Universidade Federal do Pampa
Wolmir Luiz Frizzo Nemitz Neto

Cristina: Uma ferramenta para transformar protótipos de jogos em código reutilizável

Alegrete
2015

Wolmir Luiz Frizzo Nemitz Neto

Cristina: Uma ferramenta para transformar protótipos de jogos em código reutilizável

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Engenharia de Software.

Orientador: Prof. Dr. Fabio N. Kepler

Alegrete

2015

Wolmir Luiz Frizzo Nemitz Neto

Cristina: Uma ferramenta para transformar protótipos de jogos em código reutilizável

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Engenharia de Software.

Trabalho de Conclusão de Curso defendido e aprovado em 04 de dezembro de 2015.

Banca examinadora:

Prof. Dr. Fabio N. Kepler
Orientador

Professor Me. João Pablo Silva da Silva
Unipampa

Professor Me. Jean Felipe P. Cheiran
Unipampa

Este trabalho é dedicado à minha Bruxinha, que generosamente cedeu seu nome a este trabalho.

Agradecimentos

Agradeço à minha família e ao meu orientador, Dr. Fábio Kepler. Mas, principalmente, agradeço a Deus e a minha namorada por me darem a força e o motivo para terminar esta monografia.

"To invent, you need a good imagination
and a pile of junk."
(Thomas Edison)

Resumo

A aquisição de *game engines* é cara. Além disso, adaptá-los a projetos específicos constitui grande parte dos custos no desenvolvimento de jogos. Por isso, estúdios selecionam os *middlewares* mais fáceis de modificar ou desenvolvem a própria tecnologia. Em ambos os casos, reusabilidade de código é importante. A indústria de desenvolvimento de jogos digitais utiliza muito a prototipação como um modelo de processo de *software*. *Game jams* são eventos de desenvolvimento rápido de jogos. Eles são equivalentes ao processo de prototipação e geralmente têm um arquivo de submissões de códigos-fonte de jogos disponíveis na *Web*. Isso significa uma grande oferta de código pouco reutilizável. Conforme o objetivo desse trabalho, os autores desenvolveram uma aplicação que produz bases de códigos reutilizáveis a partir de um conjunto de protótipos de jogos. Para isso, eles pesquisaram técnicas de refatoração automática de código que a aplicação utiliza. Também pesquisaram métricas de reusabilidade para validar a aplicação através de experimentos que comprovam sua eficácia em gerar classes mais reutilizáveis que o código original. A aplicação obteve sucesso em transformar os códigos fonte de protótipos submetidos ao *game jam PyWeek*. O experimento confirmou que a base extraída é mais reutilizável que o código original. Ele também validou o método de refatoração proposto. Esse trabalho é importante porque demonstra o reaproveitamento de subprodutos de desenvolvimento para reduzir custos.

Palavras-chave: *Game Engine. Game Jam.* Processo Iterativo. Prototipação de jogos. Desenvolvimento de Jogos.

Abstract

Game engines are expensive. One of its biggest hidden costs is in adapting its features to suit a particular game. Therefore, ease of modification is a key requirement in game engine selection. Even so, traditional game development studios prefer to develop most of their technology. In both cases, code reuse is an important factor. Prototyping, as a software process model, is used extensively in the game development industry. Game jams are rapid game development events. They are equivalent to the game prototyping process and, generally, have a source code submission archive on the Web. This means a good supply of non-reusable code. As per the objective of this work, an application that transforms the source codes of a set of game prototypes into a reusable codebase was built by the authors. They researched automated refactoring techniques to use in the application. They also looked for reusability metrics to validate the tool's effectiveness in producing reusable classes from the original code. The application was successful in transforming the source codes of a set of prototypes submitted to the game jam PyWeek. An experiment confirmed that the extracted codebase was more reusable than the original code. It also validated the proposed refactoring method. This work is important because it shows the recycling of development by-products to reduce costs.

Key-words: Game Engine. Game Jam. Iterative Process. Game Prototyping. Game Development.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas

CK Chidamber e Kemerer

Lista de siglas

API Application Programming Interface

CBO Coupling Between Objects

CDM Call Based Dependence between Methods

DIT Depth of Inheritance Tree

IA Inteligência Artificial

LCOM Lack of Cohesion of Methods

MMG Massive Multiplayer Game

NIST National Institute of Standars and Technology

NOC Number of Children

RFC Response for Class

SSM Structural Similarity between Methods

UML Unified Modelling Language

WMC Weighted Methods per Class

Sumário

1 Introdução

Segundo Anderson et al. (2013), um *game engine* é um sistema de software aberto e extensível no qual um jogo de computador pode ser construído. Segundo a pesquisa de DeLoura (2011), a aquisição de *game engines* constitui um grande custo para o desenvolvimento de jogos. Ele relata que *engines* de alto nível tipicamente custam acima de um milhão de dólares. Além disso, 86% dos entrevistados responderam que o maior custo implícito está na adaptação das funcionalidades para jogos específicos. Por isso, os desenvolvedores consideram facilidade de modificação e a disponibilidade de código fonte os principais critérios de seleção de *game engines*. Ainda assim, a pesquisa reporta que a maioria dos desenvolvedores de jogos tradicionais desenvolvem sua própria tecnologia, por causa do alto custo dos *game engines*.

Em ambos os casos, reusabilidade de código é um fator importante. DeLoura (2011) cita que desenvolvedores preferem adaptar *game engines* às suas próprias ferramentas. Para evitar custos, eles precisam modificar o mínimo de código possível. No caso dos que preferem construir suas ferramentas, reusar projetos passados é fundamental para reduzir os custos envolvidos com a redundância de esforços.

Segundo Pressman (2014), prototipação é um modelo de processo de software. O autor afirma que desenvolvedores o utilizam quando há incerteza em relação aos objetivos e requisitos de um sistema. O processo é iterativo e consiste em definir os objetivos gerais do cliente; construir rapidamente um protótipo; e testar o protótipo com o cliente a fim de refinar os objetivos para a próxima iteração. No entanto, Pressman (2014) cita dois problemas desse modelo de processo. O primeiro é a possibilidade do cliente ignorar a qualidade ruim do protótipo em prol da urgência do desenvolvimento. O segundo problema ocorre quando os desenvolvedores decidem manter o código legado dos protótipos, e assim comprometem a qualidade do produto final. O autor sugere a utilização desse processo apenas quando há harmonia entre cliente e desenvolvedor sobre a função ilustrativa e natureza efêmera dos protótipos. Ele também sugere o descarte dos protótipos ao final de cada iteração.

A área de desenvolvimento de jogos preenche os requisitos descritos por Pressman (2014) para a utilização do processo de prototipação. Para Manker (2011), as decisões que um *game designer* pode tomar ao longo de um projeto representa um espaço de *design*. Nesse contexto, protótipos são ferramentas de comunicação cujo propósito é chamar a atenção para uma região específica desse espaço. Ele também aponta que protótipos permitem iterações curtas, porque são implementados rapidamente. O baixo período das iterações é importante porque, segundo Manker (2011), Callele, Neufeld e Schneider

(2005) e Fullerton (2008), a engenharia de requisitos no desenvolvimento de jogos é díficil e sensível a mudanças. Segundo Fullerton (2008), protótipos encontram problemas de *design* antes do jogo entrar em produção, quando os custos são altos. Implementam-se protótipos digitais com um problema específico em mente. O objetivo é reduzir os riscos introduzidos pela adição de complexidade ao jogo. Fullerton (2008) e Manker (2011) explicam que a adição de regras a um sistema complexo como jogos produz efeitos difíceis de antecipar.

O objetivo deste trabalho é desenvolver uma aplicação que transforma os códigos de um conjunto de protótipos de jogos implementados em *Python* em uma base de códigos reutilizável. A aplicação extrai classes dos protótipos e identifica e executa refatorações para aumentar a reusabilidade de cada classe. O algoritmo mede a reusabilidade de acordo com as métricas de Goel e Bhatia (2013). A aplicação executa refatorações *Extract Class*, sugeridas por Fowler (1999), baseando-se no algoritmo exposto no capítulo Capítulo 4. Fowler (1999) define *Extract Class* como uma refatoração de código cujo objetivo é dividir as responsabilidades de uma classe alvo com uma nova classe. O algoritmo elaborado para este trabalho é uma versão modificada daquele proposto por Bavota et al. (2014) (ver Capítulo 2) e sua vantagem é que ele preserva hierarquias de classes. O *design* desse algoritmo levou em consideração os desafios de adaptar as métricas de Goel e Bhatia (2013) e as refatorações de Fowler (1999) a uma linguagem dinâmica como o *Python*.

O segundo capítulo apresenta os trabalhos relacionados. O terceiro capítulo é a fundamentação teórica desse trabalho. Na primeira seção delineamos brevemente as definições e o processo de desenvolvimento de jogos digitais. Na segunda seção aludimos a importância da prototipação na área de jogos. Apresentamos o conceito de *game jams* e sua relação com a prototipação de jogos. Ainda no terceiro capítulo apresentamos o conceito de *game engines*, a importância da pesquisa nessa área e as arquiteturas de *game engines* que a literatura aborda.

Num quarto capítulo abordamos a proposta da utilização de protótipos na geração de bases de código reutilizáveis. Também abordamos as métricas de reusabilidade que pesquisamos e as técnicas de refatoração. Em seguida relatamos o processo de desenvolvimento e o *design* da ferramenta. Por último explicitamos o experimento de validação da ferramenta. Finalmente, o quinto capítulo contém os resultados dos experimentos de validação da ferramenta.

2 Trabalhos Relacionados

Começamos a pesquisa por trabalhos relacionados através de palavras-chave como *game engines* e *automated refactoring techniques* na plataforma da CAPES e pelo *Google Scholar*. Em seguida selecionamos as publicações mais recentes de cada área e inspecionamos suas referências para encontrar mais artigos. Repetimos esse último passo em todos os artigos até obtermos um número significativo de publicações. Finalmente, classificamos as publicações em cinco categorias, de *n1* a *n5*, de acordo com sua relevância, ordenados de maior para menor. Apresentamos seções seguintes os artigos da categoria *n1*.

2.1 Automatização de Refatorações

Bavota, Lucia e Oliveto (2011) propõem um método de refatoração *Extract Class* baseado em grafos. Os autores utilizam uma combinação de métricas estruturais e semânticas como pesos em um algoritmo de partição de grafos. A finalidade desse algoritmo é extrair duas classes mais coesas que a original. Um experimento com sistemas conhecidos pela qualidade de *design* comprovou a eficácia do método. Um segundo estudo de caso validou o trabalho em um cenário de uso real, no qual usuários avaliaram a qualidade das refatorações. Os autores concluem que uma combinação de métricas estruturais e semânticas é mais efetiva na extração de classes do que uma das métricas tomada isoladamente. No entanto, o método assume códigos com informações semanticamente relevantes. Um dos experimentos de extração de classes comprova a relativa ineficácia de métricas semânticas em um sistema onde relevância semântica não é prioridade. O mesmo se aplica a protótipos de jogos, cujos códigos refletem a pressa das suas construções. Outra desvantagem do método proposto por Bavota, Lucia e Oliveto (2011) é não levar hierarquias de classes em consideração. O algoritmo também não extrai mais de duas classes por classe.

