
- Os *stakeholders* podem agora requerer informações como, por exemplo, qual o lugar em que os jogadores morrem com mais frequência no mapa.

Nas subseções a seguir apresentamos sugestões de GG com relação à arquitetura de servidores e como é possível melhorar extensibilidade e interoperabilidade, a partir de abordagens diferentes do estado da arte.

Figura 15 - Processo usual de captura e persistência de dados de gameplay com o GG em formato BPN

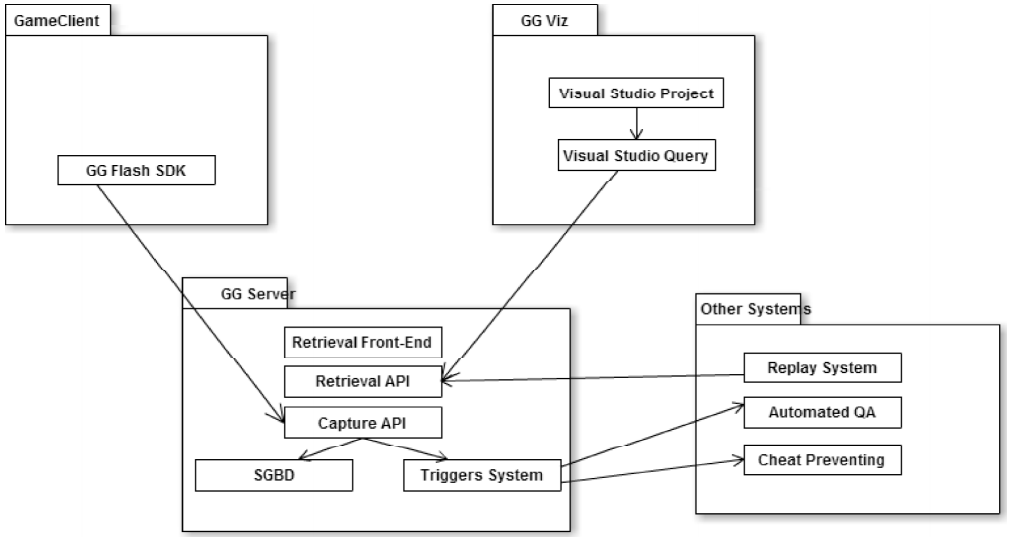


Fonte: o autor

5.5.1 Interoperabilidade GG Server

A divisão geral dos módulos do *GG Server* é descrita na figura a seguir:

Figura 16 - Visão geral da arquitetura de GG e os componentes envolvidos



Fonte: o autor

1. Captura: *Capture API*

Responsável pelas ações dos três primeiros passos no fluxo apresentado na Figura 16: (1) ações de gameplay são capturadas e (2) Registro de sessão enviado ao servidor.

O componente é formado por uma *API* no servidor que recebe os dados do *Software Development Kit* (SDK) no cliente. Essa formação é utilizada por diversas soluções comerciais e *GG* segue a mesma abordagem.

A parte da *API* responsável pelo registro de sessões pelo cliente, a *Capture SDK*, é bem simples e composta por uma única função que recebe um arquivo *JSON* com a representação da sessão. Essa função utiliza o protocolo *HTTP* padrão em *APIs REST* e o *JSON* é enviado como parâmetro.

Visando interoperabilidade, essa abordagem difere um pouco da comumente utilizada que éenviar um evento por vez ao servidor. Devido à quantidade de dados, o *overhead* dos cabeçalhos de rede e uma grande quantidade de acessos via rede 3G móvel, optamos por enviar somente uma mensagem ao final da sessão de jogo com a jogada completa.

O design do *Capture SDK* no cliente e das classes do jogo impactam na separação de conceitos e é discutido no final do capítulo. Infelizmente, como discutiremos adiante, esse

problema é intrínseco às linguagens orientadas a objetos e a solução fica fora do escopo do *GG*. Cada linguagem pode se utilizar de diferentes funcionalidades para atacar o problema. No entanto, ao final do capítulo uma seção é dedicada a ajudar o desenvolvedor a trabalhar o problema.

2. Armazenamento: SGBD

Após a chegada do registro no servidor, ele precisa ser pré-processado e ter seus dados guardados em formatos apropriados para diferentes usos. Importante lembrar que, em jogos comerciais com relativo sucesso, a quantidade de usuários diários supera os milhões. Sendo assim, a arquitetura e design de código do servidor é de extrema importância. Felizmente, esse problema tem soluções de código fechado e *open source* disponíveis. Ou seja, pode ser feita por um dos *frameworks* já existentes, como o *Flurry*, ou implementada usando tecnologias de servidor. No GG, as duas opções foram usadas.

A criação de um servidor novo, usando a tecnologia *Google App Engine* e um banco de dados não-relacional (*NoSQL*[79]), facilita a criação dos outros dois componentes, mas tem um esforço enorme de implementação para garantir todas as funcionalidades de um *framework* já consolidado como o Flurry - principalmente para captura e visualização de dados que não são de *gameplay*, especialmente demográficos e de software.

A segunda opção, de uso de um *framework* existente, mostrou-se muito eficiente e menos custosa. Sendo assim, devido ao baixo impacto da contribuição de GG nesse aspecto, decidimos por não abordar esse componente neste trabalho.

3. Recuperação: *Retrieval API*

Enquanto a *Capture API* guarda os registros, a *Retrieval API* possibilita a recuperação dos dados para análise por componentes externos. Por se tratar de um componente de software tradicional, a equipe utilizou um processo padrão de levantamento de requisitos com casos de uso e posterior implementação. O processo é detalhado mais à frente no capítulo.

4. Recuperação: Trigger System

Como a análise usualmente envolve uma massa de dados muito grande, a possibilidade de análise por algoritmos é crucial para um entendimento relevante. A possibilidade de otimização

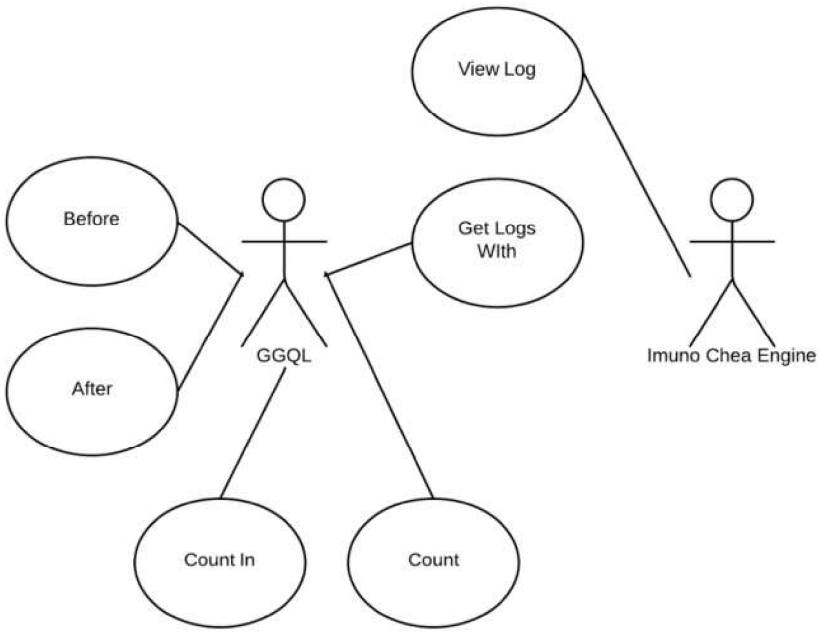
e extensão das análises de uma maneira simples e sem necessidade de compilação do servidor é de grande valia.

Para isso, GG se vale de aspectos de sistemas baseados em regras que não seguem a abordagem usual de orientação a objetos. Para simular tais sistemas, um componente de triggers deve ser criado e adicionado ao servidor como explicado adiante.

5.5.2 Extensibilidade: *Retrieval API*

Retrieval API uma *API Rest* para disponibilizar os dados a outros sistemas. Para esse componente foi feito um processo usual de levantamento de requisitos com casos de uso. As funções estão representadas no diagrama de casos de uso da Figura 17.

Figura 17 Diagrama de casos de uso do componente de recuperação de registros com dois usuários figurativos.



Fonte: o autor

Após algumas iterações na construção de GG, identificamos que são recomendados, pelo menos, os seguintes casos de uso (diagramados na Figura 17) para a *API* de recuperação de registro:

a) *View Log*: Recupera o registro inteiro de uma dada sessão a partir de um identificador único da partida. O identificador é normalmente recuperado nas outras funções da API abaixo. Exemplo: Registro de uma partida de um dado jogador em um certo dia.
b) *Get Logs With*: Recupera os registros de partida que contenham eventos ligados a uma ou mais manipulações de entidades. Exemplo: Todos os registros no qual existiu uma colisão entre uma bala tripla do jogador e um inimigo do tipo X.
c) *Count*: Conta a quantidade de registros que possuem uma ou mais manipulações de entidades. Exemplo: Em quantos registros a pistola Y foi utilizada?
d) *Count in*: Conta a quantidade de vezes em que uma ou mais manipulações de entidades ocorreram em um mesmo registro de partida. Exemplo: Quantas vidas o jogador perdeu em uma partida?
e) *Before*: Lista as manipulações de entidade que acontecem antes de uma ou mais manipulações de entidade. Exemplo: O que os jogadores fazem antes de atirar com a pistola X?
f) *After*: Lista as manipulações de entidade que acontecem depois de uma ou mais manipulações de entidade. Exemplo: O que os jogadores fazem depois do inimigo atirar com a pistola X?

Importante salientar que as buscas por registros relacionados a uma certa manipulação de entidade (como o *Get Logs With* ou o *Count*), podem receber uma lista de eventos. A busca pode ser feita por todos os eventos (*AND*) ou por pelo menos por um dos eventos (*OU*) ou uma composição.

As buscas também devem suportar atributos auxiliares de apoio, baseadas nas iterações executadas nos estudos de casos feitos. Definimos como necessidade: a versão do jogo, a data em que os registros foram feitos por jogador (no *count*, por exemplo, quantos jogadores utilizaram a pistola).

Para a otimização de recuperações frequentes e também no caso de uso não listados, as chamadas podem ser indicadas como persistentes. Dessa maneira, ficam guardadas no servidor e são otimizadas para serem utilizadas novamente no futuro. A otimização é feita a partir da criação de *listeners,* usando o componente descrito a seguir.

5.5.3 Extensibilidade: *Trigger System*

Usada por componentes externos ou scripts para receberem “avisos” de eventos específicos, no momento em que são enviados pelo cliente. Esse componente “imita” uma base de regras e visa facilitar a extensão das análises de dados.

O mecanismo de regras idênticas às regras de produção bem conhecidas em inteligência artificial [80], usado em soluções baseadas em ontologia, facilita a extensão das análises. Nos sistemas de produção, a adição de uma nova regra não demanda modificações nas demais já que cada uma é uma unidade autônoma - facilitando a extensibilidade. Além disso, uma vez que uma nova regra é inserida no sistema, ela está imediatamente habilitada a “disparar” se os fatos validam suas premissas. Dessa maneira, esta solução é bastante extensível, já que novas análises podem ser facilmente adicionadas ao projeto e serão feitas sempre que uma sessão possuir características definidas na regra.

Os *listeners* são similares às condições que disparam regras em um sistema como o *Drools* [81]. A cada registro enviado pelo cliente, o servidor de GG lê o registro inteiro e dispara eventos para *listeners* de cada uma das manipulações de entidade contidas no registro. Cada um dos *listeners* recebe o registro e pode transformá-lo em um formato mais adequado aos seus propósitos.

O caso de uso Count in, por exemplo, pode ter uma instância que deve contar os registros contendo a manipulação de entidade ToOwn. Nesse caso, o listener é criado e sempre que algum registro com ToOwn é enviado ao servidor, este é, em seguida, enviado para o *listener*. Esse *listener* atualiza um contador no banco de dados criado em seu registro. Quando uma nova chamada é feita, o servidor não precisa contar todos os registros novamente, somente acessar a variável que foi incrementada quando um novo registro chegou ao servidor.

Na prática, esse mecanismo foi implementado em JAVA com a criação de um formato *JSON* que indica quais manipulações de dados o *listener* deseja “escutar” e uma série de classes compiladas (dentro de um arquivo .jar) responsáveis por receber o evento. O *JSON* também deve indicar qual é a classe principal do pacote que implementa uma interface pré-definida por GG, o *GGListener*. Importante salientar que, devido à padronização na representação dos dados

de *gameplay*, esse componente pode ser reutilizado em todos os jogos, ao contrário da representação que é instanciada para cada novo jogo.

Um exemplo do JSON e o código da interface estão abaixo:

Pseudo-código 11 - Json de configuração do servidor GG

```
{
  "listener" : "com.bighutgames.BoneyMonkeyListener",
  "listenTo" : {"entityManipulation" : "ToOwn", "params" : ["player",
"mummy_monkey"] },
  "gameVersion" : 3.0,
},
{
  "listener" : "com.bighutgames.DoubleJumpCounterListener",
  "listenTo" : {"Before" : "ToJump", "params" : ["player"] },
  "gameVersion" : 2.3,
```

Fonte: o autor

Pseudo-código 12 - Código da interface do listener de GG

```
}
public interface GGListener
{
  public void registerLog(JSON log);
}
```

Fonte: o autor

5.5.4 Separação de conceitos: GG SDK

O estudo do estado da arte apresentado evidencia que soluções comerciais voltadas para *analytics frameworks* mostram pouca ou nenhuma preocupação no lado cliente da aplicação, no jogo em si. Mais ainda, os *frameworks* comerciais, para facilitar o uso em escala, forçam o desenvolvedor a se adaptar ao padrão de codificação do *framework*. Além da necessidade de adaptação ao padrão do framework, problemas de separação de conceitos dificultam a manutenção do código do jogo, como discutido no capítulo anterior. A separação de conceitos é de solução complexa pois é um problema inerente às linguagens orientadas a objeto.

Por outro lado, as soluções baseadas em ontologia utilizam, em geral, um paradigma de programação declarativo para as análises - normalmente um sistema baseado em regras como

o *Drools* [27] - evitando problemas na separação de conceitos que são inerentes a linguagens orientadas a objetos. No entanto, para o uso comercial em larga escala em jogos, o uso de um paradigma que não seja imperativo de orientação a objetos, não é viável economicamente, pela falta de ferramentas de suporte ao desenvolvimento como *Engines* de jogos e Ambientes de Desenvolvimento (*IDEs*).

A solução indicada para esse problema em linguagens orientadas a objetos seria a utilização de aspectos [82]. No entanto, as ferramentas disponíveis para o uso de aspectos em projetos comerciais de jogos de larga escala também são deficientes [83], levando em consideração que a maioria dos jogos são feitos em *Actions Script,* C# ou C++.

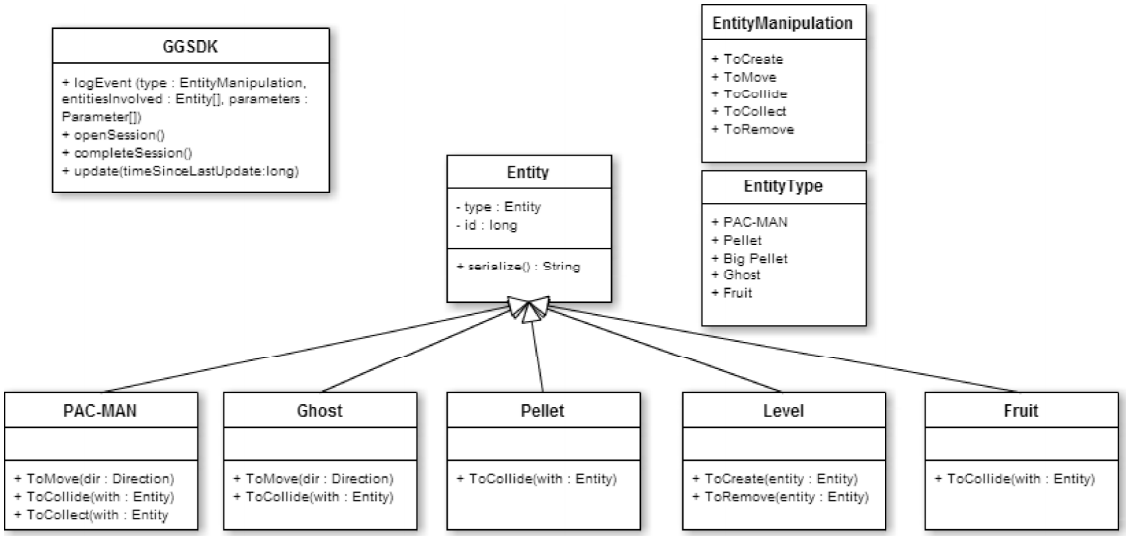
GG, não resolve o problema de separação de conceitos. No entanto, sugerimos a seguir uma estrutura de classes para potencialmente evitar o problema de separação de conceitos por completo nas linguagens com suporte a classes parciais e minimizar nas demais.

O *SDK* deve ser criado da seguinte forma:

a) A partir da lista de entidades criadas no *GG Instance*, criar uma classe para cada uma delas. Essas classes devem se estender de uma classe comum básica *Entity,* que conterá uma parte do código de captura para separação do código de lógica de jogo e que será implementado na classe filha.
b) A partir da mesma lista, criar uma enumeração *EntityType* com cada um dos tipos como um valor.
c) Cada uma das manipulações de entidade relacionadas com a entidade deve ser transformada em método da classe.
d) Criar uma outra enumeração *EntityManipulation* com cada uma das manipulações como valor, para facilitar a identificação dos tipos.

Para o *PAC-MAN*, por exemplo, teríamos a estrutura de classes representada na Figura 18.

Figura 18 - Diagrama com as classes sugeridas para o jogo PAC-MAN



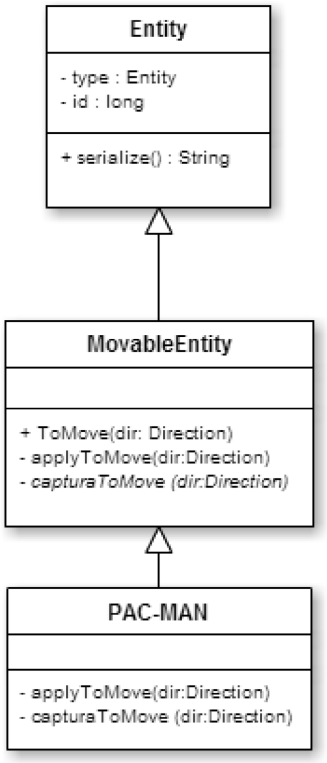
Fonte: o autor

Além das classes resultantes do processo e, particulares de cada jogo, uma classe *GGSDK* faz o trabalho de captura e envio dos eventos para o servidor. A captura deve ser inserida na função referente a cada manipulação de entidade. com uma chamada à função *logEvent* do *GGSDK. O* envio do registro para o servidor é feito usando o *GGSDK* via *Capture API.*

Em um ambiente que utiliza a linguagem C#, por exemplo, podemos utilizar o artifício de *partial classes* no *GGSDK* e implementar diferentes aspectos, como a captura do registro, em diferentes partes da classe. Essa abordagem, junto com a estrutura de classes sugerida, é suficiente para remover o *cross-cutting* [84]. Dessa maneira, cada uma das classes pode ter o código de captura efetivamente separado em uma classe parcial responsável por isso.

No caso de tecnologias que não possuem suporte a *partial classes* é possível diminuir os impactos com o uso do padrão de projeto [85] *Template Method* [39]. Nesse padrão, um esqueleto de algoritmo é usado no método de uma classe abstrata e, nas extensões, é necessário implementar os métodos privados. Na figura 19, um exemplo de como seria o uso no GG*SDK* com a classe *Entity* e *PAC-MAN*.

Figura 19 - Diagrama de classes com a estrutura para separação do aspecto de captura do de gameplay em entidades que se movem.



Fonte: o autor

5.6 COMPONENTES *GAMEGUTS*: GG VIZ

A dicotomia entre amigável ao usuário e também aos computadores pode ser superada com uma divisão clara da linguagem de representação dos dados e da linguagem de apresentação dos dados. Dessa maneira, para complementar o *GG Instance*, focado na criação de uma representação de dados de *gameplay computer-friendly*, o *GG Server*, série de *APIs* e *SDKs* para armazenamento e recuperação dos dados de *gameplay,* o *GG* sugere um componente adicional para visualização dos dados por humanos denominado *GG Viz*.

GG Viz é o componente de *GG* responsável por um dos últimos passos na adoção de *GG*: criar linguagem de recuperação de dados. Nesse passo, um processo cria uma linguagem visual para recuperação de dados de *gameplay*. Como veremos nas seções a seguir, esse passo adicional visa diminuir os problemas relacionados à dificuldade de *stakeholders* em geral com a visualização dos dados.

5.6.1 Formato Amigável ao Usuário: Visual *DSL*

Um dos processos do GG *Instance* resulta em uma *DSL* textual. *DSLs* textuais funcionam para a interoperabilidade pois, por serem específicas, facilitam o processamento por computadores, podem ser menos verborrágicas, diminuindo a carga de dados enviada ao servidor, e possuem uma gama de ferramentas disponíveis para facilitar tanto na criação quanto na posterior manipulação.

Diversos *stakeholders* apresentaram dificuldades na recuperação dos dados. A apresentação dos dados de forma de gráfica, como em soluções como *Flurry,* é satisfatória para tratar a apresentação dos dados. *GG* segue o mesmo padrão para apresentação. No entanto, é difícil para o *stakeholder* expressar quais dados gostaria de ver. Para minimizar o problema, *GG* sugere a criação uma linguagem visual, amigável para os usuários não-técnicos, para buscar informações específicas nos dados.

O *GG Viz* - que utiliza a *API* do GG *Server* para recuperar informação - utiliza um processo para criação de uma DSL Visual. Uma DSL Visual representa as buscas, em um formato amigável para usuários, para facilitar o acesso aos dados por parte dos *stakeholders*. O conceito é definir visualmente cenários ou padrões de jogo para então buscar por eles no banco de dados de partidas [86]. *C*omo veremos a seguir, essa abordagem diminui o tempo e esforço necessário para o entendimento dos conceitos por parte dos usuários.

A *API* do servidor descrita na seção anterior possui os instrumentos necessários para recuperar os dados e, nesta seção, definimos como criar a linguagem visual, integrar com o servidor e, finalmente, como usar a linguagem para recuperar informação e apresentá-la ao usuário. A criação da ferramenta segue um processo passo a passo sugerido por GG descrito a seguir. Como no restante do texto, usaremos o jogo *PAC-MAN* como exemplo.

5.6.2 Criando a linguagem de recuperação visual

A linguagem visual de recuperação consiste nos elementos (entidades) de jogos e todas as possíveis relações (manipulações de entidades) entre os mesmos. Esses conceitos já foram devidamente modelados durante a criação da linguagem de representação e estão documentados no diagrama de classes *UML*.

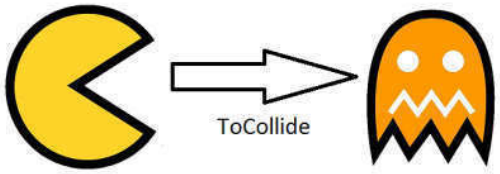
Para facilitar a identificação de cada uma das entidades do jogo na linguagem visual, é recomendável usar o próprio *sprite* do jogo para representar as entidades. A figura 21 mostra as entidades do jogo *PAC-MAN*.

Figura 20 - Entidades do jogo PAC-MAN e sua representação visual para utilização em uma linguagem de recuperação definida por GG.

Fonte: o autor

As manipulações de entidades do *GOP* são modeladas como relacionamento entre as entidades. Com as entidades e as manipulações de entidades representadas visualmente, uma *query* para recuperar registros, em que o *PAC-MAN* colide com um dos fantasmas, é representada como na Figura 21.

Figura 21 - Representação visual de uma “query” para recuperar registros onde o PAC-MAN colide com um dos fantasms do jogo.



Fonte: o autor

Além dos componentes visuais relacionados ao jogo em si, um *query* ainda precisa de algumas informações para facilitar o uso posterior pelo usuário: um nome identificador pelo qual o usuário pode buscar, no servidor, a versão mínima do jogo em que deve ser feita a busca. Esse atributo é muito importante para novos recursos do jogo. Por exemplo, um tipo específico de inimigo que só existe na versão 2.0 do jogo não precisa ser buscado em registros feitos em versões anteriores. Dessa maneira, registros nos quais as manipulações de entidades não ocorrem não serão buscados.

Para os estudos de casos deste trabalho, essa linguagem visual foi criada utilizando a solução da *Microsoft: Visual Studio DSL Tools* [6]. A partir do *Template Minimal Language*, a criação de uma linguagem como essa é bem simples. A linguagem toda é criada com a utilização das *partitions Classes* e *Relations* e a *partition Diagram Elements* é usada para integrar os *sprites* do jogo como elementos da linguagem. O próprio *Visual Studio* é usado para criação das *queries*, como editor.

Cada uma das *queries* criadas é compilada para um programa executável que registra a *query* em um servidor auxiliar. Essa compilação verifica a validade das relações de acordo com a cardinalidade, seu domínio e imagem, como formalizado no GG *Instance*. O executável faz as chamadas necessárias para o componente de recuperação de registros do servidor de GG que depois é utilizado para exibir as informações em diversos tipos de gráficos diferentes (barra, pizza, etc.) de acordo com a preferência dos usuários - seguindo a abordagem padrão do estado da arte.

Normalmente, como na busca de colisão entre *PAC-MAN* e fantasmas, o exemplo da figura 21, o usuário pode querer saber quantas vezes esse padrão aconteceu em todas as sessões ou quantas vezes em média por jogada, por exemplo. São feitas várias chamadas pois cada uma das buscas pode ser vista de várias maneiras diferentes. A seção a seguir entra em detalhes de que visualizações são sempre feitas a partir de uma *query*.

5.6.3 Integração com o servidor

A linguagem visual, ao enviar uma *query* para o servidor, deve fazer uso de todas as possibilidades de otimização, já que esses cenários deverão ser recuperados do servidor diversas vezes. Afinal, esse é o objetivo da linguagem. Quando enviada ao servidor, a *query* é otimizada

com o uso do *Trigger System.* Como uma *query* pode usar várias chamadas do servidor de recuperação, diversos *listeners* são criados para cada *query* e cada uma delas pode ser identificada pelo nome.