Fokaefs et al. (2009) usa um algoritmo de *clustering* para realizar a refatoração *Extract Class*. Sua abordagem analisa as dependências estruturais existentes entre as entidades de uma classe a ser refatorada, i.e. atributos e métodos. Usando essa informação os autores computam o conjunto de entidades para cada atributo (i.e., o conjunto de métodos que o usam), e para cada método (i.e., todos os métodos invocados por um método e os conjunto de atributos que este usa) da classe. Em seguida eles computam a distância entre todos os pares de conjuntos de entidades da classe para agrupar grupos coesos de entidades que podem ser extraídos como classes separadas. Os autores usam um algoritmo de agrupamento hierárquico para esse fim. Assim como nossa abordagem, apenas informações estruturais são levadas em consideração. No entanto, como todas as abordagens baseadas em agrupamento hierárquico, o alogritmo requer a definição de um

limiar para podar o *dendograma* gerado. Um dendograma é uma árvore que representa a saída do algoritmo: as folhas da árvore representam as entidades para agrupar, enquanto os nós remanescentes representam os possíveis agrupamentos aos quais as entidades pertencem, até a raiz que representa o grupo que contém todas as entidades. A distância entre grupos mesclados aumenta com o nível da mescla (começando nas folhas até a raiz). Nós de maior nível mesclam grupos cuja distância também é maior (i.e., menos similares). Encontrar o nível certo para podar o dendograma (i.e., determinar o número de grupos) é um problema difícil sem conhecimento prévio da classe a ser refatorada. Fokaefs et al. (2009) tentam mitigar essa questão ao sugerirem diferentes refatorações para cada limiar. Ainda assim, o engenheiro de software precisa analisar as diferentes soluções para identificar a mais adequada.

Simon, Steinbrckner e Lewerentz (2001) implementam uma ferramenta visual baseada em métricas para ajudar o engenheiro de software a identificar componentes que precisam de refatoração. Especificamente, sua abordagem permite indentificar quatro tipos de refatoração: deslocamento de método, deslocamento de atributo, extração de classe e classes implícitas. Simon, Steinbrckner e Lewerentz (2001) também utiliza apenas métricas estruturais.

Bavota et al. (2014) propõem um novo método de extração automática de classes que supera algumas limitações do seu trabalho anterior. O novo método utiliza as mesmas métricas do método anterior para construir uma matriz *método-por-método*, da classe a ser refatorada, que contém a probabilidade que os métodos de cada linha e coluna pertencem a uma mesma classe. Em seguida, o algoritmo constrói subgrafos que representam cadeias de métodos, baseado na matriz. No final do processo cada subgrafo se torna uma nova classe. Os autores verificaram o método em seis sistemas *open-source* e confirmaram seus resultados positivos. Eles também conduziram um estudo de validação com estudantes de mestrado e profissionais da área, que consideraram a ferramenta como útil. Entretanto, o método ainda não considera hierarquias de classes ou linguagens de programação dinâmicas, e ainda assume códigos semanticamente relevantes.

O'Keeffe e Cinnéide (2008) abordam a automatização de refatorações como um problema de busca. Os autores testam diferentes algoritmos de otimização para melhorar a qualidade do código-fonte de dois programas implementados em Java. A qualidade foi medida sob a perspectiva de três fatores: compreensibilidade, reusabilidade e flexibilidade. Esses três fatores eram determinados através da combinação linear de 11 métricas. Cada fator representa uma combinação com coeficientes diferentes. O'Keeffe e Cinnéide (2008) utilizaram uma versão mais precisa das métricas propostas por Bansiya e Davis (2002) (ver seção 2.2). Os resultados de flexibilidade e compreensibilidade foram positivos. Os autores argumentam que os resultados negativos das métricas de reusabilidade sugerem sua ineficácia em algoritmos de busca. Eles concluem que a ferramenta desenvolvida tem

potencial para uso em tarefas complexas de reengenharia. No entanto, diametralmente a Bavota et al. (2014), seu trabalho focou apenas em reestruturação de hierarquias de classes. A ferramenta também não implementa as refatorações *Extract Class* e *Extract Method*. Reusabilidade não foi o foco dos autores, o que explica seus resultados negativos, pois eles mencionam a existência de *trade-offs* entre este e os outros aspectos. Assim como os demais trabalhos dessa área, O'Keeffe e Cinnéide (2008) assumem uma linguagem de tipagem forte no cálculo das medidas de acoplamento.

2.2 Métricas

De acordo com Bansiya e Davis (2002), a literatura da época validou métricas para sistemas orientados a objetos com conjuntos de dados pequenos e não realistas. Segundo os autores, isso levanta dúvidas quanto à aplicabilidade dessas métricas em um âmbito industrial. Eles também citam a inexistência de uma conexão definida entre medições individuais e atributos de qualidade de um produto. Além disso, os autores afirmam que as medições se aplicam apenas a programas já implementados e aludem a necessidade de métricas para a fase de *design* do projeto. Os autores propõem um modelo de qualidade composto por quatro níveis e três mapeamentos entre esses níveis. Os níveis são: atributos de qualidade de *design*; propriedades de *design* orientado a objetos; métricas de *design* orientado a objetos; e componentes de *design* orientado a objetos. Bansiya e Davis (2002) identificam empiricamente os atributos de qualidade: funcionalidade; efetividade; compreensibilidade; extensibilidade; reusabilidade; e flexibilidade. Propriedades de *design* são características tangíveis dos componentes do *design*. Os autores identificam 11 propriedades, como coesão, polimorfismo e encapsulamento. Em seguida, o trabalho apresenta 11 métricas, através das quais essas propriedades são determinadas. Componentes de *design* são classes, métodos, objetos e relações entre eles. Finalmente, os autores mapeiam cada atributo de qualidade a uma combinação linear das métricas, na qual os coeficientes representam o impacto de cada métrica no atributo. Bansiya e Davis (2002) validaram o modelo com várias versões de dois grandes sistemas comerciais. Os autores testaram a capacidade do modelo de prever qualidade de *design* em vários projetos e observaram uma correlação positiva entre as previsões do modelo e avaliações de especialistas. Entretanto, segundo O'Keeffe e Cinnéide (2008), o modelo falha em uma definição eficiente das métricas. O'Keeffe e Cinnéide (2008) afirmam que o trabalho apresenta as métricas em linguagem natural e, frequentemente, de forma ambígua. Além disso, as métricas de Bansiya e Davis (2002) concernem somente à fase de *design*, enquanto o foco do nosso trabalho é o nível de implementação.

Goel e Bhatia (2013) argumentam que reuso é um atributo chave para reduzir custos. Segundo os autores, o departamento de defesa dos Estados Unidos economizaria $300 milhões anualmente se aumentasse seu reuso em 1%. Eles também afirmam que reu-

tilizar componentes de software aumenta a produtividade no desenvolvimento e reduzem custos em até 20%. No entanto, melhoria de qualidade só pode ser compreendida se for medida objetivamente (GOEL; BHATIA, 2013). Os autores utilizam uma combinação das métricas de Chidamber e Kemerer (1994) para prever a reusabilidade de sistemas. Eles mediram três sistemas que apresentavam diferentes estratégias de herança e compararam os resultados com a análise de reuso desses programas. Segundo os autores, os resultados comprovaram a eficácia das métricas ao indicar a herança multinível como a melhor estratégia de reuso. Entretanto, Goel e Bhatia (2013) não provêem uma definição clara das métricas de Chidamber e Kemerer (CK). Sua interpretação da métrica Lack of Cohesion of Methods (LCOM) é subjetiva, por exemplo. O nosso trabalho, por outro lado, propõe expressões determinísticas para essas medidas.

2.3 Game Engines

Freitas et al. (2012) apresentam em seu trabalho um *game engine* extensível, baseado em componentes. Segundo os autores, *game engines* são interfaces de programação otimizadas que aumentam reuso e flexibilidade de arquiteturas de jogos. Eles afirmam que *game engines* geralmente tem uma arquitetura monolítica: entidades do jogo se definem por forte acoplamento e relações de dependência. Isso através de uma hierarquia de classes por múltipla herança, o que causa problemas como colisão de nomes, combinação de métodos e repetição de heranças. Para mitigar esses problemas, *game engines* comerciais permitem a definição de entidades através da agregação de componentes. No entanto, os autores afirmam que o acesso direto a esses componentes em tempo de execução introduz ainda mais acoplamento e problemas de dependência ao jogo. Freitas et al. (2012) apresentam *Gear2D*, um *game engine* baseado em agregação não-acoplada de componentes. Para isso, os autores aplicaram padrões de *design* orientado a objetos. Utilizaram *Composite* (GAMMA et al., 2000), por exemplo, para lidar com o problema de hierarquias de classe longas demais. Validaram o *game engine* resultante através da sua utilização em um jogo desenvolvido para um festival. O trabalho dos autores conseguiu iluminar os benefícios de uma arquitetura de *game engines* orientada a componentes.

2.4 Fechamento

Os trabalhos relacionados a métricas possuem ligações com reusabilidade. Alguns trabalhos, no entanto, não as definem objetivamente, ao contrário do de Bavota et al. (2014). Os métodos de extração de classe existentes na literatura assumem uma linguagem de programação de tipagem forte. Por outro lado, propomos um método para uma linguagem dinâmica, o *Python*. A importância dessa diferença está no fato de que a mioria dos protótipos desenvolvidos para *game jams* são implementados com linguagens dinâmi-

cas. Uma possível explicação é que elas favorecem o desenvolvimento rápido. Quanto aos trabalhos relacionados a *game engines*, vemos uma abordagem de *design top-down* que parte de alguns requisitos chave. Nosso trabalho, no entanto, consiste em uma ferramenta que obtém uma base de códigos reutilizável, a partir de um conjunto de protótipos de jogos.

3 Fundamentação Teórica

3.1 Jogos

Segundo Koster (2004), citado por Gregory (2009), um jogo é uma experiência interativa que oferece ao jogador uma sequência de padrões incrementalmente desafiadores que ele deve aprender e, eventualmente, dominar. Para ele, o reconhecimento de padrões é o cerne daquilo que consideramos divertido.

Salen e Zimmerman (2004) fazem uma revisão abrangente das definições de **jogo**. Ao comparar essas definições, eles evidenciam a falta de um consenso. No entanto, eles também observam a existência de padrões, como regras formais, resolução de conflitos e o cumprimento de objetivos. Traduzimos o conceito apresentado naquele trabalho: "um jogo é um sistema no qual jogadores se engajam em um conflito artificial, definido por regras, que resulta em uma consequência quantificável" (SALEN; ZIMMERMAN, 2004, p. 81).