Alguns *listeners* essenciais foram identificados para facilitar nas *queries* visuais, otimizando assim o tempo de busca no servidor. São eles:

a) Contador: Conta as ocorrências de todas as possíveis manipulações de entidade no jogo. Ou seja, para cada registro que chega no servidor, um contador para cada manipulação ocorrida (*ToCollide*, *ToCreate*, etc.) é atualizado.

b) Média por sessão: Conta as ocorrências como no anterior mas, em vez de simplesmente adicionar a um contador, ajusta uma média de cada manipulação de entidade por sessão. Esse *listener* facilita a visualização de informações como: quantas vezes em média um jogador colide com o fantasma por jogada?

c) Agrupamento por ambiente: A identificação de que ambiente o jogador está jogando é normalmente muito importante (somente o terceiro estágio, por exemplo). Esse agrupamento é feito de acordo com as manipulações *ToCreate* que indicam quem está criando as entidades (normalmente estágios ou mapas).

d) Agrupamento por tempo: A visualização de manipulação de entidades também é feita com um gráfico de média por tempo na sessão. Cada uma das manipulações é indexada de acordo com o tempo em que aconteceu. Dessa maneira, facilita a visualização de perguntas do tipo: em que momento do estágio ocorre a maioria das mortes dos jogadores?

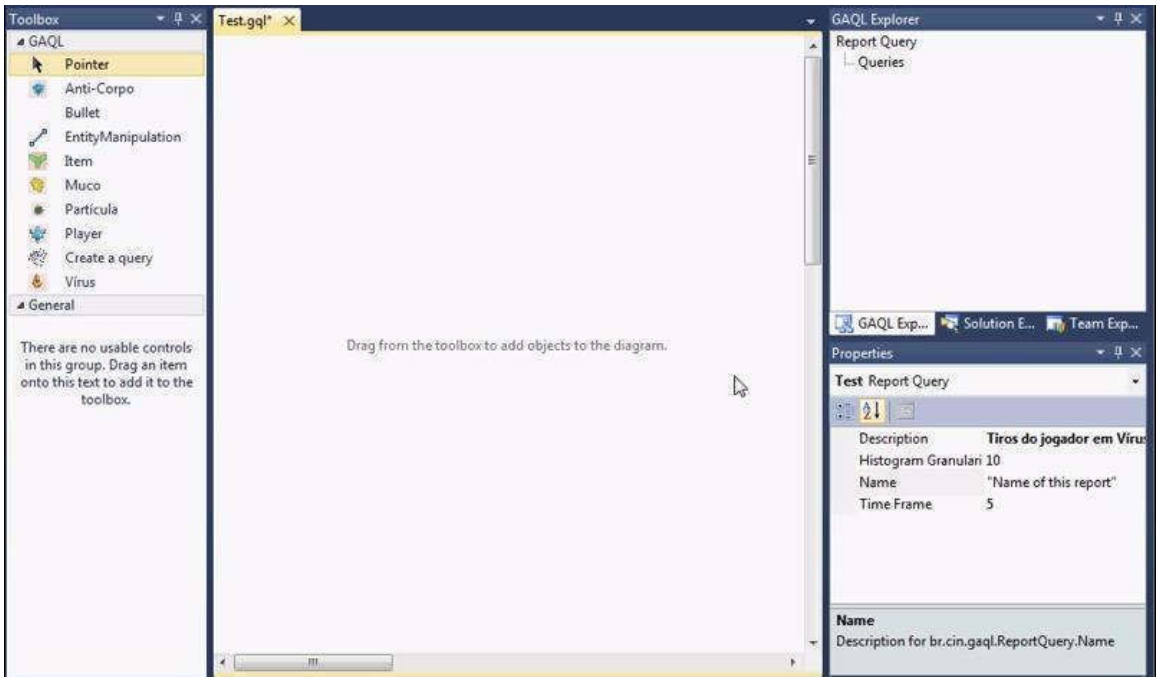
Ações mais complexas podem também ser representadas como uma série de ações. Nesse caso, as manipulações também são agrupadas pelas entidades participantes e na ordem em que acontecem. E o servidor faz várias *queries* utilizando o conjunto de respostas da *query* anterior como entrada para a *query* subsequente.

5.6.4 Criando e visualizando "*queries*"

As *queries* são criadas no *Visual Studio* da *Microsoft*. A criação é feita a partir de um *template*.

A criação no *Visual Studio* é simples e envolve o *drag'n'drop* de entidades e manipulações de entidades de uma lista lateral para a área de busca. A figura 22 contém o início da criação de uma *query* em um dos estudos de caso.

Figura 22 - Ambiente de criação de "queries" no Visual Studio DSL Tools.



Fonte: o autor

Após a criaçao da *query*, o usuário precisa apenas "executar" o programa que irá cadastrá-la no servidor de GG. Após o cadastro, o *stakeholder* vai ao servidor e escolhe a *query* pelo nome. A *query* então é executada e os vários resultados mostrados em vários formatos diferentes. Os gráficos gerados são similares aos utilizados pelas soluções comerciais. Esse formato é preferível já que se mostrou extremamente popular.

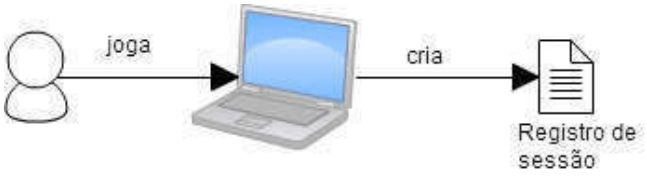
5.7 COMPONENTE *GAMEGUSTS: GG REPLAY*

A reprise de partidas é de grande ajuda para o desenvolvedor, tanto para entender melhor algumas anomalias nos dados, quanto para facilitar a busca por erros - reproduzindo passo a

passo os contextos. Dessa maneira, para aumentar ainda mais a interoperabilidade e o fator amigável aos usuários, *GG* sugere a adaptação do jogo para possibilitar reprises.

Em geral, um jogador humano é quem faz os comandos para a manipulação do jogo e, com a integração de *GG, GG* faz o registro da sessão. Como na Figura 23.

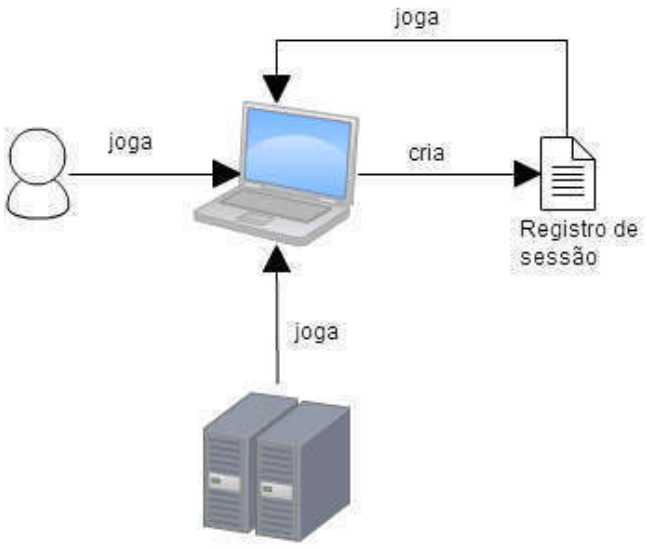
Figura 23 - Jogador joga o jogo e GG captura o registro de sessão



Fonte: o autor

No entanto, para possibilitar a reprise de partidas e testes automáticos, *GG* sugere o isolamento dos comandos, baseado no *Client SDK* do componente de captura do *GG Server*, para que o jogo possa ser jogado por um computador. Como na Figura 24.

Figura 24 - Com as alterações sugeridas por GG Replayer, o jogo pode ser jogado de outras maneiras.

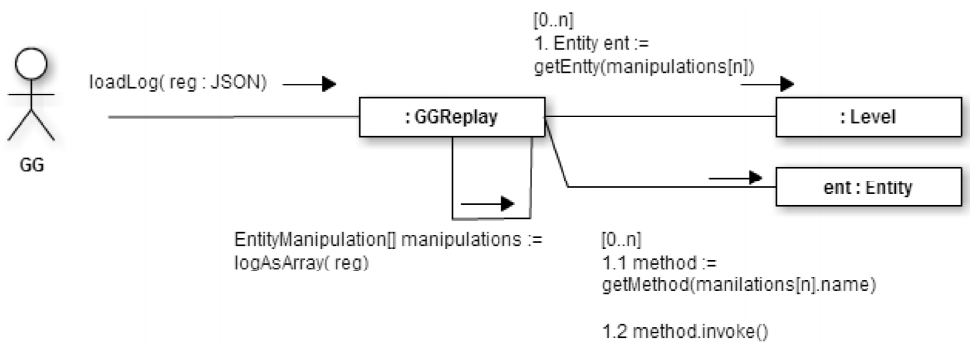


Fonte: o autor

Em versões do jogo que não estão abertas ao público, deve haver uma opção de jogo a partir de um algoritmo ou, pelo carregamento de uma sequência de manipulações em um arquivo de *DSL* do *GG Instance*. Essa funcionalidade pode utilizar a estrutura de classes criada para o *GGSDK*.

A cada linha do registro de sessão, o *GGReplay* deve chamar o método da manipulação de entidade nas entidades com o identificador (atributo do tipo *long* na classe *Entity*). Essa dinâmica é diagramada na Figura 25.

Figura 25 - Diagrama de colaboração do funcionamento do GGReplay.



Fonte: o autor

5.8 CONCLUSÃO

O Quadro 5 abaixo resume como as decisões de projeto feitas impactam nas propriedades para uma solução de registro de sessões com relação a *GG*.

Quadro 5 - Resumo com comentários das decisões de projeto de GG com relação aos atributos de um framework de dados de gameplay

Propriedade	Avaliação	Comentário
Conhecimento Prévio	+	Um processo para instanciação de GOP de acordo com o jogo a ser desenvolvido facilita a definição do que será capturado pelo desenvolvedor.
Expressividade	+	Possibilita a expressão de qualquer evento como nas soluções comerciais.
Separation of concerns	-	Apesar de reconhecer, não resolve o problema por completo.
Interoperabilidade	+	Os dados são apresentados em formatos voltados para o uso por computadores e fornece uma maneira simples, por meio de uma API, para recuperação.
Amigável ao usuário	+-	Utiliza uma abordagem visual para recuperação de dados por humanos. No entanto, requer um esforço a mais por parte do desenvolvedor.
Extensibilidade	+-	Adapta a abordagem de regras para facilitar a criação de novas análise para cada tipo de fato inserido na base.

Fonte: o autor

6. ESTUDOS DE CASO

Este capítulo tem como objetivo apresentar estudos de caso e experimentos para demonstrar a efetividade e potencial do uso de *GG* em jogos.

Para a validação completa de *GG*, o *framework* foi usado em diferentes situações na plataforma da Olimpíada de Jogos e Educação, OjE [87], que foi usada por mais de 100.000 jogadores, estudantes de escolas públicas nos estados de Pernambuco, Acre e Rio de Janeiro.

Readaptações de jogos clássicos para incluir conteúdo educacional, os jogos da OjE foram todos desenvolvidos usando a tecnologia *Adobe Flash* [88] na linguagem de programação ActionScript 3.0. O GG foi integrado nos jogos Imuno, Tony Jones, "Jorginaldo, Doutor Antena e a pipoca" e Chuteira Premiada. Para tanto, cada um dos jogos teve as entidades e manipulações de entidades mapeadas, as linguagens de representação e visual de *query* criadas e, finalmente, foram integrados ao servidor de GG. Mais de meio milhão de sessões de jogo, de todos os jogos, foram registradas.

Um ambiente de produção real levanta algumas questões para a adoção de uma nova tecnologia. Com GG não seria diferente e algumas perguntas surgiram para justificar o uso de uma nova abordagem:

- Qual o esforço necessário para integração de GG em um jogo?
- Como esse esforço se compara à integração com um *framework* mais tradicional como *Flurry*?
- Como esse esforço evolui no tempo? Com mais experiência com *GG* ele diminui?
- Depois da integração, existe algum ganho posterior nas análises?

Nesta seção apresentamos o detalhamento dos experimentos realizados em busca de respostas. Todo o trabalho desenvolvido foi feito por uma mesma equipe com experiência prévia em jogos, no uso do *Adobe Flash* e na interação de *Flurry*. A equipe padrão para desenvolvimento de jogos da OjE era composta por: um produtor e um *game designer* para três projetos, um programador, dois artistas e um engenheiro de qualidade (ou testes).

6.1 INTEGRAÇÃO DO GG

O objetivo desta seção é tentar responder as duas primeiras perguntas sobre o esforço envolvido na integração de *GG*. Para tanto, especificamos tarefas dentro de uma lógica de teste A/B (*Flurry x GG*) para se obter alguns dados experimentais.

As tarefas, detalhadas a seguir, foram executadas em quatro jogos diferentes. Em todo os casos, o jogo possuía um documento de game design confeccionado pelo *game designer* e o produtor do jogo. As tarefas foram executadas em jogos já em fase beta. Ou seja, todas as funcionalidades planejadas para o lançamento já estavam implementadas.

É importante salientar que a equipe de desenvolvimento já estava familiarizada com o uso do *Flurry* e não tinha nenhum conhecimento prático do GG. Claro, isto introduz um viés forte no teste A/B que deverá ser levado em conta na ponderação dos resultados. No entanto, entendemos que esta situação seria aceitável ou até mesmo preferível. Primeiramente, ela distorce o resultado contra o GG e não a favor dele, o que significa que se os resultados do GG forem próximos ao do *Flurry*, os primeiros serão possivelmente melhores com a ponderação. Segundo, a alternativa “mais rigorosa” experimentalmente seria fazer o teste A/B com equipes sem nenhum conhecimento ou prática nenhuma das ferramentas. Ora, essa escolha nos levaria a trabalhar com estudantes e iniciantes na área de jogos, já que os profissionais conhecem o *Flurry*. Além disso, dificilmente teríamos acesso a demandas de jogos (game designs) do mercado. Assim, achamos que seria melhor lidar com o viés pró-flurry, realizando os testes com uma equipe de profissionais experientes em cima de uma demanda real de mercado, do que eliminar o viés citado, para possivelmente cair em outros relacionados ao fato de utilizarmos equipe pouco experiente em demanda artificial.

O primeiro passo foi a definição de que dados capturar. Em geral, essa tarefa é feita pelo produtor do jogo em conjunto com o *game designer*. Assim, os dois definiram os dados *adhoc* para os quatro jogos (como era feito usualmente) e depois, fizeram o mesmo usando o *GG Instance*. O Apêndice A detalha todo o processo de integração do GG *Instance* no jogo Imuno.

A tabela 1 mostra o custo de integração em homens/hora dessa definição.

Tabela 1 - Esforço para definição dos dados usando um processo adhoc e usando o GG Instance

Jogo	Homens/hora Flurry	Homens/Hora GG	Diferença de Custo
Imuno	2	8	300%
Jorginaldo, Dr Antena e a Pipoca	2	6	200%
Tony Jones	1	3	200%
Chuteira Premiada	1	2	100%
Media	1.5	4.75	216%

Fonte: o autor

O processo do GG *Instance* tem um custo maior que o dobro do *Flurry,* até porque a equipe já conhecia e já vinha utilizando o *Flurry*. No entanto, é possível notar uma diminuição no custo relativo a GG na medida em que o processo é executado, ou seja, a equipe se torna mais produtiva ao conhecer melhor o processo do GG.

Em um segundo experimento, dois programadores diferentes de uma mesma equipe de projeto, ambos com experiência prévia em *Flurry*, receberam uma das seguintes tarefas: (1) integrar *Flurry* no jogo e (2) integrar o *GG Server* no jogo e criar o *GG Viz*. Os programadores deveriam capturar os dados definidos no experimento anterior com o *game designer* e o produtor. O *GG Replay* ficou fora dessa comparação pois não é um serviço oferecido por *Flurry* e, portanto, não seria uma comparação de escopos similares. Esse experimento foi feito separadamente como veremos nesta seção.

A tabela 2 mostra o custo de integração em homens/hora das duas tarefas em cada um dos jogos.

Tabela 2 - Esforço para integração do GG Server nos jogos comparado ao Flurry

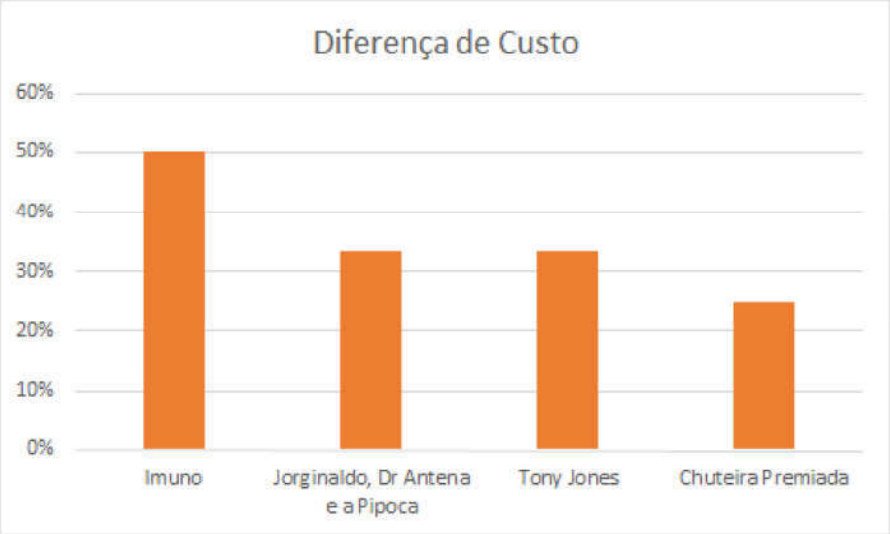
Jogo	Homens/hora Flurry	Homens/Hora GG	Diferença de Custo
Imuno	8	12	50%
Jorginaldo, Dr Antena e a Pipoca	6	8	33%
Tony Jones	6	8	33%
Chuteira Premiada	4	5	25%
Media	6	8.25	38%

Fonte: o autor

O custo em homens/hora para realizar a integração completa foi, em média, apenas 38% maior que o custo para integração do *framework Flurry,* como mostra a quarta coluna da tabela

2. Importante notar que o custo de integração do GG *Server* e do *GG Viz* nos jogos, em comparação ao Flurry, foi decrescente assim como no GG *Instance*. À medida que o time aprendeu a executar o processo, o custo diminuiu para os dois casos.

Figura 26 - Esforço relativo da integração do GG x integração do Flurry



Fonte: o autor

Por fim, a tarefa de criar a possibilidade de assistir as reprises dos dados capturados em uma partida foi dada aos mesmos programadores. Ou seja, adaptar o jogo para que o mesmo pudesse ser jogado a partir da leitura de um registro e não por um jogador humano. O esforço necessário para essas tarefas está detalhado na Tabela 3.

Tabela 3 - Esforço para possibilitar reprises de partias previamente capturadas

Jogo	Homens/hora Flurry	Homens/Hora GG	Diferença de Custo
Imuno	30	16	-47%
Jorginaldo, Dr Antena e a Pipoca	18	12	-33%
Tony Jones	16	12	-25%
Chuteira Premiada	12	6	-50%
Media	19	11.5	-39%

Fonte: o autor

A inclusão dessa funcionalidade mostra um ganho de economia no uso de GG para essa tarefa especificamente.

Ao calcular o esforço total das três etapas, mostrado na tabela 4, verificamos que o esforço despendido é similar entre as duas soluções, com uma leve economia de 6% para o uso de *GG*.

Tabela 4 - Comparação do esforço das três tarefas somadas até a criação da funcionalidade de reprises de Flurry x GG

Jogo	Homens/hora Flurry	Homens/Hora GG	Diferença de Custo
Imuno	38	36	-5%
Jorginaldo, Dr Antena e a Pipoca	26	26	0%
Tony Jones	23	23	0%
Chuteira Premiada	17	13	-24%
Media	26	24.5	-6%

Fonte: o autor

Consideramos este resultado muito positivo pois, como já explicamos, havia um claro viés pró-flurry, já que a equipe tinha conhecimento prévio e prática no seu uso, e nunca tinha usado o GG.

Vale salientar que, durante a implementação da reprise pelo programador que não estava usando GG, notamos que em várias ocasiões a reprise precisava de informações que não haviam sido capturadas. Por isso, os dados de *gameplay* a capturar precisavam ser revistos com o produtor e *game designer* e, em seguida, o código de captura precisava ser ajustado. Em outras palavras, a tabela de custos da reprise inclui custos das atividades anteriores no uso do *flurry*, o que indica que possivelmente o *flurry* não fornece apoio suficiente para se evitar retrabalho, por não antecipar certos problemas.

6.2 ANÁLISE DE DADOS PÓS-INTEGRAÇÃO

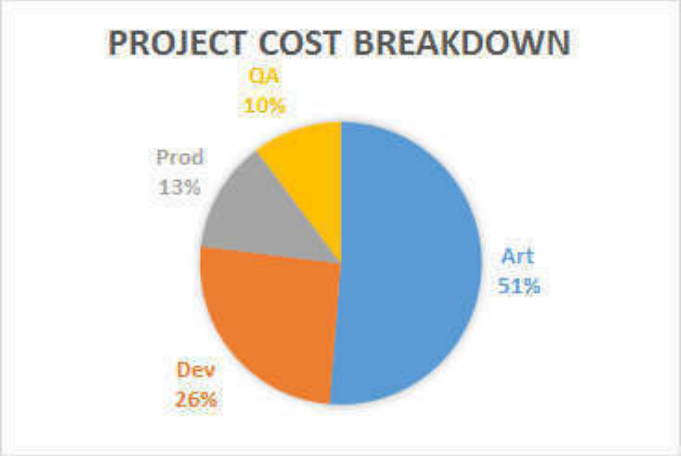
Além dos experimentos de integração de GG, dois problemas relacionados à análise dos dados de *gameplay* foram analisados. O primeiro experimento tinha o foco em qualidade de *software*, com o objetivo de reduzir as horas de testes manuais. O segundo tinha como objetivo dar credibilidade aos resultados - evitando “roubos” - a um custo razoável de operação.

6.2.1 Automação de Testes

O objetivo desse experimento era tentar novas soluções para reduzir o custo de testes manuais que consumiam até 64 homens/hora mensalmente, em torno de 10% do custo do projeto. A alocação do engenheiro de testes era dividida igualmente entre testes exploratórios e testes de regressão.

Novamente, dois programadores ficaram encarregados das tarefas. Desta vez, automatizar testes de regressão e testes exploratórios.

Figura 27 - Divisão do esforço empregado na produção do jogo Tony Jones por responsabilidade no projeto



Fonte: o autor

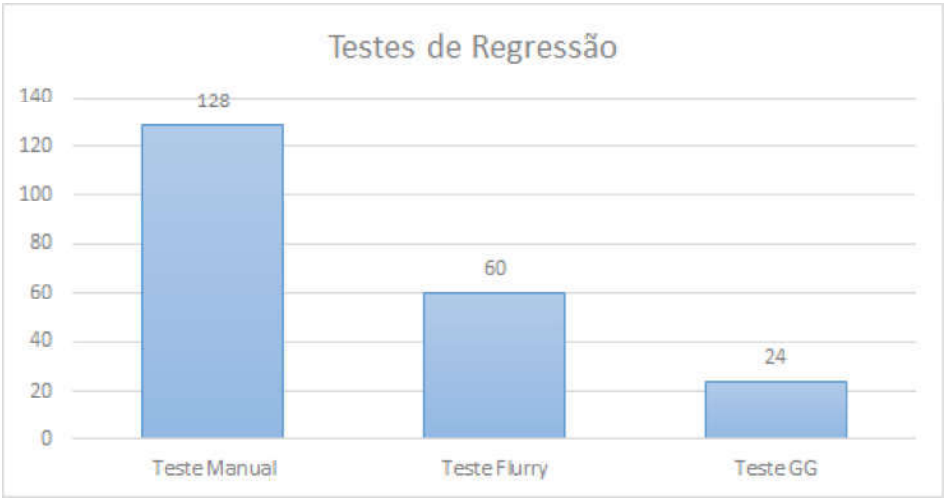
Os testes de regressão são feitos a cada nova versão do jogo. O processo consiste em repetir o contexto de todos os *bugs* já registrados, para garantir que novas funcionalidades não inseriram novamente o erro no jogo.

A automação desses testes foi feita com a catalogação dos registros de jogadas de todos os quatros jogos que continham *bugs*. Assim, bastava alimentar o jogo com o registro contendo o *bug* e ajustar o registro da jogada para indicar o teste. Por exemplo, se ocorreu um erro na contagem de pontos, indicar que era necessário checar ao final da jogada que a pontuação era a esperada.

Como nos outros casos, dois programadores diferentes foram encarregados de implementar essa funcionalidade. Um baseado no registro feito com o *Flurry* e o outro com o

registro feito por GG. O esforço necessário para essa automação no *GG* foi 40% do esforço usado para o registro feito em *Flurry* e está detalhado na figura 28.

Figura 28 - Comparação dos custos, em homens/hora, nos testes de regressão feitos manualmente, com Flurry com o GG Replay



Fonte: o autor

Flurry não tem, em sua API, as mesmas funcionalidades do *GG Server*. Por esse motivo, foi necessário um passo a mais para fazer o *download* dos registros do *Flurry* e em seguida separá-los em sessões individuais. Esse foi um dos motivos do tempo maior em *Flurry*. No entanto, apesar de demostrar uma maior preocupação de *GG* com a interoperabilidade, avaliamos que essa é uma falha individual de *Flurry* e não de todas as soluções comerciais.

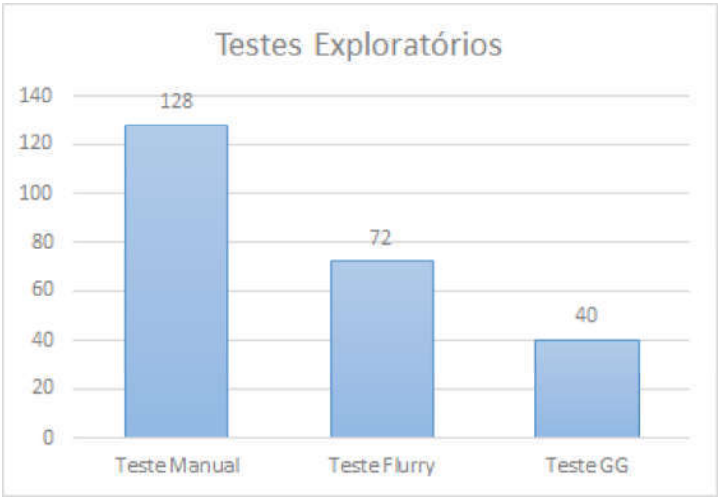
Outro problema, esse mais relevante, foi a falta de padronização entre os registros. Cada jogo possuía diferentes representações das manipulações de entidade, o que requereu diferentes tratamentos para indicar as falhas. Em *GG*, mesmo os jogos possuindo formalizações diferente do mesmo evento, essa padronização na representação e no código do GGSDK, proporcionou reuso de código, resultando nessa economia.