Um tema recorrente na revisão de Salen e Zimmerman (2004) é a abrangência das definições. Elas podem incluir sistemas que não são jogos, como uma eleição, por exemplo. Em uma eleição um conjunto de candidatos competem entre si para alcançar um objetivo comum, mas não é um conflito artificial. Algumas definições excluem sistemas que são considerados jogos, mas não apresentam um aspecto da definição. Jogo-da-velha e Xadrez não têm o aspecto de **faz-de-conta** atribuído a muitos jogos.

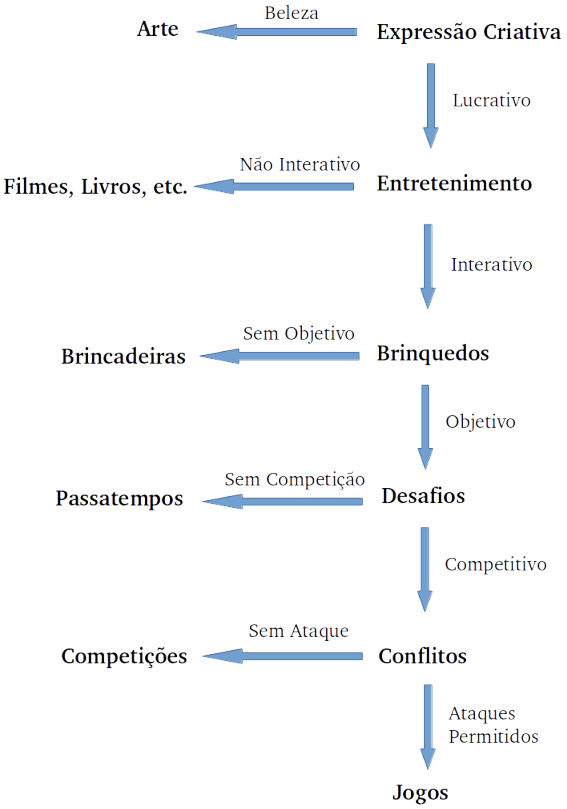
Crawford (2003) tenta contornar os problemas dessa terminologia ao apresentar uma taxonomia simples de expressões criativas, ilustrada na Figura 1. As setas representam as decisões realizadas ao se classificar uma expressão criativa. Jogos, portanto, são expressões motivadas por dinheiro. Também são interativos, onde as interações são dirigidas a objetivos que são alcançados através da superação de desafios. Esses desafios emergem da resolução de conflitos entre competidores, que tentam impedir o progresso um do outro, através de ataques.

3.1.1 Jogos Digitais

Gregory (2009) descreve jogos digitais como simulações não críticas em tempo real. Especificamente, "a maioria dos jogos bi- e tri-dimensionais são exemplos do que cientistas da computação chamariam **simulações de computador interativas não críticas em tempo real baseadas em agentes**" (GREGORY, 2009, p. 9).

O autor explica que simulações são uma representação da realidade através de um

Figura 1 – Taxonomia de expressões criativas.



Fonte: Crawford (2003)

modelo matemático. No entanto, como o objetivo de um jogo é diversão, essa representação não precisa ser fiel. De fato, o papel do *game designer* é criar uma simplificação e aproximação de uma realidade que torne o jogo divertido.

Segundo Gregory (2009), jogos são baseados em agentes por causa da frequente interação entre as entidades que compõem os seus mundos virtuais. Ele cita veículos, personagens, e demais entidades do jogo como exemplos de agentes. O autor afirma que é por esse motivo que a maioria dos jogos modernos são implementados utilizando linguagens orientadas a objetos.

Ainda segundo o autor, jogos precisam atualizar o seu estado interno de acordo com eventos. Esses eventos podem ser originados no mundo virtual do jogo ou a partir das entradas do jogador. Além disso, as respostas do sistema a esses eventos precisam ser em temnpo real. Por isso jogos são **simulações interativas em tempo real**.

Finalmente, o autor afirma que jogos são simulações não críticas, porque a violação de um *deadline* não tem consequências sérias. *Deadlines* (ou prazos), são restrições temporais impostas às iterações de um sistema. Um exemplo de *deadline* é a necessidade

da maioria dos jogos de atualizar a tela em 1/60 segundos.

Gregory (2009) diz que essas simulações utilizam um modelo numérico, ao invés de analítico, para representar a realidade. Modelos analíticos podem ser avaliados por qualquer uma de suas variáveis independentes, dadas apenas as condições iniciais e constantes do modelo. Jogos, no entanto, precisam responder a eventos imprevisíveis e, por isso, são modelos numéricos. Esse modelo é atualizado a cada iteração de um loop principal cronometrado, e os resultados são emitidos através de gráficos, sons ou outras formas de *feedback* de interação.

3.1.1.1 Equipe

O desenvolvimento de jogos digitais é uma atividade multidisciplinar. Isso se reflete na formação das equipes de desenvolvimento. Segundo Gregory (2009), Salen e Zimmerman (2004), Fullerton (2008), Callele, Neufeld e Schneider (2005) e Rabin (2005), elas são compostas por: engenheiros; artistas; *game designers*; produtores; e um time de gerência e apoio.

Segundo Rabin (2005) e Gregory (2009), engenheiros projetam e implementam o jogo. Programadores de execução (*runtime programmers*) implementam o *game engine* e o jogo em si. Programadores de ferramentas (*tools programmers*) focam nas ferramentas utilizadas pelo restante da equipe. Em equipes grandes é provável que existam programadores dedicados exclusivamente a um subsistema, como inteligência artificial e renderização.

Segundo Rabin (2005) e Gregory (2009), artistas criam o conteúdo do jogo. Artistas de conceito, modeladores 3D e artistas de animação trabalham no aspecto visual, enquanto compositores e *designers* de som trabalham na sonorização. A equipe geralmente escolhe um diretor de arte para representar esse departamento nas decisões do projeto.

Game designers, segundo Gregory (2009), projetam a experiência interativa do jogador. Um grupo mais experiente de *designers* trabalha em questões de jogabilidade de alto nível, como a história do jogo e as metas e objetivos do jogador. Um segundo grupo trabalha em áreas específicas do mundo virtual, e projetam o *layout* dos níveis e a localização de itens de interesse. E ainda pode existir um terceiro grupo, que foca nos aspector técnicos do *design*, trabalhando em conjunto com programadores de jogabilidade (*gameplay programmers*) (GREGORY, 2009).

Segundo Noel Llopis (RABIN, 2005), o papel de *designers* e artistas se torna mais importante, a medida que jogos são dirigidos a dados (subseção 3.3.3). Dessa forma, programadores e engenheiros assumem um papel de suporte, implementando ferramentas para criação de conteúdo e mantendo códigos legados, como o *game engine* utilizado.

Rabin (2005) e Gregory (2009) também citam o papel dos produtores. Produtores determinam o cronograma do jogo e gerenciam os recursos humanos. Eles também fazem a ligação entre a equipe de desenvolvimento e a unidade de negócios da companhia. Estúdios pequenos, no entanto, preferem dividir essas responsabilidades entre os demais integrantes e não criam o papel de produtor.

3.2 Prototipação de Jogos

Para Manker (2011), as decisões que um *game designer* pode tomar ao longo de um projeto representa um espaço de *design*. Nesse contexto, protótipos são ferramentas de comunicação cujo propósito é chamar a atenção para uma região específica desse espaço. Ele também aponta que protótipos permitem iterações curtas, porque são implementados rapidamente. O baixo período das iterações é importante porque, segundo Manker (2011), Callele, Neufeld e Schneider (2005) e Fullerton (2008), a engenharia de requisitos no desenvolvimento de jogos é díficil e sensível a mudanças.

Segundo Fullerton (2008), protótipos encontram problemas de *design* antes do jogo entrar em produção, quando os custos são altos. Implementam-se protótipos digitais com um problema específico em mente. O objetivo é reduzir os riscos introduzidos pela adição de complexidade ao jogo. Fullerton (2008) e Manker (2011) explicam que a adição de regras a um sistema complexo como jogos produz efeitos difíceis de antecipar.

3.2.1 *Game Jams*

Game jams são eventos nos quais pequenas equipes devem desenvolver um jogo sob certas restrições. Segundo Musil et al. (2010), *game jams* possuem o seguinte conjunto de regras:

1. O objetivo é prototipar rapidamente jogos experimentais minúsculos e, ao fazer isso, injetar novas ideias na indústria de jogos.

2. Existe um foco temático geral que todos os jogos criados devem compartilhar.

3. Todo mundo que puder contribuir com a produção do jogo pode participar.

4. O evento tem uma duração de 24h-48h.

5. A criação *ad hoc*[1] de equipes é incentivada, e o tamanho deve permanecer entre 2-5 membros.

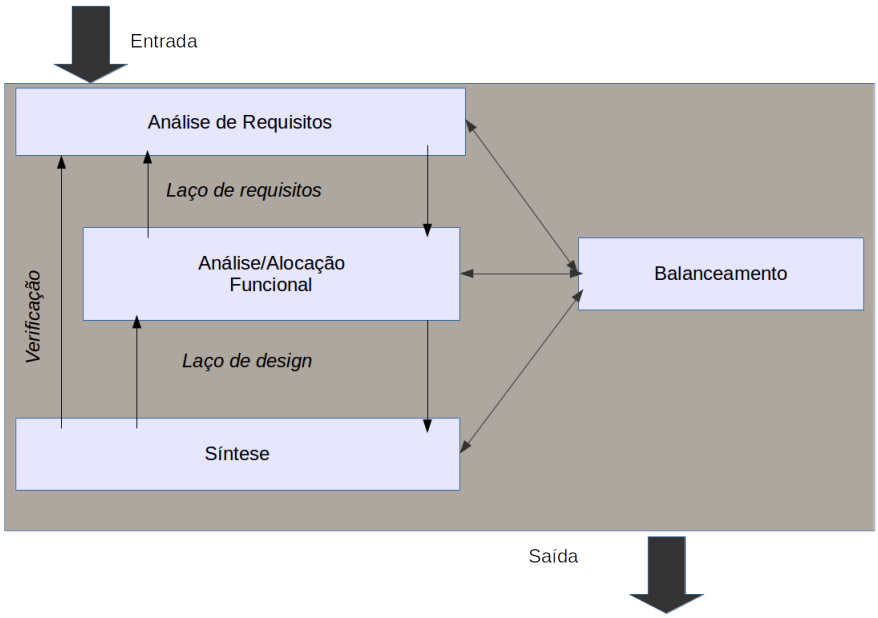
[1] formado, arranjado ou feito apenas para um propósito específico.

6. O evento é agnóstico em relação a software e hardware, de forma que cada equipe pode utilizar as plataformas e ferramentas com as quais são mais proficientes para concretizar sua visão.

7. Ao final do evento, existe uma apresentação pública na qual os jogos são premiados por um painel de juízes especialistas e pela audiência.