Para o segundo experimento, automação dos testes exploratórios, foi usado apenas o jogo *Tony Jones*. A primeira rodada de testes exploratórios foi realizada por engenheiros de teste e as jogadas foram devidamente capturadas. A partir de um conjunto de 20 jogadas, foi criado um método de exploração automática utilizando algoritmos genéticos. O *framework* de

algoritmo genético utilizado foi o mesmo. O algoritmo genético utilizado e mais detalhes sobre o uso com GG está detalhado no Apêndice C.

O experimento seguiu o mesmo protocolo de dois programadores, um usando os registros de GG e o outro usando os registros no *Flurry*. O esforço necessário para esse trabalho está detalhado na figura 29.

Figura 29 - Comparação dos custos, em homens/hora, nos testes exploratórios feitos manualmente, com Flurry com o GG Replay



Fonte: o autor

Novamente, o GG *Instance* foi primordial na economia de esforço. Vários dados de *gameplay*, principalmente das entidades não relacionadas ao jogador, como os inimigos, não estavam presentes na representação de *Flurry*. Isso dificultava a comparação entre diferentes indivíduos para a operação de *crossover*.

6.2.2 Análise de "roubos"

Para um novo experimento, os mesmos dois programadores foram designados para automatizar a auditoria feita nos registros de maneira a evitar confrontos com os jogadores. Os alunos eram entrevistados na escola e deveriam provar que eram capazes de atingir certas pontuações suspeitas.

Nesse caso, os programadores deveriam criar um leitor dos registros e incluir os eventos das partidas em uma base de regra *Drools*. A base *Drools* era efetivamente responsável pela análise. O processo de criação da base de regras para o uso com *GG* está detalhado no Apêndice B.

Tabela 5 - Comparação de custos para auditoria manual x auditoria automatizada

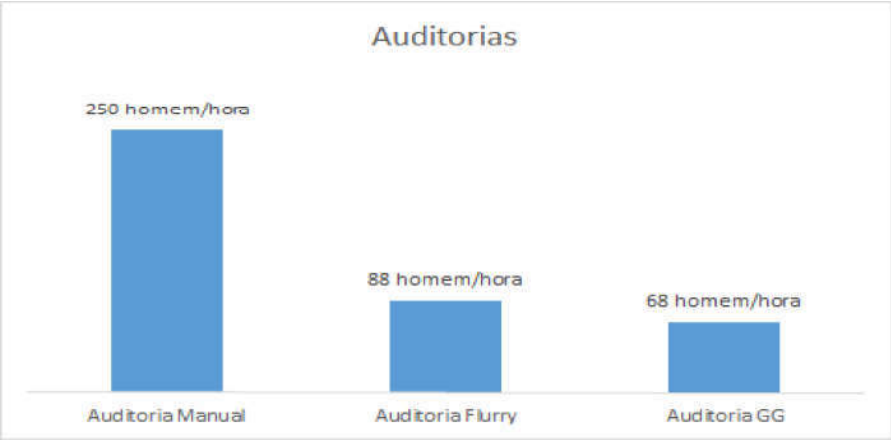
	Auditorias			Visitas			Total
	Sessões	Custo Médio	Custo Total	Qtd	Custo Visita	Custo Total	
Auditoria Manual	10	1 homem/hora	10 homem/hora	10	24 homem/hora	240 homem/hora	250 homem/hora
Auditoria Flurry	100000	0.00040 homem/hora	40 homem/hora	2	24 homem/hora	48 homem/hora	88 homem/hora
Auditoria GG	100000	0.00020 homem/hora	20 homem/hora	2	24 homem/hora	48 homem/hora	68 homem/hora

Fonte: o autor

Como descrito na tabela 5, o custo das visitas nas escolas para entrevistar os suspeitos era proibitivo para as dezenas de milhares de sessões de um jogo. No entanto, com a análise proporcionada pelos dados de *gameplay* capturados, esse custo foi diminuído drasticamente. A codificação das regras em *Drools* exigiu um esforço de 16 horas/homem que pode ser diluído entre as análises e incluído no cálculo dos dois casos do experimento (*Flurry* e GG).

Além disso, somente as pontuações indicadas como suspeitas pelo sistema requeriam uma visita (no caso, somente duas). Assim, como mostrado na figura 30, a automação com GG acarretou em uma diminuição de quase 75% nos custos de auditoria semestral da plataforma. Maior que a *Flurry* que foi em torno de 65%.

Figura 30 - Comparação dos custos, em homens/hora, das auditorias manuais, usando os dados de Flurry e usando os dados em GG



Fonte: o autor

Novamente, a padronização de *GG* foi essencial para diminuir os custos. Nesse caso, o registro feito para o *Flurry foi* suficiente. Ou seja, não foi necessário voltar ao jogo e incluir mais código de captura. No entanto, para cada jogo, um novo parser era necessário devido à grande diferença de representação de um jogo para outro. No caso de *GG* os *parsers* eram muito parecidos e o reuso de código foi importante.

6.3 OUTROS RESULTADOS NÃO QUANTITATIVOS

Alguns resultados obtidos não foram feitos com o rigor dos experimentos. No entanto, são casos que valem ser relatados para reforçar o potencial da captura organizada de dados de *gameplay*.

Um dos resultados interessantes foi atingido na busca por estratégias dominantes. Estratégias dominantes são aquelas que, garantidamente, geram o melhor resultado no jogo (por exemplo, um habilidoso jogador de jogo da velha nunca perde) e por isso não são desejáveis em nenhum jogo. No caso do *Tony Jones,* uma estratégia dominante foi encontrada utilizando a pontuação no *replay* como função de *fitness*, pois todos os indivíduos da população convergiam para uma mesma jogada que foi facilmente identificada assistindo ao *replay* e as regras do jogo modificadas para que mais de uma maneira de jogar pudesse gerar a pontuação máxima.

Como, inicialmente, a pontuação levava em conta o tempo para concluir o nível, a melhor estratégia era sempre acionar a garra toda vez que estivesse disponível (ou seja, acoplada ao trator do *Tony Jones*). Mudando a pontuação para ser mais baseada em precisão, quanto menos disparos da garra mais pontose a estratégia dominante foi eliminada.

A linguagem visual desenvolvida para o Imuno se provou de extrema utilidade para *game designers* que tinham dificuldade em expressar que tipos de informações gostariam de obter a partir das analíticas. Por exemplo, um erro de design foi encontrado no último inimigo (o chefão) do nível dois do jogo. Os jogadores encontravam um certo local na tela em que a nave do jogador não era atingida pelo inimigo e podia atirar indefinidamente sem atingir o inimigo também. No entanto, esse inimigo criava novos inimigos a cada intervalo de tempo e

o jogador podia estacionar na posição segura e ficar matando esses inimigos criados pelo chefão e ganhar uma quantidade infinita de pontos. Posteriormente, descobriu-se que vários jogadores encontraram tal erro e, após posicionar a nave na posição correta, colocavam algum peso na tecla de espaço (utilizada para atirar) e deixavam o jogo aberto por horas e gerando mais pontos. A mecânica desse chefão foi então repensada e refeita. Importante salientar que o jogo passou por dezenas de horas de testes com engenheiros de qualidade, grupos focados e designers de jogos e em nenhuma dessas sessões foi possível identificar esse erro. Só com a análise de milhares de sessões o problema foi encontrado e somente com a visualização das reprises pode ser corretamente diagnosticado.

Por fim, embora não fosse nossa preocupação principal, o *GG* conseguiu tratar mais de 500 mil jogadas de mais de 100 mil jogadores da OjE. O que mostra uma certa robustez e escalabilidade, apesar de ainda estar longe dos ambientes de jogos de sucesso que passam dos 100 milhões de downloads.

6.4 CONCLUSÕES DOS EXPERIMENTOS

Os experimentos indicam que a integração do GG no pior caso, com a equipe já habituada com o *Flurry*, tem custo equivalente àquela com *Flurry*. No entanto, os ganhos com o *GG* nas fases de análise, automação de testes e análise de “roubos”, são bastantes superiores à solução baseada em Flurry. Para a OjE especificamente, aumentou a economia na produção, tornando o produto mais barato e mais competitivo. Além disso, no caso da segurança, deu um nível de confiabilidade que foi essencial para diminuição das desistências e, principalmente, para a legitimidade dos vencedores que eram presentados sem suspeitas.

7. CONCLUSÕES E TRABALHOS FUTUROS

Este capítulo apresenta considerações finais sobre os principais tópicos abordados nesta tese, incluindo as contribuições alcançadas e indicações para trabalhos futuros.

O desenvolvimento de jogos como serviço traz um novo conjunto de preocupações. Tais preocupações são traduzidas em perguntas acerca do jogo. Tais perguntas são respondidas com o uso de diversos tipos de dados, que demandam um novo conjunto de ferramentas para auxiliar o desenvolvedor. Os *frameworks* disponíveis resolvem com sucesso uma parcela desses dados, principalmente dados de loja, dados de engajamento e dados demográficos.

Os dados de *gameplay*, por outro lado, não são tratados da mesma maneira. Apesar do suporte disponível, esse ainda é muito incipiente e precário. Nesse contexto, surge *GG. GG* tem como objetivo auxiliar os desenvolvedores no tratamento dos dados de *gameplay*.

A partir de todos os experimentos relatados no capítulo anterior, consideramos que o *framework* criado para captura e representação de conhecimento sobre partidas de jogos é um grande avanço com relação ao estado da arte da área. Não só criamos uma representação expressiva para representar tais dados, como um processo extensível para sua aplicação em uma gama de jogos. Os resultados são encorajadores: o GG se mostrou uma maior economia de esforço no longo prazo, apesar de apresentar uma curva de aprendizagem maior com relação aos outros *frameworks*.

Importante salientar que essa abordagem possui alguns pontos fracos claros. O primeiro deles é a dependência extrema a *GOP* limitando, dessa maneira, a sua expressividade e completude. Além disso, ao exigir uma instanciação para cada novo jogo, o reuso do conhecimento fica restrito a *GOP*. Cada desenvolvedor pode utilizar a experiência anterior em um novo projeto, mas isso não é tratado apropriadamente por *GG*.

Outro ponto importante a ser destacado é o foco de *GG* em jogos casuais. Os jogos de OjE em geral, apesar de variadas mecânicas, estão classificados como jogos casuais de pequeno para médio porte, com produção de equipes pequenas, de até oito pessoas, por um período de até seis meses. O GG não foi testado em jogos *hardcore,* ou até casuais, de produções de grande

porte onde as equipes podem facilmente chegar a centenas de pessoas e o tempo de produção é maior que dois anos. Esse contexto impõe outros desafios.

A principal contribuição é um primeiro passo na direção da organização da captura e na padronização da representação de dados de *gameplay*. Claro que a própria implementação do *GG* é uma contribuição, porém, entedemos que ambas são contribuições mais conceituais e que podem ser utilizadas por outros *frameworks, para* que possam ser criadas com o foco nos dados de *gameplay* que entedemos ser uma tendência futura.

7.1 TRABALHOS FUTUROS

Como dito anteriomente, uma das limitações desse trabalho é a depência do GOP. Faz-se necessário um maior número de estudos de caso e integração, em diferentes tipos de jogos, para avaliar a necessidade de revisão do próprio GOP. Além disso, a representação e ontologia criadas para um jogo não é reusada no GG *Instance*. Ou seja, o desenvolvedor sempre tem o GOP como ponto de partida mesmo quando ele cria vários jogos semelhantes (vários *shooters* como o Imuno, por exemplo).

GG auxilia o desenvolvedor na fase de produção do jogo. Um novo componente para trabalhar processos de acompanhamento dos dados e também de ajustes no *gameplay* de novas funcionalidades se faz necessário. Como, por exemplo, o GG pode ajudar na criação de novos níveis do Imuno?

Por fim, toda infra-estrutura disponível em ferramentas comerciais deve ser usada de maneira mais sistemática por GG. É possível utilizar a representação de *GG Instance* em um desses *frameworks*, no entanto, é necessário um trabalho extra não formalizado por GG nesse momento.

REFERÊNCIAS

1 GLASSER, A. **The Smurfs & Co Top This Week`s List of Fastest-Growing Games by DAU**. Disponível em: <http://bit.ly/1jbQWnA>. Acesso em: 12 maio. 2012.

2 VAN RIJMENAM, Mark. **Zynga Is A Big Data Company Masqueraded As A Gaming Company.** Disponível em: <https://datafloq.com/read/zynga-is-a-big-data-company-masqueraded-as-a-gamin/505> Acesso em: 21set. 2015.