O trabalho de Musil et al. (2010) aborda *game jams* sob a perspectiva da prototipação de novos produtos. Uma das suas motivações é a dificuldade em identificar requisitos nas fases iniciais do desenvolvimento de um jogo (CALLELE; NEUFELD; SCHNEIDER, 2005). Eles citam o estabelecimento de *game jams* há mais de uma década, e a existência de jogos de sucesso que tiverem origem nesses eventos, como justificativa da sua abordagem. O objetivo do trabalho era delinear os elementos que dão a *game jams* seu efeito acelerador no desenvolvimento de novos produtos e identificar novas oportunidades de pesquisa.

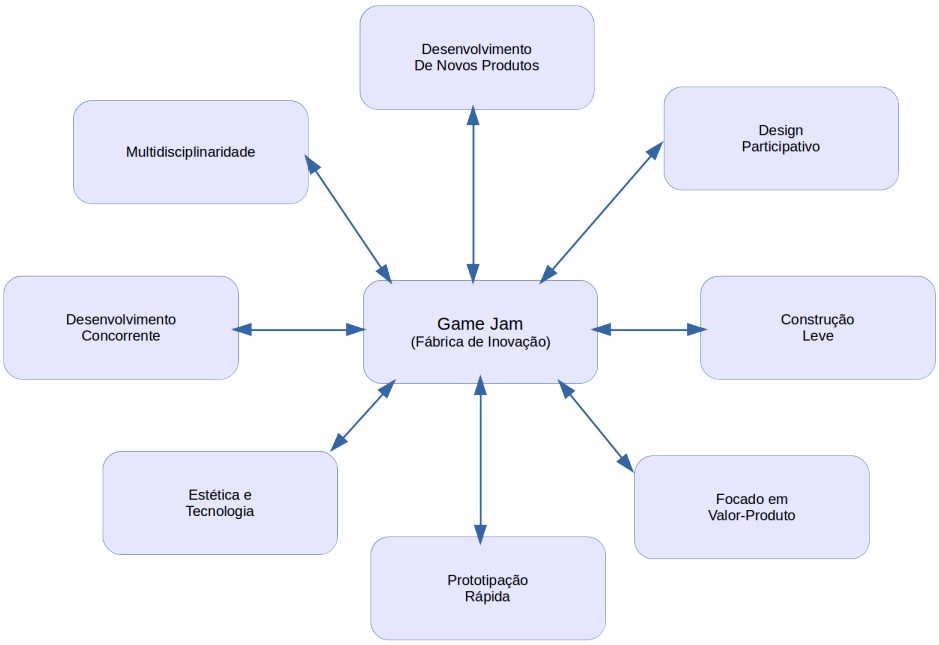
Figura 2 – Processo simplificado de engenharia de sistemas.



Fonte: Musil et al. (2010, p. 184)

Musil et al. (2010) caracterizam *game jams* entre outros processos e técnicas sob três perspectivas: paradigmas de *design* de interações, processos de engenharia de sistemas e conceitos de colaboração existentes. Para eles, *game jams* negam o paradigma tradicional de *design* como produto de um processo racional, guiado por regras e leis científicas. Eles se enquadram melhor em uma visão pragmática, segundo a qual:

Figura 3 – Elementos de um *game jam*.



Fonte: Musil et al. (2010, p. 185)

> (...) o *design* emerge de um sistema auto-organizável, *bricoleur*[2], reflexivo, cujos pontos fortes estão em uma visão e experiência compostas. O produto final é resultado de um diálogo contínuo, uma conversação reflexiva que apresenta uma solução a um problema situacional único, cujas restrições são definidas, principalmente, pelo *designer* (MUSIL et al., 2010, p. 184)

Quanto a processos de desenvolvimento, Musil et al. (2010) reconhecem que *game jams* não se enquadram em nenhum processo conhecido. No entanto, eles identificam a realização de um processo básico de engenharia de sistemas (Figura 2). A equipe resolve um problema de forma iterativa e recursiva, onde os requisitos são balanceados em relação ao tempo de desenvolvimento disponível e à proficiência do time com as ferramentas utilizadas. Iterativa porque o processo se repete para cada funcionalidade. Recursiva, porque cada passo de uma recursão pode incluir os mesmos passos. Os testes realizados são testes de usabilidade, baseados em cenários e focados na avaliação do produto final.

Musil et al. (2010) observam que cada equipe implementa sua interpretação do tema proposto. Como consequência, o conjunto dos produtos finais é uma aproximação abrangente do cenário sugerido pelo tema. Eles concluem que, sob uma macro-perpespectiva, *game jams* são uma abordagem pragmática para protipação de jogos e

[2] Palavra originária do francês, que significa alguém que cria ou constrói a partir de um conjunto diverso de partes disponíveis.

identificação de oportunidades de inovação.

Em seguida, para analisar *game jams* em um nível mais baixo, eles os dividem em oito elementos, ilustrados na Figura 3. Sob a perspectiva da seleção de novos produtos para desenvolvimento, *game jams* são práticos porque oferecem executáveis interativos, ao invés de conceitos.

Segundo Musil et al. (2010), *game jams* oferecem segurança para a sugestão de ideias, pois o risco de uma decisão ser feita por causa de papéis e organizações hierárquicas é baixo. Isso é positivo do ponto de vista de **design participativo**. Devido às restrições de tempo, a equipe favorece ideias que agregam **valor imediato ao produto**, e prefere descartar ideias rapidamente, se o custo percebido for alto. Ela também favorece a **construção leve** do produto, utilizando de bases de código prontas para uso. Isso reduz a complexidade do produto e aumenta a velocidade do desenvolvimento.

Para Musil et al. (2010), *Game jams* são **prototipações rápidas de experiências**, porque favorecem experiências subjetivas e integridade estética a qualidades técnicas. São essas experiências subjetivas que decidem o equilíbrio entre **estética e tecnologia**. Equipes priorizam a utilidade do produto a reusabilidade de código, por exemplo.

Musil et al. (2010) classificam *game jams* como **desenvolvimento paralelo** baseado em conjuntos. Isso significa que os resultados do evento são conjuntos de soluções que aproximam o domínio de um problema. As equipes também são usualmente formadas por membros de diferentes áreas, como programadores, artistas e *designers*. Isso introduz **multidisciplinaridade** ao evento, e colabora com a exploração de alternativas de *design*.

3.3 Game Engines

Segundo Gregory (2009), o termo game engine surgiu na metade dos anos 90 em referência a jogos de tiro em primeira pessoa (first person shooters ou FPS), como o *Doom* da id Software. *Doom* foi arquitetado com uma separação arquitetural entre seus componentes de software principais (como o sistema de renderização 3D, detecção de colisões ou o sistema de áudio) e os artefatos de arte, o mundo do jogo e as regras de jogabilidade que compõem a experiência do jogador.

O objetivo dessa separação era incentivar a formação de uma comunidade de jogadores que pudesse licenciar seus jogos e transformá-los em novos produtos através da criação de novos conteúdos com mínimas alterações no software do *engine*. Isso marcou a origem da comunidade de *modding* - indivíduos e estúdios independentes que construíram novos jogos a partir de ferramentas livres providenciadas pelos desenvolvedores originais.

No final dos anos 90 reuso e *modding* eram objetivos importantes do *design* de jogos digitais. *Engines* eram feitos para serem customizáveis através de linguagens de script, e

o seu licenciamento se tornou uma fonte secundária de lucro para os seus criadores. Hoje, desenvolvedores de jogos podem obter uma licença para um *game engine* e reutilizar porções significativas dos seus componentes de software principais para construir novos jogos.

3.3.1 Definições

Segundo Anderson et al. (2008), o conceito de *game engines* é um problema, pois não existe uma terminologia comum para o desenvolvimento de jogos digitais. Ele cita a confusão entre o termo e o conceito de *rendering engine* (motor de renderização), como se observa em (ZERBST; DUVEL, 2004), e a dificuldade em diferenciar um jogo do seu *game engine*. Em um trabalho posterior ele propõe a seguinte definição para *game engines*: um sistema de software aberto e extensível em que construímos jogos digitais. O autor afirma que ele não oferece dados ou funções que possam ser associados a um jogo específico ou outra aplicação do *game engine*. Ainda segundo Anderson et al. (2008), o que diferencia um *game engine* de uma *Application Programming Interface (API)* é que aquele tem uma camada unificadora que conecta seus componentes. Ou seja, é mais que a soma dos seus componentes e subsistemas. Gregory (2009) faz uma afirmação semelhante ao dizer que o que diferencia *game engines* de bibliotecas e *frameworks* é sua arquitetura dirigida por dados.

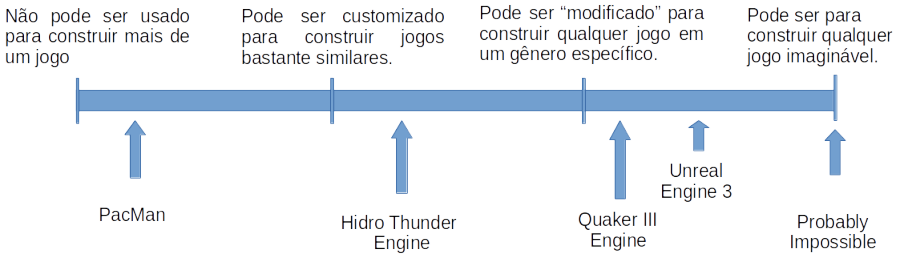
Esse conceito é baseado no artigo de Lewis e Jacobson (2002), no qual a definição está na natureza modular dos jogos. *Game engines* são uma coleção de módulos que não interferem na lógica ou no ambiente de um jogo específico. Gregory (2009) afirma que na maioria dos jogos é difícil separar tais módulos, pois esses conceitos mudam a medida que o design do jogo solidifica-se. Para ele, uma arquitetura dirigida a dados (ver subseção 3.3.3) é o que difere um *game engine* de um pedaço de software que é um jogo, mas não é um *engine*.

Gregory (2009), também afirma que o termo *game engine* deve ser reservado a software que seja extensível e permita a criação de vários jogos diferentes sem grandes modificações. No entanto, o autor admite que essa distinção não é binária e que existe um espectro de reusabilidade no qual um *engine* se enquadra (ver Figura 4).

Gregory (2009) diz que todo projeto de software envolve decisões conflitantes baseadas em pressuposições sobre a plataforma na qual irá executar. Por isso, é provável que nenhum *engine* se enquadre na extrema-direita desse espectro. Ou esses produtos são projetados para um jogo e plataforma específicos ou são otimizados para um gênero.

Nesse trabalho *game engines* são compostas por um conjunto de ferramentas de produção de conteúdo e um componente de execução. Conteúdo são scripts de inteligência artificial, imagens e sons que fazem parte de um jogo. O componente de execução é um

Figura 4 – Espectro de reusabilidade de *game engines*.



Fonte: Gregory (2009, p. 12)

sistema de software dirigido por dados que transforma o conteúdo providenciado pela equipe de desenvolvimento em um jogo digital.