3 PEDERSEN, C; TOGELIUS J.; YANNAKAKIS, G.N. **Modeling Player Experience in Super Mario Bros.** Proc. IEEE Symp. Computational Intelligence and Games. p. 132-139, set. 2009.

4 Marketing Tech. **App Annie report reveals freemium rules for app revenue**. Disponível em <http://www.marketingtechnews.net/news/2014/mar/28/app-annie-report-reveals-freemium-rules-app-revenue/> Acesso em 7 set. 2014.

5 SOTAMAA, O; KARPPI, T. **Games as Services Final Report**. TRIM Research Report 2010.

6 **Overview of Domain-Specific Language Tools**. Disponível em <http://msdn.microsoft.com/en-us/library/bb126327.aspx> Acesso em 23 jul. 2014

7 WILLIAMS, J. **Acquisition, Retention, Monetization: Understanding & Optimizing player behavior on any plataforma**. Disponível em <http://casualconnect.org/lectures/monetization/acquisition-retention-monetization-josh-williams>. Acesso em: 27 de outubro de 2014.

8 NELSON, M.J. **Game Metrics Without Players: Strategies for Understanding Game Artifacts**. AAAI Technical Report WS-11-19 2009

9 HARPSTEAD, Erik et al. **Using extracted features to inform alignment-driven design ideas in an educational game**. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 26 abr – 01 maio, 2014.

10 HARPSTEAD, E.; MYERS, B.A.; ALEVEN, V. **In Search of Learning: Facilitating Data Analysis in Educational Games**. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. p. 79-88 CHI 2013, ACM Press (2013), ISBN: 978-1-4503-1899-0

11 EL-NASR, S.; DRACHEN, A.; CANOSSA, A. (Eds.). "**Game Analytics**", 1st ed. New York, Sprint, 2013.

12 CHAN, J.T.C.; YUEN, W.Y.F. **Digital game ontology: Semantic web approach on enhancing game studies**. Proceedings of the 9th International Conference on Computer-Aided Industrial Design and Conceptual Design, 2008. p. 425-429 CAID/CD 2008.

13 HÜRSCH W. L.; LOPES C. V. **Separation of Concerns**. Northeastern University, Boston, USA. Technical report NU-CCS-95-03. 1995.

14 CANOSSA, A. et al. **Benefits of Game Analytics: Stakeholders, Contexts and Domains**. In: A. Canossa et al. Editors Game Analytics: Maximizing the Value of Player Data. London: Springer-Verlag, p. 41-52. 2013

15 **Metacritic**. Disponível em < http://www.metacritic.com/> Acesso em: 27 jan. 2016.

16 SPRONCK, P. et al. **Adaptive game al with dynamic scripting**. Machine Learning, 63 (3), p. 217–248. 2006

17 SPRONCK, P; SPRINKHUIZEN-KUYPER, I; POSTMA, E. **Online adaptation of computer game opponent AI**. In Proceedings of the 15th Belgium-Netherlands Conference on Artificial Intelligence, p. 291–298. 2003

18 Admob. Disponível em: <http://www.google.com/Admob> Acesso em: 27 jan. 2016.

19 Chartboost App Monetization Platform. Disponível em:< http://www.chartboost.com/> Acesso em: 27 jan. 2016.

20 App Annie – The App Analytics and App Data Industry Standard. Disponível em: <http://www.appannie.com/> Acesso em: 27 jan. 2016.

21 App Figures – App Store Analytics for iOS and Android Developers. Disponível em: <http://www.appfigures.com/> Acesso em: 27 jan. 2016.

22 MOREIRA, A.V.; VICENTE FILHO, V.; RAMALHO, G.L. **Understanding mobile game success: a study of features related to acquisition, retention and monetization**. SBC 5, p. 2–12 .2014.

23 CHIU, K.; CHAN, K. **Using data mining for dynamic level design in games**. Foundations of Intelligent Systems. ISMIS 2008. Lecture Notes in Computer Science, Vol 4994 Springer, Berlin, Heidelber 2008.

24 SORENSON, N.; PASQUIER, P. **The evolution of fun: Automatic level design through challenge modeling**. Proc. 1st Int. Conf. Comput. Creativity, p.258 -267. 2010.

25 **Pinup Heroines Game**. Disponível em: <https://www.playfg.com/pinup-heroines-game.html> Acesso em: 21 set. 2015.

26 JOHNSON, Eric. **In Free Mobile Games, a Tiny Minority of 'Whales' Still Does Most of the Buying**. Disponível em: <http://recode.net/2015/04/09/in-free-mobile-games-a-tiny-minority-of-whales-still-does-most-of-the-buying/> Acesso em: 21 set. 2009.

27 WAWRO, Alex. **Why freemium mobile developers can't succeed without whales**. Disponível em:

<http://www.gamasutra.com/view/news/211764/Why_freemium_mobile_developers_cant_succeed_without_whales.php> Acesso em: 21 set. 2015.

28 NASH, T. **Are Freemium Apps Killing Game Developers?** Disponível em: <http://www.developereconomics.com/freemium-apps-killing-game-developers/> Acesso em: 21 set. 2015.

29 ROTH-BERGHOFER, T.; ADRIAN, B. **From Provenance-awareness to Explanation-awareness**---When Linked Data Is Used for Case Acquisition from Texts in: Cindy Marling (ed.): ICCBR 2010 Workshop Proceedings, Alessandria, Italy, Dipartimento di Informatica Università del Piemonte Orientale "A. Avogadro", Alessandria, p.103-106. 2010.

30 BOSC, G. et al. **Strategic pattern discovery in rts-games for e-sport with sequential pattern mining.** 2013**.**

31 GONG Wei et al. **In-game action list segmentation and labeling in real-time strategy games**, 2012 IEEE Conference on Computional Intelligence and Games (CIG), pp.147-154, 11-14 Sept. 2012.
.
32 MEDLER, B. **Play With Data – An Exploration of Play Analytics and its Effects on Player Experiences**. Tese de Doutorado. 2012.

33 **World of Logs**. Disponível em: <http://www.worldoflogs.com/> Acesso em: 21 set. 2015.

34 FULTON, B.; MEDLOCK, M. **Beyond focus groups: Getting more useful feedback from consumers.** Proc. Game Developer's Conference. San Jose, CA, 2003. Association for Computer Machinery (ACM)

35 Shacknews.com. **EA continues to add servers to alleviate SimCity crush**. Disponível em <http://www.shacknews.com/article/78127/ea-continues-to-add-servers-to-alleviate-simcity-crush> Acesso em: 26 jul. 2013.

36 Arstechnica.com. **Blizzard fixes Diablo III gold duplication bug, but the damage may be done.** Disponível em: <http://arstechnica.com/gaming/2013/05/blizzard-fixes-diablo-iii-gold-duplication-bug-but-the-damage-may-be-done/>Acesso em 26 jul. 2013.

37 Crash Reporter. Disponível em: <http://en.wikipedia.org/wiki/Crash_reporter> Acesso em 26 jul. 2013.

38 WHITTAKER J. A. **What Is Software Testing? And Why Is It So Hard?** IEEE Software, v.17, n.1, p.70-79, jan /fev. 2005.

39 P, Wolfgang. **Design patterns for object-oriented software development**. Reading, Mass.: Addison-Wesley. 1994

40 YAN, J; RANDELL, B. **An Investigation of Cheating in Online Games**. IEEE Security & Privacy, vol. 7, no. 3, mai/jun. 2009.

41 STALLINGS, W. **Cryptography and Network Security: Principles and Practice**. 4. ed. New Jersey: Prentice Hall, 2005.

42 BELLMAN, R.E. **Dynamic programming**. Princeton University Press. 1957

43 ZAGAL, J. **Game Ontology Project**. Disponível em: <http://www.gameontology.com/index.php/Main_Page>. Acesso em: 29 de março de 2014

44 HAZEWINKEL, M. **Predicate calculus**. Encyclopedia of Mathematics, Springer, 2001

45 MANSOURI, D.; HAMDI-CHERIF, A. **Ontology-oriented case-based reasoning (CBR) approach for trainings adaptive delivery**. 15th WSEAS Int. Conf. on Computers (CSCC2011), p.328-333. 2011

46 MACHADO, A.; AMARAL, F.; CLUA, E. **A trivial study case of the application of ontologies in electronic games**. Proceedings of the VIII Simpósio Brasileiro de Jogos e Entretenimento Digital, Sociedade Brasileira da Computaçao. 2009.

47 TAIRAS, R.; MERNIK, M.; GRAY, J. **Using ontologies in the domain analysis of domain-specific languages**. Proceedings of the 1st International Workshop on Transforming and Weaving Ontologies in Model Driven Engineering. CEUR Workshop Proceedings., CEUR-WS.org, vol. 395. 2008.

48 DOBBE, J. A. **Domain-Specific Language for Computer Games**, MSc dissertation, Department of Software Technology, Delft University of Technology, 2007

49 **Unreal Engine**. Disponível em <http://www.unrealengine.com>. Acesso em 01 ago. 2015.

50 FUNK, M.; RAUTERBERG, M. **PULP scription: a DSL for mobile HTML5 game applications**. Proceedings of the 11th international conference on Entertainment Computing, p.26-29 set. 2012.

51 MERNIK, M.; HEERING, J.; SLOANE A. M. **When and how to develop domain-specific languages**. *ACM Computing Surveys*, 37(4): p. 316–344. 2005.

52 COOK, S. et al. **Domain-specific development with visual studio dsl tools**. Pearson Education. 2007.

53 **Unity Engine** Disponível em <http://www.unity3d.com>. Acesso em: 01 ago.2015.

54 KIM, M. et al. **An Ethnographic Study of Copy and Paste Programming Practices in OOPL**. Proceedings of 3rd International ACM-IEEE Symposium on Empirical Software Engineering (ISESE'04), p. 83-92. 2004.

55 MCGUINNESS D.L; VAN HARMELEN F. eds. **OWL Web Ontology Language Overview**. World Wide Web Consortium (W3C) recommendation. Fev. 2004. Disponível em < http://www.w3.org/TR/owl-features>. Acesso em: 27 de setembro de 2014

56 THOMPSON, C. **Halo 3: How Microsoft Labs invented a new science of play - Wired**. 8 ago. 2007. Disponível em <http://www.wired.com/gaming/virtualworlds/magazine/15-09/ff˙halo>. Acesso em: 13 de julho de 2015

57 CORNELIS, F. et al. **A taxonomy of execution replay systems**. Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet. 2003.

58 DIONNE, C.; FEELEY, M.; DESBIENS, J. **A Taxonomy of Distributed Debuggers Based on Execution Replay**. In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, p. 203–214. 1996.

59 DUNLAP, G. W. et al. **Revirt: Enabling intrusion analysis through virtualmachine logging and replay**. In Proceedings of OSDI, p. 211–224. 2002.

60 NARAYANASAMY, S.; POKAM, G.; CALDER, B. **Bugnet: Recording application-level execution for deterministic replay debugging**. IEEE Micro v. 26, p.100–109. 2006.

61 MICKENS, J; ELSON, J; HOWELL, J. **Mugshot: deterministic capture and replay for Javascript applications**. In Proceedings of the 7th USENIX conference on Networked systems design and implementation (NSDI'10). USENIX Association, Berkeley, CA, USA, 11-11. 2010.

62 CHOI, J.-D.; SRINIVASAN, H. **Deterministic replay of Java multithreaded applications**. In Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, p. 48–59. 1998

63 GEELS, D. et al. **Replay debugging for distributed applications**. Proceedings of USENIX Technical, p.289–300. 2006.

64 **Flurry**. Disponível em: <http://www.flurry.com/>. Acesso em 30 nov. 2013.

65 **EventMetrics**. Disponível em: <https://developer.yahoo.com/flurry/docs/api/code/eventmetrics/>. Acesso em: 22 set. 2015.

66 ALLEMANG, D.; HENDLER J. **Semantic Web for the Working Ontologist**. Morgan Kaufmann, 2008.

67 DAVOU, K. I.; IDRUS R. **Ontology-based generic template for retail analytics**, p.188-194. APA. 2013.

68 MEPHAM W. **Semantically enhanced games for the Web**. Proceedings of the Web-Sci'09. Society On-Line, Athens, Greece. 2009.

69 JESS. **The Rule Engine for the Java Platform**. Disponível em <http://herzberg.ca.sandia.gov/>. Acesso em 26 jul. 2013.

70 ZAGEL, J.P., et al. **Towards an ontological language for game analysis**. DiGRA 2005 Conference: Changing Views – Worlds in Play. 2005. Páginas?

71 **Protégé**. Disponível em <http://protege.stanford.edu>. Acesso 27 jan. 2016.

72 WALLNER, G. **Play-Graph: A methodology and visualization approach for the analysis of gameplay data**. FDG. 2013. APA

73 BOEHM, B. **A Spiral Model of Software Development and Enhancement**. IEEE Computer, p.61 -72. 1988.

74 RIES, E. **The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses**. Crown Publishing. 2011

75 FALBO, R.A.; GUIZZARDI, G.; DUARTE, K.C. **An ontological approach to domain engineering**. Proceedings of the 14th International Conference on Software engineering and knowledge engineering. ACM. p. 351-358 2002.

76 ZAGAL, J. **Game Ontology Project.** Disponível em <http://www.gameontology.com/index.php/To_Move>. Acesso em: 11 de janeiro de 2015.

77 ISO/IEC, EXTENDED BNF 1996. Disponível em: <http://www.dataip.co.uk/Reference/EBNF.php> Acesso em 23 jul. 2014

78 NURSEITOV, N. et al. **Comparison of JSON and XML Data Interchange Formats: A Case Study**. Caine, 9, p.157-162. 2009

79 CATTELL, R. **Scalable SQL and NoSQL data stores**, ACM SIGMOD Record, v. 39, n. 4, p.12–27. 2010.

80 BROWNSTON, L.; FARRELL R.; KANT, E. **Programming Expert Systems in OPS5 Reading**. Addison-Wesley. 1985

81 **Drools.** Disponível em: <http://www.jboss.org/drools/>. Acesso: em 26 jul. 2013.

82 US patent 6467086, KICZALES et al. **Aspect-oriented programming**. 15 out. 2002

83 ALEXANDER R. **The Real Costs of Aspect-Oriented Programming**. IEEE Software. P 92-93. 20 jun, 2003.

84 WEILKIENS, T. et al. **Cross-Cutting Concerns**. Model-Based System Architecture, p. 265-270. 2016

85 GAMMA, E. et al. **Behavioral Patterns “Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, p. 325. 1994.

86 FURTADO, A.W. B. **SharpLudus: Improving Game Development Experience through Software Factories and Domain-Specific Languages**. Dissertação de Mestrado. Universidade Federal de Pernambuco. 2006.

87 **Olimpiadas de Jogos e Educação** (OJE). Disponível em: <http://www7.educacao.pe.gov.br/oje>. Acesso: em 12 jul. 2013.

88 Flash Developer Center. Disponível em < http://www.adobe.com/devnet/flash.html>. Acesso em: 23 jul, 2014.

89 **Music Ontology Specification**. Disponível em: <http://musicontology.com/>. Acesso em: 26 jul. 2013.

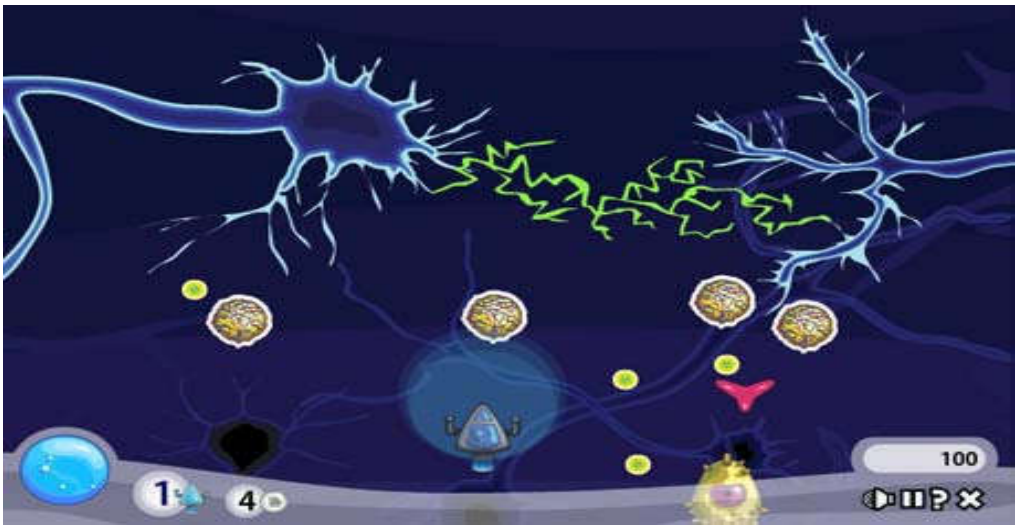
APÊNDICES

APÊNDICE A - INTEGRAÇÃO DO GG AO IMUNO

Este apêndice mostra os detalhes do processo de GG usado para o jogo Imuno na criação da representação com GG *Instance*, a integração com o *GG Server* e a criação da linguagem de *query* com o *GG Viz.*

Imuno é uma adaptação do jogo *shooter* de ação clássico do *Atari*, o *River Raid*, para o estudo de biologia. Nesse jogo, o jogador é uma nave com três vidas que pode, dentro do "sistema nervoso humano", mover-se nas quatro direções, atirar projéteis simples (um único), duplos ou triplos e também usar bombas para destruir os inimigos. No Imuno, os inimigos são as diferentes infecções do corpo humano e a nave é, na verdade, uma nano-nave que precisa eliminar as infecções. A Figura 31 mostra uma tela do Imuno, no estágio dentro do sistema nervoso.

Figura 31 - Tela do jogo Imuno em um estágio dentro do sistema nervoso.



Fonte: Joy Street S/A

Aqui relatamos o esforço para a integração completa com GG. O primeiro passo, de acordo com o processo definido no *GG Instance,* foi a listagem das entidades existentes no jogo. A lista de entidades do Imuno foi feita em inglês por questões de padronização de código.

- Player Ship - a nano nave que o jogador controla dentro de um corpo humano.

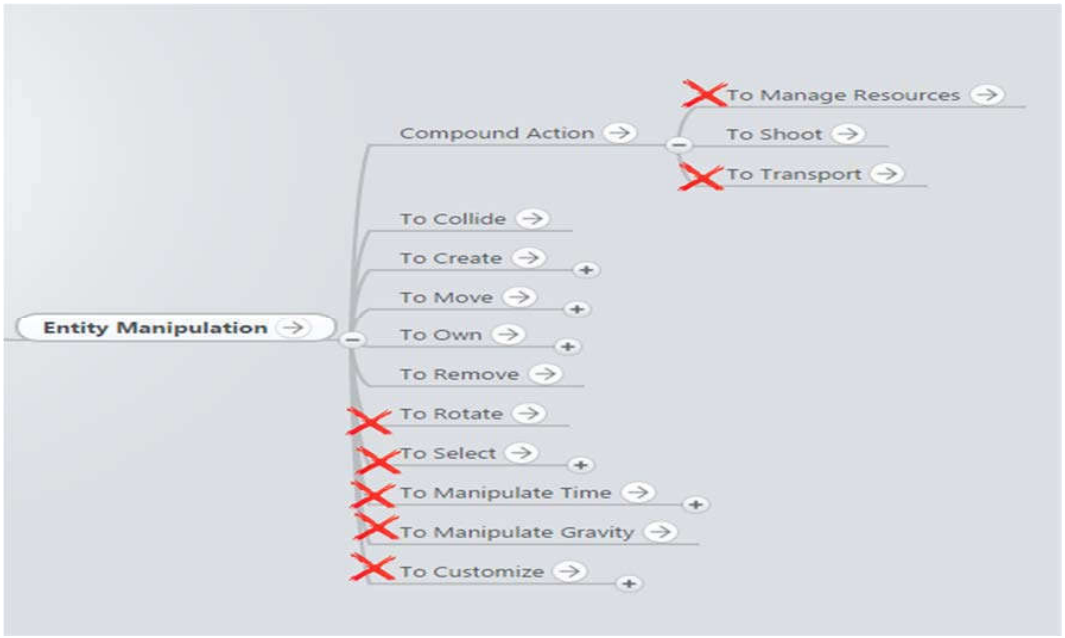
- Player Bullet - a bala simples que a nave do jogador dispara
- Tripple Bullet Powerup
- Single bullet Powerup
- Bombs
- Antibody
- Medicine Capsule
- Virus
- Bacterium
- Inflamed Epiglote
- Pneumococci
- Kosh Bacillus
- Mucus
- Infeceted Alveoli
- Polution Particles
- Infected Bronchia
- Respiratory System
- Brain

O segundo passo foi definir o subconjunto das manipulações de entidades do *GOP* que faziam parte do jogo a partir da eliminação dos conceitos não relacionados ao Imuno, como na Figura 32. Também foram utilizados nomes em inglês por padronização de código:

- ToCreate: criação de uma entidade dentro do ambiente (estágio) de jogo que o Player Ship se encontra;

- ToMove: Movimentação de várias das entidades no jogo;
- ToOwn: O jogador pode possuir um powerup que modifica seu tiro ou possuir bombas;
- ToShoot: A nave do jogador e vários dos inimigos podem atirar;
- ToCollide: Duas entidades podem colidir desde que uma delas seja a nave do jogador ou um tiro;
- ToRemove: Quando a nave do jogador ou um dos inimigos colidem entre si ou com um tiro.

Figura 32 - Conceitos da Entity Manipulation não relacionados ao Imuno removidos



Fonte: o autor

Continuando com os passos do *GG Instance*, cada uma das manipulações de entidades precisou ser definida em mais detalhes, em termos de atributos relevantes para análise. *ToCreate*, por exemplo, tem dois atributos: a entidade criadora e a entidade criada. O sistema respiratório (Respiratory System) é um dos ambientes em que o jogador pode estarinserido. Dessa maneira, é comum existirem entradas no registro como *ToCreate*(*KochBacillus, Respiratory System*) - quando o sistema respiratório cria um bacilo de Koch- ou

ToCreate(Player Ship, Respiratory System) - quando no início do jogo a nave do jogador é criada. O Quadro 6 mostra em mais detalhes as definições feitas para o *ToCollide*. O mesmo processo foi feito para todas as outras manipulações.

Quadro 6 - ToCollide para o Imuno. Posteriormente transformada em um diagrama de classe UML.

Entidade A	**EntidadeB**
Player Ship	*Enemy*
Player Ship	*EnemyBullet*
Player Ship	*Obstacle*
Player Ship	*Power up*
Player	*Life up*
Enemy	*Player Bullet*
Enemy	*Player Bomb*

Fonte: o autor

O passo seguinte do processo foi a criação de um diagrama de classes UML. A Figura 33 mostra o diagrama de classes final para as manipulações de entidades do Imuno.

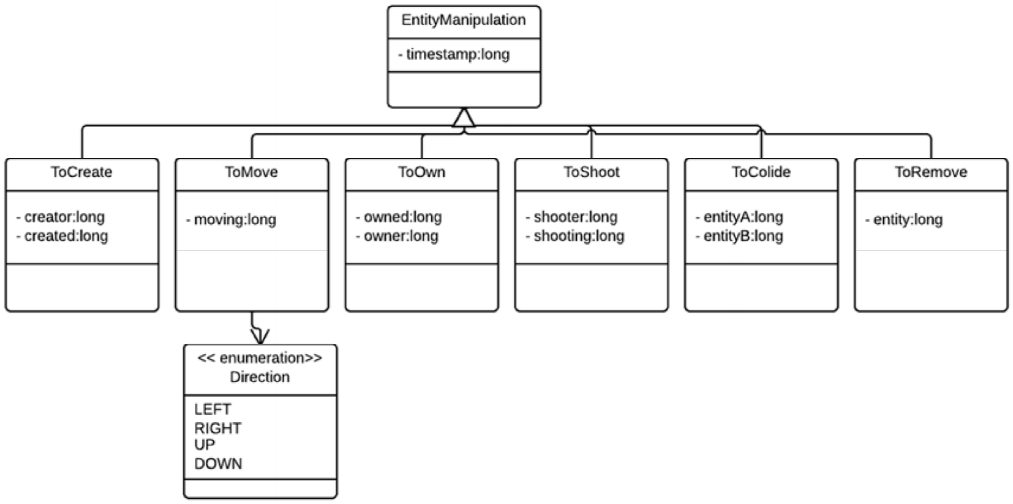
A partir do diagrama de classes, foram aplicadas transformações [17] até chegarmos a uma gramática livre de contexto na forma Backus-Nauer. No caso do Imuno terminamos com regras como as abaixo:

Pseudo-código 13 - Parte da BNF para representação de jogadas no Imuno

ToMove ::= moving DIRECTION
DIRECTION ::= “left” | “right” | “up” | “down”
EntityManipulation ::= entity ToCreate | ToMove | ToOwn | ToShoot | ToCollide | ToRemove timestamp
moving:: = number

Fonte: o autor

Figura 33 - Diagrama de classes da manipulação de entidade do Imuno



Fonte: o autor

Para facilitar o uso da linguagem na prática, GG recomenda a adaptação para um formato baseado em *JSON*. Dessa maneira, registros de sessão possuem um formato como:

Pseudo-código 14 - Trechos de um registro de jogada do Imuno feita por GG

```
{
  "entities":[
    {“id”: 0, “type”: "minigame.game.imuno.enemies.MucoPequeno"},
    {“id”: 1, “type”:"minigame.game.imuno.ship.ImunoShip"},
    {“id”: 2, “type”:"minigame.game.imuno.itens.AnticorpoVerde"},
    {“id”: 3, “type”: "minigame.game.imuno.enemies.Virus"},
    {“id”: 4, “type”: "minigame.game.imuno.levels.SistemaRespiratorio"},
    {“id”: 5, “type”: "minigame.game.imuno.PlayerBullet"},
    {“id”: 6, “type”: "minigame.game.imuno.PlayerBullet"},
```

```
    {“id”: 7, “type”: "minigame.game.imuno.PlayerBullet"},
    {“id”: 8, “type”: "minigame.game.imuno.PlayerBullet"}
  ],
  "EntityManipulations":{
    "ToCreate":{ "creator":4, "created":1, "timestamp":403},
    "ToShoot":{"shooter":1, "shooting":5, "timestamp":686},
    "ToMove":{ "moving":1, "direction":"left", "timestamp":1055},
    "ToMove":{ " moving ":1, "direction":"left", "timestamp":1146},
    "ToCreate":{"creator":4, "created":3, "timestamp":1507 },
...
    "ToShoot":{ "shooter":1, "shooting":7, "timestamp":5971},
    "ToShoot":{"shooter":1, "shooting":6, "timestamp":6134},
...
    "ToCollide":{"entity":5, "entity":3, "timestamp":7580},
    "ToRemove":{"entity":3, "timestamp":7803}
  }
}
```

Fonte: o autor

Com o *GG Viz*, cada um dos elementos é representado por seu *sprite* no jogo e cada uma das manipulações de entidades é uma ligação entre os mesmos. Para analisar a facilidade de matar um vírus, o *game designer* precisa saber, traduzindo para o registro, quantas vezes o padrão abaixo aparece:

- *ToShoot(PlayerShip, Player Bullet)*
- *ToCollide(Player Bullet, Virus)*

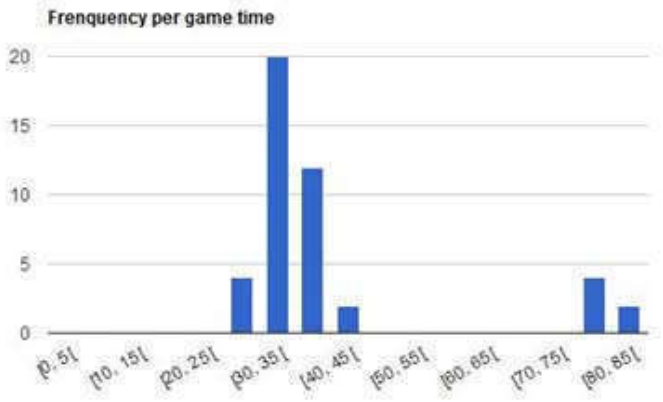
A figura 34 mostra como seria a representação do padrão acima numa linguagem criada para o Imuno.