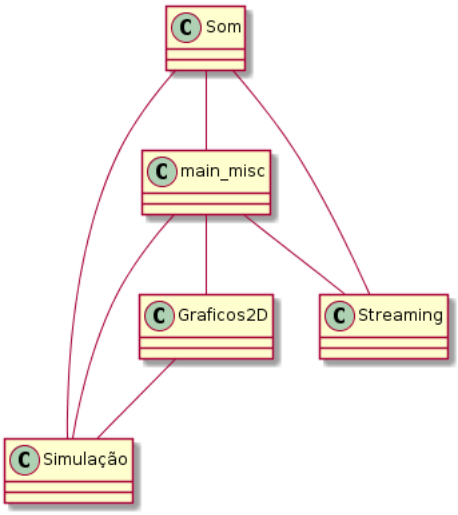
3.3.2 Importância da Pesquisa em *Game Engines*

De acordo com Anderson et al. (2008), existe um consenso de que *game engines*, mais do que úteis, são necessários, devido à complexidade dos jogos modernos. Apesar disso, a literatura da área é escassa e as pesquisas existentes focam em subsistemas como renderização e inteligência artificial.

Lewis Lewis e Jacobson (2002) argumenta a favor do uso de *game engines* na área acadêmica devido ao seu custo baixo em relação a tecnologias especializadas. Esse conceito foi posto à prova em Wang, Lewis e Gennari (2003), no desenvolvimento de uma simulação da arena de procura e resgate urbano por robôs autônomos, do *National Institute of Standars and Technology (NIST)*.

No trabalho de Miao Miao et al. (2011), um *game engine*, Delta3D, é utilizado para desenvolver uma plataforma de computação e modelagem de sistemas de transporte artificiais. A arquitetura de agentes do Delta3D facilitou a simulação de populações artificias. Seu subsistema de rede possibilitou a paralelização das simulações.

Figura 5 – Arquitetura de um jogo 2D em 1994.

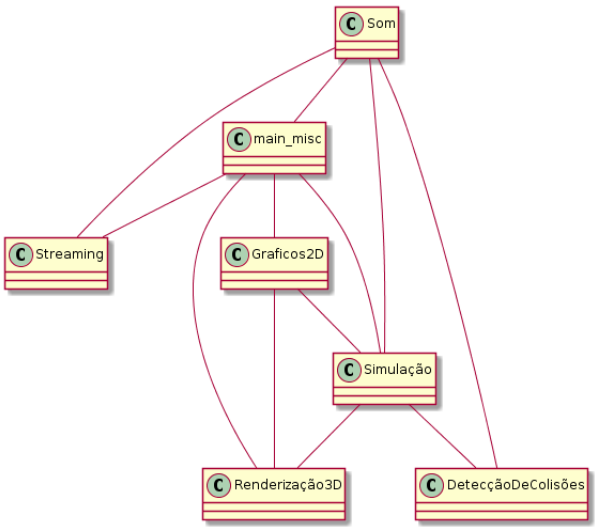


Fonte: Blow (2004, p. 30)

3.3.3 Arquiteturas

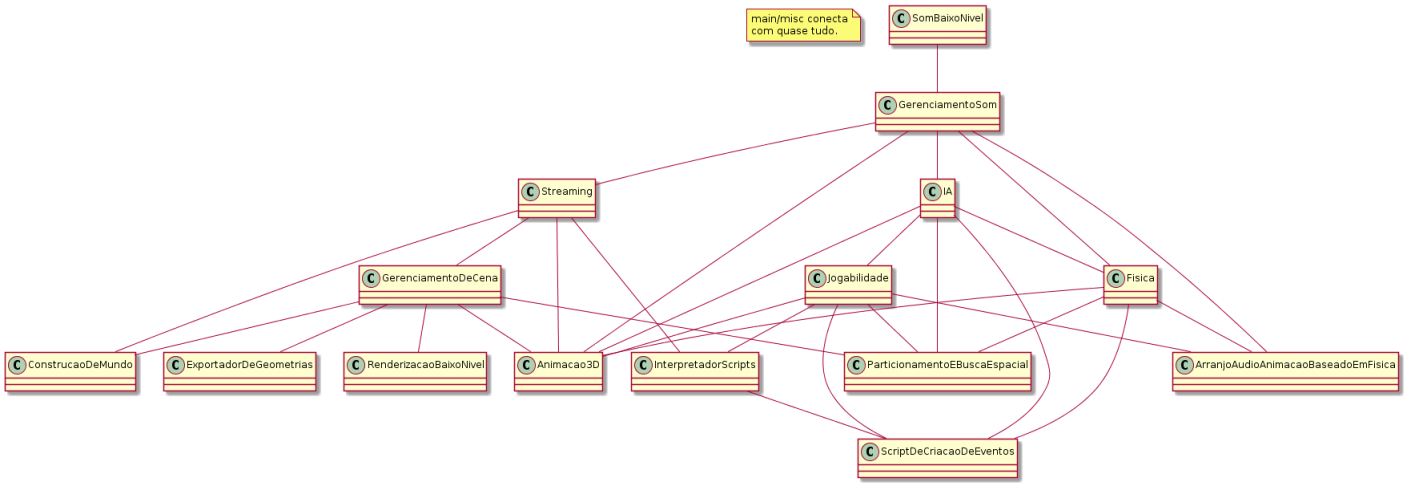
A mudança nas arquiteturas de jogos digitais em um período de dez anos é mostrada por Blow (2004). A Figura 5, Figura 6, e Figura 7 mostram a evolução da complexidade dos jogos digitais durante esse período. A Figura 8 mostra a arquitetura de um *game engine* 3D típico, segundo Gregory (2009).

Figura 6 – Arquitetura de um jogo 3D em 1996.



Fonte: Blow (2004, p. 30)

Figura 7 – Arquitetura de um jogo 3D em 2004.



Fonte: Blow (2004, p. 33)

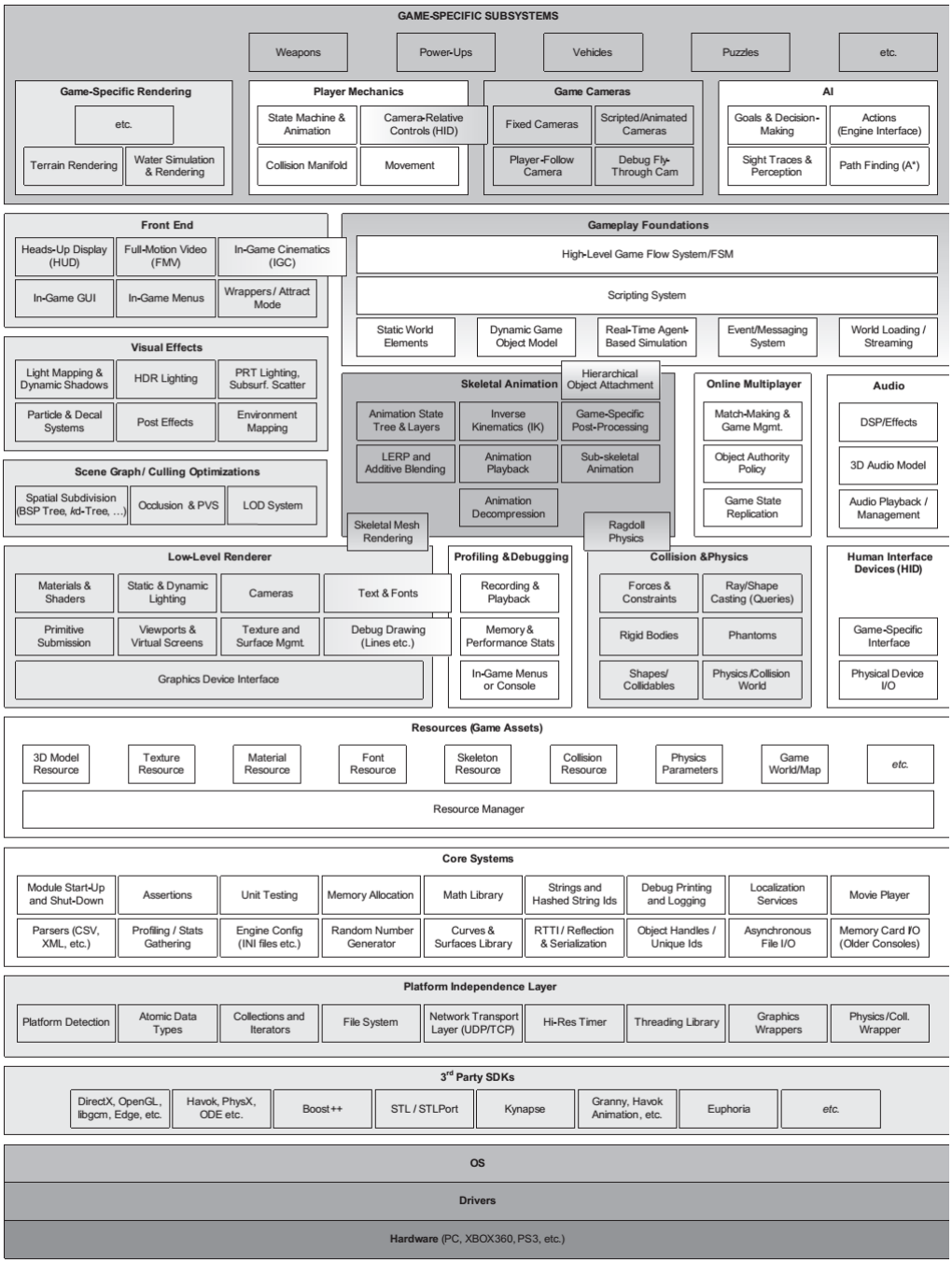
Blow (2004) explica que os nós representam uma área de funcionalidades e as arestas são acoplamentos de conhecimento entre as duas áreas. A Figura 5 é uma aproximação de um jogo 2D em 1994. Variações podem ocorrer, como a inclusão de um módulo de Inteligência Artificial (IA), por exemplo. No caso geral, os comportamentos dos agentes era implementado em **main/misc**. A Figura 6 representa um jogo 3D em 1996, e a Figura 7 representa um jogo moderno de apenas um jogador. Para Blow (2004), o maior desafio para a indústria de jogos, atualmente, são *Massive Multiplayer Game (MMG)*, jogos multi-jogadores acessíveis pela *internet*.

Segundo Gregory (2009), *game engines* variam nos detalhes das suas implementações, mas padrões têm emergido nas suas arquiteturas. A maioria tem similaridades no espectro funcional dos seus componentes. Motores de renderização, de física e colisões, e módulos de Inteligência Artificial se fazem presentes em quase todas os *engines*.

Um *game engine* dirigido a dados significa que o comportamento de um jogo é determinado pelo conteúdo produzido por *game designers* e artistas (GREGORY, 2009). Zerbst e Duvel (2004) conota essa ideia ao fazer uma analogia com motores de carros para explicar a origem do termo game engine. Tanto um motor de carros quanto um motor de jogos abstraem detalhes de baixo nível e são reutilizáveis. Enquanto o motor de um carro tranforma combustível em energia cinética, um motor de jogos transforma descrições de conteúdo em uma experiência interativa.

A vantagem de uma arquitetura dirigida a dados é aumentar a produtividade da equipe de desenvolvimento e reduzir a carga de trabalho do quadro de engenheiros. Outra vantagem é reduzir o tempo de iteração, pois alterações nos conteúdos podem ser testadas rapidamente, sem auxílio de um programador, segundo Gregory (2009). A desvantagem é a complexidade maior do *game engine*, e a escolha de quais componentes devem ser

Figura 8 – Arquitetura de um *game engine*.



Fonte: Gregory (2009, p. 29)

dirigidos por dados é uma decisão fundamental do projeto.

Direção por dados e orientação a dados são coisas diferentes. Segundo Llopis (2009), orientação a dados é um paradigma de design. Nesse paradigma o sistema é visto sob a perspectiva dos seus dados, ao invés de código (classes ou funções). Seu uso no desenvolvimento de jogos digitais é discutido em (LLOPIS, 2009) e (LLOPIS, 2010).

Direção por dados, de acordo com Gregory (2009), significa que o comportamento do sistema é determinado por dados. Um sistema dirigido por dados pode ser construído sob qualquer paradigma.

Gregory (2009) especifica que um *game engine* é constituído de um conjunto de ferramentas e um componente de execução. A Figura 8 representa a arquitetura de um típico componente de execução.

Para Gregory (2009), *Game engines* são construídos em camadas, onde as camadas superiores dependem das camadas inferiores. A três primeiras camadas respresentam interfaces que abstraem funcionalidades das plataformas e sistemas operacionais no topo das quais os jogos executarão. Na quarta camada estão *middlewares*, bibliotecas e *SDKs* [3] utilizados para implementar subsistemas específicos do *engine*. O comportamento desses *middlewares* varia de acordo com a plataforma utilizada. A camada de **independência de plataforma** abstrai essas questões. A sexta camada contém utilidades acessadas frequentemente, como gerenciamento de alocações e desalocações eficientes de memória. **Gerenciamento de recursos** é a camada que unifica os acessos aos ativos físicos do jogo, como arquivos de textura, fontes e configurações de subsistemas.

O motor de renderização, segundo Gregory (2009) é uma das partes mais complexas de um *game engine*. Embora as arquiteturas desse subsistema variem, existem similiaridades entre as implementações modernas. A Figura 8 mostra a divisão em quatro camadas, uma abordagem comum. O **renderizador de baixo nível** é responsável por primitivas gráficas, sem se preocupar com determinações de visibilidade. Esta é a responsabilidade da camada de **grafos de cena**. O objetivo é economizar processamento através da identificação de modelos ou detalhes de modelos que não precisam ser renderizados. Algoritmos e técnicas para **efeitos visuais**, como sombras e sistemas de partículas (fogo, água, fumaça), constituem a terceira camada. *Heads-up displays*, menus, interfaces gráficas e outras formas de superfícies 2D são implementadas na camada de *front end*.

A camada de **colisões e física** impede a interpenetração de corpos rígidos e, assim, permite interações realistas entre os objetos do jogo. Segundo Gregory (2009), essa camada é frequentemente terceirizada, devido a sua complexidade e à maturidade de *middlewares* disponíveis no mercado.

3.4 Métricas de Reusabilidade

As métricas de reusabilidade são uma combinação das métricas de Chidamber e Kemerer (1994), apresentadas em Goel e Bhatia (2013).

Chidamber e Kemerer (1994) apresentaram seis métricas de *design* orientado a

[3] *Software Development Kit*

objetos, medidas em cada classe:

1. **Depth of Inheritance Tree (DIT):** mede a profundidade da classe na árvore de herança.
2. **Number of Children (NOC):** número de subclasses diretas de uma classe.
3. **Coupling Between Objects (CBO)**: número de classes distintas às quais a classe em consideração está acoplada.
4. **Lack of Cohesion of Methods (LCOM):** número de conjuntos disjuntos de métodos locais. Nesse caso, a disjunção é determinada através da presença das variáveis de instância em cada método. Numa classe com coesão total, todas as variáveis de instância são referenciadas em todos os métodos.
5. **Weighted Methods per Class (WMC):** soma das complexidades dos métodos. A métrica de complexidade pode ser qualquer uma, mas normalmente é a complexidade ciclomática.
6. **Response for Class (RFC):** número de métodos de uma classe somado ao número de chamadas de métodos e funções.

No trabalho de Goel e Bhatia (2013), essas métricas foram divididas em três somas que, segundo o autor, tem impacto proporcional na reusabilidade. Elas são:

1. **DIT + NOC**: tem impacto positivo na reusabilidade da classe.
2. **CBO + LCOM**: tem impacto negativo. Ambas as métricas indicam uma possível subdivisão da classe.
3. **WMC + RFC**: tem impacto negativo. Métodos complexos indicam pouca facilidade de modificação.

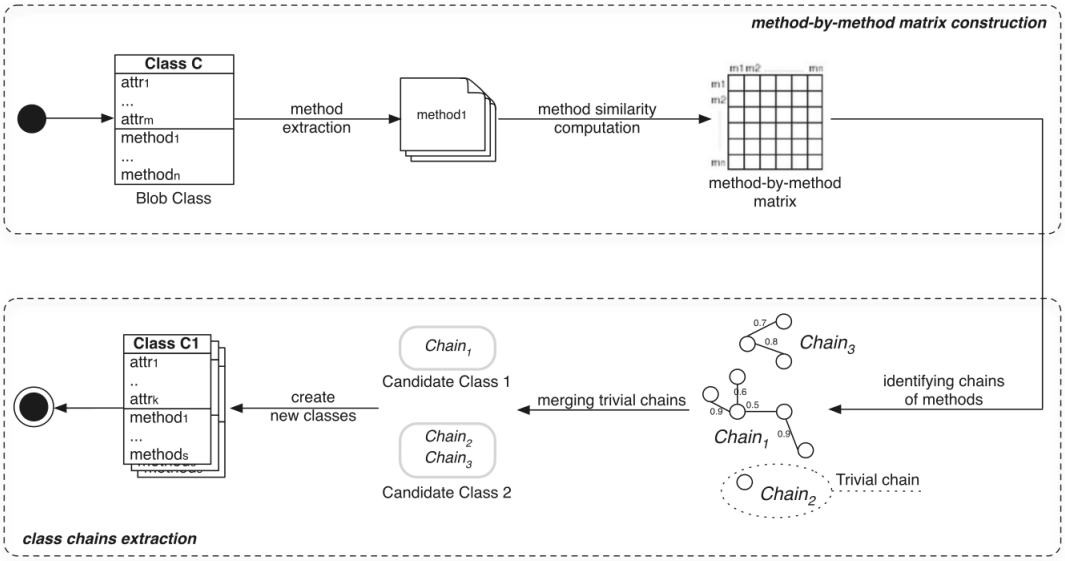
Segundo Goel e Bhatia (2013), multiplos níveis de herança tem maior impacto na reusabilidade de uma classe.

3.5 Automatização de Extract Class

O método para realizar as refatorações *Extract Class* nas classes dos protótipos é uma versão modificada do método de Bavota et al. (2014). O objetivo é identificar classes com baixa coesão e, para cada classe, dividi-la em duas ou mais classes com maior coesão. No entanto, Bavota et al. (2014) explicam que essa extração pode aumentar o acoplamento entre as classes do sistema. Os autores explicam essa relação inversamente proporcional

pelo fato de que tanto as métricas de acoplamento como as de coesão se baseiam em medidas de similaridade entre métodos. Por isso o método utiliza essas medidas a fim de produzir uma solução equilibrada. Ilustramos um *overview* do método na Figura 9.

Figura 9 – Overview do método de extração de classes.



Fonte: Bavota et al. (2014, p. 1624)

A primeira parte da abordagem consiste em montar uma matriz $n \times n$, onde n é o número de métodos da classe a ser refatorada. Uma célula $c_{i,j}$ dessa matriz representa a probabilidade de que os métodos i e j devem estar na mesma classe. Essa matriz representa um grafo cujas arestas representam ligações entre os métodos. Essas ligações serão filtradas por um limiar pré-definido. O resultado dessa filtragem é um conjunto inicial de subgrafos, que representam as classes extraídas. Mesclamos os subgrafos de tamanho pequeno (novamente, de acordo com um limiar pré-estabelecido) com o subgrafo a que mais se acoplam.

A probabilidade de dois métodos pertencerem à mesma classe é uma combinação de métricas. No trabalho de Bavota et al. (2014) e Bavota, Lucia e Oliveto (2011), elas são três: *Structural Similarity between Methods (SSM)*, de Gui e Scott (2008); *Call Based Dependence between Methods (CDM)* (CDM), de Bavota, Lucia e Oliveto (2011); e *Conceptual Similarity between Methods* (CSM), uma métrica semântica proposta por Poshyvanyk et al. (2009). Bavota, Lucia e Oliveto (2011) provaram que essas três métricas são ortogonais entre si. Isso significa que elas capturam dimensões diferentes do acoplamento entre métodos. No entanto, os autores constataram que a terceira métrica depende da qualidade dos comentários presentes no código-fonte, bem como dos nomes de métodos e variáveis. Essas duas características são frequentemente ignoradas no desen-

volvimento de protótipos, como sugerem Pressman (2014), Musil et al. (2010), e Fullerton (2008). Por esse motivo, métricas semânticas serão ignoradas nesse trabalho.

Structural Similarity between Methods (SSM) é uma métrica baseada nas variáveis de instância referenciadas nos métodos. Seja I_i o conjunto de variáveis de instância referenciados pelo método m_i. O SSM de m_i e m_j é a razão entre o número de variáveis de instância que os métodos compartilham e a soma do número de variáveis de instância referenciadas por cada um dos dois métodos (BAVOTA et al., 2014):

$$SSM(m_i, m_j) = \begin{cases} \frac{|I_i \cap I_j|}{|I_i \cup I_j|}, & \textbf{se } |I_i \cup I_j| \neq 0; \\ 0, & \textbf{caso contrário} \end{cases}$$

Bavota, Lucia e Oliveto (2011) apresentou o CDM, que é outra métrica estrutural que considera as chamadas feitas pelos métodos. Seja $calls(m_i, m_j)$ o número de chamadas feitas pelo método m_i ao método m_j e $calls_{in}(m_j)$, o número total de chamadas feitas ao método m_j. Então Bavota et al. (2014) define $CDM_{i \to j}$ como:

$$CDM_{i \to j} = \begin{cases} \frac{calls(m_i, m_j)}{calls_{in}(m_j)}, & \textbf{se } calls_{in}(m_j) \neq 0; \\ 0, & \textbf{caso contrário} \end{cases}$$

Se $CDM_{i \to j} = 1$, significa que o método m_j é chamado apenas pelo método m_i, e, portanto, devem estar na mesma classe. Se $CDM_{i \to j} = 0$, significa que m_i nunca chama m_j e, portanto, podem ficar em classes diferentes sem aumentar o acoplamento do sistema. Para não haver ambiguidade na matriz de métodos, todas as métricas são comutativas. Portanto, Bavota et al. (2014) define CDM como:

$$CDM(m_i, m_j) = max(CDM_{i \to j}, CDM_{j \to i})$$

Após a montagem da matriz, identificamos subgrafos que representam relações estruturais fracas entre os métodos. Para isso, de acordo com Bavota et al. (2014), definimos um limiar mínimo (*minCoupling*) para as arestas e executamos uma filtragem das mesmas:

$$\tilde{c}_{i,j} = \begin{cases} c_{i,j}, & \textbf{se } c_{i,j} > minCoupling; \\ 0, & \textbf{caso contrário} \end{cases}$$

Segundo Bavota et al. (2014), a definição do limiar *minCoupling* se encaixa em duas categorias. A primeira é o limiar constante, que o próprio usuário define manualmente. Embora seja simples de implementar, é difícil saber *a priori* o valor dessa constante que produz os melhores resultados. A segunda categoria é o limiar variável, que depende da classe a ser refatorada. Os autores exemplificam essa categoria através de um *minCoupling* determinado pela mediana dos valores da matriz de métodos. Enquanto essa categoria

resolve os problemas oriundos de um limiar escolhido manualmente, ela também não é facil de calcular. Bavota et al. (2014) concluíram que sua aplicação beneficiou-se mais de um limiar variável. Os resultados dessa filtragem são subgrafos que os autores chamam de *cadeias de métodos*.