Figura 34 - Representação da busca pelo padrão de tiros certeiros em Vírus

Fonte: o autor

Um dos resultados finais da busca é mostrado na Figura 35. A cada intervalo de tempo medido a partir do início da sessão, definido pela busca, o gráfico mostra quantas vezes o padrão acontece.

Figura 35 - Frequência que um vírus é atingido por sessão. Um vírus é atingido em média 20 vezes entre 30 e 35 segundos de sessão, por exemplo.



Fonte: o autor

É possível inferir, por exemplo, que o jogador passa muito tempo sem atirar nesse contexto, com um pico de tiros no meio do nível e alguns poucos no final. Esse comportamento pode não ser o esperado pelo *designer* que precisa averiguar o que está acontecendo.

APÊNDICE B - ANÁLISE DE ROUBOS NO IMUNO

Este apêndice explica a integração do *GG* com um conjunto de regras em *Drools* para a análise de roubos nas partidas do Imuno.Durante a operação do jogo Imuno, após o lançamento, a equipe do projeto notou que alguns jogadores utilizavam artifícios para conseguir mais pontos nas jogadas. Os dois principais eram:

1. Congelar a quantidade de vidas na memória: utilizando programas que vasculham a memória do computador, jogadores podem encontrar a posição de memória na qual a quantidade de vidas está guardada e colocar o valor que desejarem. Isto é impossível prevenir e difícil de identificar com a programação clássica dos jogos.

2. Interceptar a mensagem enviada do jogo para o servidor com a pontuação final e modificá-la. Fácil de fazer, mas muito difícil de prevenir. Mesmo com a utilização de criptografia das mensagens.

Como o jogo possui um ranking *online*, é possível identificar jogadores com pontuação muito alta e auditar a jogada de alguma maneira. Esse processo era, em geral, feito com visitas às escolas e o confronto direto com os jogadores. Abordagem que, além de só ser possível para um jogo de alcance local, gerava atrito entre os desenvolvedores e a comunidade de jogadores.

Utilizando uma base de regras no *framework* Drools [27] para avaliar registros de sessão foi possível, pela análise dos dados de *gameplay*, indicar se um dos dois ataques descritos acima foi executado.

Primeiro, o estado inicial do jogo - no caso do Imuno definido por um arquivo no formato *eXtensible Markup Language* (XML) [89] - precisou ser inserido na base de regras como fato. Depois, precisamos de um componente periférico que pudesse recuperar a sessão via a *API* do *GG* e traduzir as entradas como fatos para o sistema de regras.

Em seguida, o sistema de regras foi executado e caso existisse um roubo, alguma das regras seria disparada.

As regras são na verdade uma simulação simplificada do jogo focada na busca de roubos. Para um roubo do primeiro tipo, congelando a quantidade de vidas, um conjunto de

regras simples identifica se mesmo após o jogador ter perdido todas as vidas a nave, ele continua se movendo, atirando, etc. O código Drools para fazer isso seria como o abaixo:

Pseudo-código 15 - Código Drools para análise de roubo no número de vidas no Imuno utilizado no GG.

```
rule "Check player's health and update lives accordingly"
        when
          $p : Player(health <= 0);
          $l : Lives(amount > 0);
        then
          $l.setAmount($l.getAmount() - 1);
          $p.setHealth(3); # player always has 100 health, 3 for testing
          update($p);
          update($l);
          System.out.println( "Player died" );
end

rule "Game Over"
  when
          $l : Lives(amount == 0);
          $p : Player();
  then
      retract($l);
      retract($p);
      System.out.println("Game Over");
      # insert a fact stating that the game is over at this time
end

rule "Life cheated"
  when
      $p : Player(health >= 0);
      $c : ToCollide (entityA == $p.ID || entityB == $p.ID);
      not Lives();
  then
      System.out.println(“There are player collisions after his death" );
End
```

Fonte: o autor

Infelizmente, baseado em um acordo de confidencialidade, não podemos divulgar o formato do arquivo de definição do estágio em XML nem o código Drools que o insere na base.

Para o segundo tipo de ataque, de mudança de pontuação, uma análise mais extensiva do código foi necessária. Primeiro, identificamos pelo documento de regras do jogo que somente quando existe uma colisão entre a bomba ou bala do jogador, que resulta na remoção

de uma entidade inimiga, o jogador deve receber mais pontos. Esse aspecto foi mapeado no quadro 7. A regra deve também garantir que o inimigo foi de fato criado, que só foi destruído uma vez e que existiram colisões suficientes entre ele e uma bomba ou bala para destruí-lo. O código mostra como essa regra foi criada para atualizar a pontuação, baseado em inimigos que foram destruídos com uma bala do jogador.

Quadro 7 - ToCollide para o Imuno adicionada do resultado esperado para cada colisão.

Entidade A	**EntidadeB**	**Resultado esperado**
Player Ship	*Enemy*	Diminuir a vida do jogador e do inimigo
Player Ship	*Bullet*	Diminuir a vida do jogador, bala some
Player Ship	*Obstacle*	Diminuir a vida do jogador
Player Ship	*Power up*	A arma do jogador é modificado ou melhorada
Player	*Life up*	Aumenta a vida do jogador
Enemy	*Player Bullet*	Diminuir a vida do inimigo, bala some
Enemy	*Player Bomb*	Diminuir a vida do inimigo, jogador tem uma bomba a menos

Fonte: o autor

É importante observar que o jogador pode modificar o arquivo de definição de estágio de entrada. Assim, o estágio a ser jogado pode ter mais inimigos que o original, possibilitando um aumento da sua pontuação máxima. Por isso, precisamos garantir que o inimigo destruído foi efetivamente criado a partir do documento seguro sobre o estágio guardado no servidor. Além disso, o jogador pode alterar o próprio registro da sessão de dados e também precisamos

garantir que nenhum inimigo foi criado duas vezes ou que um inimigo não existe no documento original de descrição do estágio.

Pseudo-código 16 - Código Drools para análise de roubo no número de vidas no Imuno utilizado no GG.

```
rule "Bullet-Enemy Collision"
  when
     $e : Enemy( isDead == false );
     $p : Player();
     $b : Bullet(createdBy == $p.ID);
     $c : ToCollide (
        (entityA == $e.ID || entityB == $e.ID)
        && (entityA == $b.ID || entityB == $b.ID) )
  then
     $e.setHealth($e.getHealth() - $p.getDamage());
     retract($b);
     update($e);
     System.out.println( "Enemy lost some health: " + $e.getID());
end

rule "Cheated Bullet-Enemy Collision (Enemy alredy dead)"
  when
     $e : Enemy( isDead == true );
     $p : Player();
     $b : Bullet(createdBy == $p.ID);
     $c : ToCollide (
        (entityA == $e.ID || entityB == $e.ID)
        && (entityA == $b.ID || entityB == $b.ID) )
  then
     System.out.println( "Cheated collision involving enemy: " + $e.getID());
end

rule "An enemy has died, update player's score"
  when
     $e : Enemy (health <= 0, isDead == false);
     $p : Player();
  then
     $p.setScore($p.getScore() + 200);
     $e.setIsDead(true);
     update($e);
     System.out.println( "Player got some score from: " + $e.getID() );
End
```

Fonte: o autor

Após a base de regra ser executada, a variável da pontuação do jogador ($p.score) precisa ser comparada pelo servidor com a pontuação enviada na mensagem que chegou do cliente.

APÊNDICE C - AUTOMAÇÃO DE TESTES DO TONY JONES

Esse apêndice explica a automação dos testes de regressão e exploratórios a partir do registro de partidas no jogo Tony Jones.

No Tony Jones, figura 36, o jogador é um historiador que aciona uma garra com o objetivo de recuperar objetos em uma escavação. Alguns objetos são mais valiosos que outros. Existe um limite de tempo representado como “combustível”. No final do combustível, o jogo acaba.

Figura 36 - Estágio do Tony Jones com pedras, fósseis de dinossauros e pinturas rupestres.

Fonte: Joy Street S/A

A primeira etapa do processo é sempre a criação da linguagem de representação. Seguindo os mesmos passos do jogo Imuno, a linguagem de representação gerada para o Tony Jones produz registros como abaixo.

Pseudo-código 17 - Trecho de sessão de jogo gerada pelo GG em uma sessão do Tony Jones.

```
{
  "entities":[
    {“id”: 0, “type”: "minigame.game.tonyjones.TonyJones"},
    {“id”: 1, “type”:"minigame.game.tonyjones.item.Rock"},
    {“id”: 2, “type”:"minigame.game.tonyjones.item.DinossaurPart"},
    {“id”: 3, “type”: "minigame.game.tonyjones.item.GreenCollectable"},
    {“id”: 4, “type”: "minigame.game.tonyjones.item.RedCollectable"},
    {“id”: 5, “type”: "minigame.game.tonyjones.item.BlueCollectable"},
   {“id”: 6, “type”: "minigame.game.tonyjones.ExcavationSite"},
    {“id”: 7, “type”:"minigame.game.tonyjones.item.DinossaurPart"},
    {“id”: 8, “type”:"minigame.game.tonyjones.item.DinossaurPart"},
   {“id”: 9, “type”: "minigame.game.tonyjones.item.BlueCollectable"},
    {“id”: 10, “type”: "minigame.game.tonyjones.item.GreenCollectable"},
    {“id”: 11, “type”: "minigame.game.tonyjones.Collector"},
  ],
  "EntityManipulations":{
    "ToCreate":{ "creator":6, "created":0, "timestamp": 210},
    "ToCreate":{ "creator":6, "created":1, "timestamp": 211},
    "ToCreate":{ "creator":6, "created":2, "timestamp": 211},
    "ToCreate":{ "creator":0, "created":11, "timestamp": 211},
...
    "ToShoot":{"shooter":0, "shooting":11, "direction":"30", "timestamp": 400},
    "ToMove":{ "moving":11, "direction":"right", "timestamp": 487},
    "ToMove":{ " moving ":11, "direction":"left", "timestamp": 556},
...
    "ToShoot":{ "shooter":0, "shooting":11,, "direction":"44", "timestamp":  1032},
...
    "ToCollide":{"entity":11, "entity":9, "timestamp": 1080},
    "ToRemove":{"entity":9, "timestamp": 1081}
  }
```

Fonte: o autor

No Imuno, o move do *Playership* só era possível em uma de quatro direções (*left*, *right*, *up*, *down*). No Tony Jones, a garra, representada pela entidade *Collector*, pode se locomover da direita para a esquerda e da esquerda para a direita. Essa garra pode ser acionada em qualquer direção e é representada por um ângulo em graus a partir da linha horizontal do solo. O primeiro *ToShoot* no registro de exemplo mostra que houve um acionamento da garra, em 30 graus, que provavelmente não acertou nada e outro em seguida, em 44 graus, que recuperou a entidade com identificador 9, do tipo *BlueCollectable*.

Utilizando os operadores de mutação e cruzamento, novas jogadas eram geradas e testadas também com o uso do GG *Replay*. A função de *fitness* variava de acordo com o objetivo do teste.

O operador de mutação escolhe uma jogada e muda o tempo de disparo da garra e, consequentemente, o angulo no qual havia saído. Dessa maneira, disparos diferentes poderiam ser feitos modificando completamente a jogada.

O operador de cruzamento, por sua vez, busca pais compatíveis em um tempo T (que possuem a mesma configuração do nível em um certo *timestamp*) e a partir de uma mudança de direção da garra, cria dois filhos desses pais, um com o registro do primeiro pai até o tempo T e completa com o registro do segundo pai e vice-versa.