Esse conjunto de cadeias de métodos pode incluir cadeias de tamanho insuficiente. Aplicamos uma nova filtragem, através de um limiar *minLength*, que remove tais cadeias. Nesse passo, computamos o acoplamento entre cadeias triviais (as cadeias de tamanho pequeno) e não-triviais. Em seguida, mesclamos cada cadeia trivial com a cadeia não-trivial a qual está mais acoplada. Segundo Bavota et al. (2014), o cálculo do acoplamento entre duas cadeias C_i e C_j é determinado pela média dos acoplamentos entre todos os possíveis pares de métodos entre C_i e C_j:

$$Coupling(C_i, C_j) = \frac{1}{|C_i| \times |C_j|} \sum_{m_i \in C_i, m_j \in C_j} c_{i,j}$$

,

onde $|C_k|$ é o número de métodos pertencentes à cadeia C_k (BAVOTA et al., 2014).

Os métodos das classes originais são então distribuídos pelas classes extraídas. Os atributos também são distribuídos de acordo com como são utilizados por cada método. Nesse ponto, este trabalho difere-se do de Bavota et al. (2014). Propomos um método no qual extraímos as classes e as inserimos na árvore de herança da classe original. A primeira classe extraída é uma subclasse da original e cada classe extraída posteriormente é uma subclasse da classe anterior. O objetivo é aumentar a métrica *DIT* que, segundo Goel e Bhatia (2013), contribui para a reusabilidade do código.

4 Construção e Validação da Ferramenta

O objetivo desse trabalho é desenvolver uma aplicação que refatora um conjunto de protótipos de jogos em uma base de código reutilizável. Para isso, utilizamos uma versão modificada do algoritmo de extração de classes de Bavota et al. (2014). Validamos a aplicação com as métricas de Chidamber e Kemerer (1994), conforme o trabalho de Goel e Bhatia (2013). Para isso, realizamos o experimento descrito na seção 4.2. O código-fonte da versão final desta aplicação está disponível no site *GitHub*[1].

4.1 Aplicação

Desenvolvemos a aplicação em *Python*. O *design* da ferramenta segue o padrão *pipes and filters* descrito por Buschmann et al. (1996). Esse padrão se define por uma cadeia de mensagens em que a saída de um processamento é a entrada do próximo. Ele se justifica pela natureza do método exposto em seção 3.5. *Filters* são componentes que manipulam os dados de uma forma específica. *Pipes* são componentes que conectam os filtros. Os códigos-fonte dos protótipos são o fluxo de dados. Cada etapa do algoritmo descrito na seção 3.5 representa um filtro na arquitetura.

O fluxo de dados inicial da aplicação são os códigos-fonte dos protótipos. O primeiro filtro identifica classes no código bruto, as tranforma em árvores sintáticas abstratas e as aglomera no segundo fluxo de dados. O segundo filtro identifica os nós de método nas árvores e transforma as classes em matrizes método-por-método. Essas matrizes, junto com os nós de métodos e os nós de classe compõem o terceiro fluxo de dados. As métricas descritas na seção 3.5 são aplicadas a cada dois métodos. Logo, o terceiro filtro popula as matrizes com os resultados das medições. No quarto fluxo de dados cada célula da matriz representa a probabilidade que o método linha e o método coluna pertencem a mesma classe. Interpretamos essas matrizes como um grafo, onde os nós são os métodos e as arestas são os resultados das medições. O quarto filtro identifica subgrafos através da remoção de arestas. O filtro remove arestas que são menores que um limiar. Esse limiar é variável, como explicado na seção 3.5. O resultado são cadeias de métodos que compõem o quinto fluxo de dados.

O quinto filtro identifica cadeias trivais, ou seja, subgrafos que contenham um número menor de nós que um valor específico. O filtro aplica as métricas de acoplamento às cadeias, de acordo com a média descrita na seção 3.5. O objetivo é mesclar cada cadeia trivial com a cadeia não trivial a qual está menos acoplada. Dessa maneira, o sexto

[1] https://github.com/wolmir/cristina

fluxo de dados contém apenas subgrafos não triviais que representam as classes extraídas. Finalmente, o sétimo filtro transforma os subgrafos em árvores sintáticas, e o oitavo filtro transforma as árvores em código.

4.1.1 Modelagem da Aplicação

Criamos um subsistema chamado **pypeline** para implementarmos uma versão paralelizada do padrão *Pipes and Filters*. O objetivo é impedir que o processamento dos códigos fique preso a gargalos como a leitura dos arquivos de código-fonte, ou a construção da matriz de métodos de uma classe muito grande. Ilustramos a modelagem desse módulo no diagrama *Unified Modelling Language (UML)* da Figura 10.

Figura 10 – Modelagem UML do módulo **pypeline**.

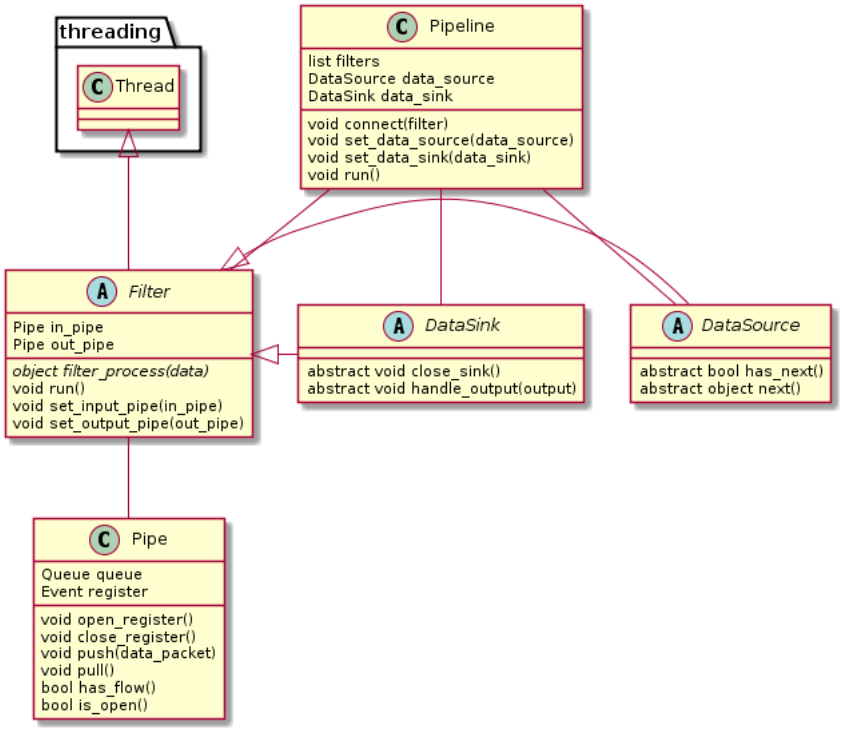
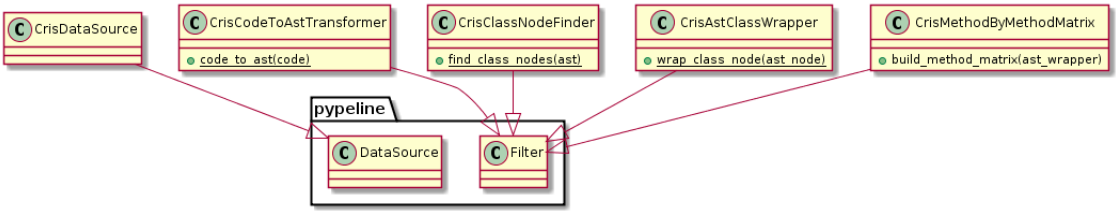


Figura 11 – Modelagem UML da ferramenta **Cristina**.



A classe *Pipeline* gerencia a conexão e a ordem de processamento dos filtros. A conexão entre cada filtro é feita pela classe *Pipe*. Os objetos dessa classe alimentam cada

filtro com os resultados do filtro anterior. Eles também armazenam esses dados até que o próximo filtro esteja pronto para processá-los.

A aplicação se chama **Cristina** e possui onze componentes principais. Cada componente representa um filtro do padrão *Pipes and Filters*. O modelo UML do projeto está na Figura 11 e Figura 12.

A classe *CrisDataSource* lê um arquivo ou uma coleção de arquivos *Python* dentro de um diretório e insere os códigos-fonte no seu *Pipe* de saída. Esse mesmo objeto *Pipe* serve como entrada de dados para *CrisCodeToAstTranformer*, que transforma código *Python* em uma árvore sintática semelhante à da Figura 13.

O filtro *CrisClassNodeFinder* percorre essa árvore para identificar nós de definição de classes (o nó *ClassDef* da Figura 13, por exemplo). O filtro *CrisAstClassWrapper* encapsula esses nós em objetos da classe utilitária *AstClassWrapper* para facilitar a manipulação dos nós. Em seguida, o filtro *CrisMethodByMethodMatrix* monta a matriz de métodos descrita por Bavota et al. (2014).

O próximo componente, *CrisChainsOfMethodsFilter*, usa o parâmetro *minCoupling* como limiar para filtrar as matrizes de métodos. O objetivo é possibilitar a formação de cadeias de métodos pelo filtro *CrisMethodChainsAssembler*. O próximo passo é identificar as cadeias triviais pelo parâmetro *minLength*. O filtro *CrisTrivialChainMerger* realiza esse passo e ainda identifica as cadeias não triviais que se mais se acoplam a cada cadeia trivial e as mescla.

As cadeias são representadas por uma lista de nós de definição de métodos (o nó *FunctionDef* na Figura 13, por exemplo). Essas listas são a entrada de dados do filtro *CrisClassAssembler*, que insere os nós de cada cadeia de métodos em um nó de classe. Finalmente, o filtro *CrisAstToCodeTransformer* transforma esses nós de classe em código-fonte e os passa para *CrisDataSink*, que armazena os códigos em um arquivo **output.py**.

4.2 Experimento

Validamos a aplicação através de um experimento cujo objetivo é testar sua eficácia em extrair classes reutilizáveis. Dividimos o experimento em três partes. Primeiro aplicamos as métricas de Goel e Bhatia (2013) à base de códigos que a aplicação trans-

Figura 12 – Modelagem UML da ferramenta **Cristina**.

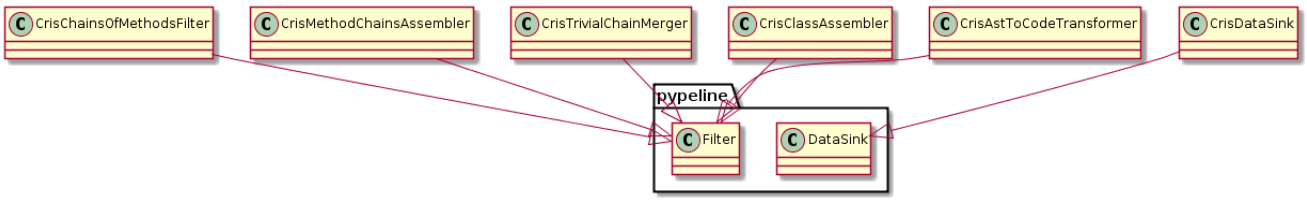
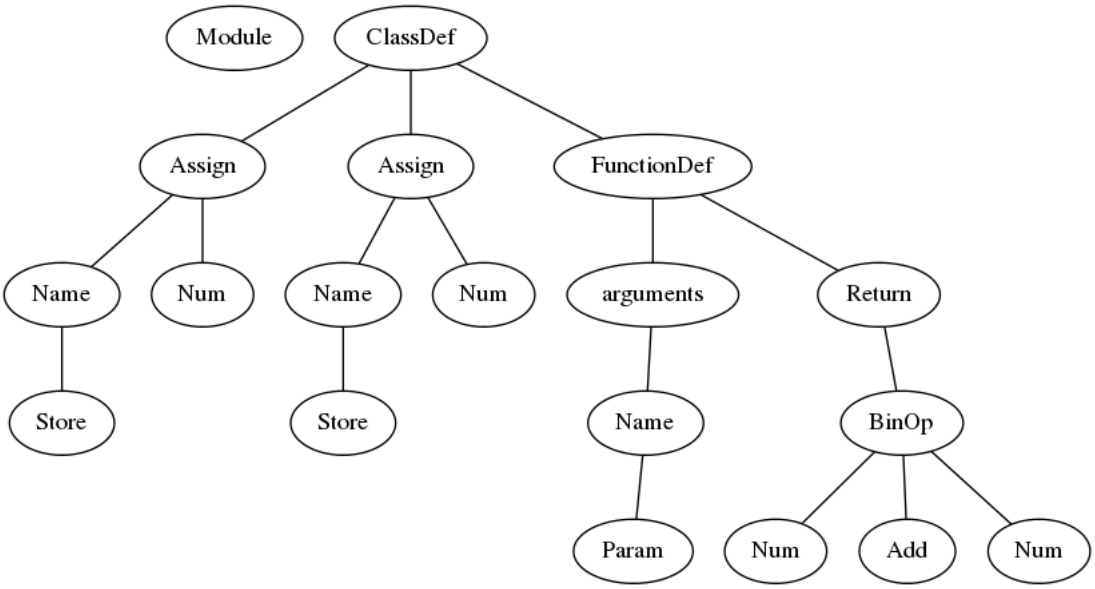


Figura 13 – Exemplo de uma árvore sintática.



formará. Essa base de códigos é uma amostra de 30 arquivos de código fonte *Python*, selecionada aleatoriamente de uma população de 322 arquivos oriundos de submissões para o *game jam PyWeek*. Em seguida executamos a aplicação com combinações dos parâmetros *minCoupling* e dos pesos das métricas *Structural Similarity between Methods (SSM)* e *Call Based Dependence between Methods (CDM)*. Também executamos a aplicação com métricas de coesão que retornam valores aleatórios, a fim de obtermos valores de controle. Utilizamos o seguinte roteiro:

1. Para cada *minCoupling* que varia de 0.0 a 0.9:
   a) Executar a aplicação com peso de *SSM* 0.0 e peso de *CDM* 1.0.
   b) Executar a aplicação com peso de *SSM* 0.2 e peso de *CDM* 0.8.
   c) Executar a aplicação com peso de *SSM* 0.5 e peso de *CDM* 0.5.
   d) Executar a aplicação com peso de *SSM* 0.8 e peso de *CDM* 0.2.
   e) Executar a aplicação com peso de *SSM* 1.0 e peso de *CDM* 0.0.
   f) Executar a aplicação com uma métrica de coesão que retorne valores aleatórios.
2. Aplicar as métricas de reusabilidade nas classes resultantes e compará-las com a base de referência.

Obtemos valores de controle quando executamos a aplicação com uma métrica que retorne valores de coesão aleatórios. O objetivo primário é verificar se a simples divisão dos métodos de uma classe contribui significativamente para o aumento da reusabilidade. O objetivo secundário é observar a eficácia das métricas de Bavota et al. (2014) em otimizar a reusabilidade das classes.

Implementamos as métricas de Chidamber e Kemerer (1994) para obtermos as métricas de Goel e Bhatia (2013). Calculamos a métrica *DIT* de forma recursiva, começando pela classe folha e subindo a árvore até chegar em um ponto sem saída, seja porque chegou em uma classe raiz ou porque a superclasse não se encontra na base de códigos. Como *Python* suporta múltipla herança, o valor final é o caminho mais comprido da classe folha até uma classe raiz. Calculamos a métrica *NOC* contando as subclasses diretas da classe alvo.

A métrica *CBO* apresenta dificuldades em sua implementação devido à natureza dinâmica da linguagem *Python*. Isso se deve ao fato de que o tipo dos objetos referenciados só é conhecido em tempo de execução. No entanto, implementamos uma aproximação dessa métrica ao construirmos um grafo cujos nós são as referências a variáveis externas à classe, e as arestas são o número de variáveis locais e de instância que compartilham essas referências. Em seguida, identificamos subgrafos desconexos, e o seu número representa o mínimo de interfaces distintas das quais a classe depende.

A métrica *LCOM* consiste no número de pares ordenados de métodos da classe que compartilham variáveis de instância, dividido pelo número total de pares de métodos da classe. Subtraímos esse valor de 1.0, a fim de obter a falta de coesão dos métodos. A métrica *WMC* é simplesmente a soma das complexidades ciclomáticas de cada método da classe. Por fim, a métrica *RFC* é a soma do número de métodos de uma classe com o números de chamadas a funções e métodos externas à classe. O último passo do experimento é calcular a média das medições das bases de códigos. Finalmente, comparamos as médias das classes extraídas com os resultados da base original.

5 Resultados

A aplicação obteve sucesso ao extrair a base de códigos reutilizável. Ilustramos os resultados da extração de classes na Figura 14 e Figura 15. A Figura 14 representa a classe original, retirada da amostra de protótipos. A segunda figura representa a classe que a aplicação extraiu desse código. Observamos que a classe extraída é uma subclasse de *CarbonScreenMode0*. Essa última classe foi extraída anteriormente da mesma classe original. Isso demonstra o mecanismo de preservação da àrvore de herança do método proposto nesse trabalho.

Figura 14 – Exemplo de classe da base original.

```
207 class CarbonScreenMode(ScreenMode):
208     def __init__(self, screen, mode):
209         super(CarbonScreenMode, self).__init__(screen)
210         self.mode = mode
211         self.width = self._get_long('Width')
212         self.height = self._get_long('Height')
213         self.depth = self._get_long('BitsPerPixel')
214         self.rate = self._get_long('RefreshRate')
215     def _get_long(self, key):
216         kCFNumberLongType = 10
217         cfkey = create_cfstring(key)
218         number = carbon.CFDictionaryGetValue(self.mode, cfkey)
219         if (not number):
220             return None
221         value = c_long()
222         carbon.CFNumberGetValue(number, kCFNumberLongType, byref(value))
223         return value.value
```

Figura 15 – Exemplo de classe extraída.

```
194 class CarbonScreenMode1(CarbonScreenMode0):
195     mode = None
196     def _get_long(self, key):
197         kCFNumberLongType = 10
198         cfkey = create_cfstring(key)
199         number = carbon.CFDictionaryGetValue(self.mode, cfkey)
200         if (not number):
201             return None
202         value = c_long()
203         carbon.CFNumberGetValue(number, kCFNumberLongType, byref(value))
204         return value.value
205     ' --> EXTRACTED <-- '
```

Os resultados estão nas seguintes três tabelas. A Tabela 1 representa as medições da base de códigos original. A Tabela 2 e a Tabela 3 são os resultados das bases extraídas. Nessas tabelas, *mC* se refere ao parâmetro *minCoupling*, *w-ssm* e *w-cdm* se referem aos pesos das métricas de coesão. Na Tabela 4 estão os valores de controle, obtidos através da execução da aplicação com uma métrica de coesão que retorna valores aleatórios.

O primeiro fato a observar é o aumento da métrica Number of Children (NOC) em relação à base de referência. O valor mínimo dessa métrica obtido pós-extração foi de 0.4899, com os parâmetros mC em 0.1, e os pesos w-ssm e w-cdm em 0.5. Mesmo assim, foi um aumento de 155%. Segundo Goel e Bhatia (2013), quanto maior o valor de NOC, melhor para a reusabilidade.

A métrica Coupling Between Objects (CBO), por outro lado, tem impacto negativo na reusabilidade. Observamos na Tabela 2 e Tabela 3 que esse valor diminuiu com as extrações, embora em alguns casos ele tenha aumentado. O melhor caso, 0.4457 com mC em 0.9 e os pesos em 0.5, representa uma melhoria de quase 20% no acoplamento das classes.

Assim como CBO, pouca coesão entre métodos indicam uma sobrecarga de responsabilidades em uma classe, o que afeta negativamente a reusabilidade. Os resultados mostram que em muitos casos a extração produziu classes totalmente coesas. No pior dos casos, houve uma diminuição de 67% em relação à base de códigos original. Esse valor está na Tabela 2, com mC em 0.2 e os dois pesos em 0.5.

As métricas Weighted Methods per Class (WMC) e Response for Class (RFC) são diretamente proporcionais ao número de métodos de uma classe. Propomos uma técnica que extrai classes e divide os métodos originais entre as classes extraídas. Logo, faz sentido que essas métricas diminuíram significativamente em relação à base de referência. Mas comparamos esses valores com os da Tabela 4, a tabela de controle, e observamos que a extração aleatória de métodos tem pouco impacto na melhoria da reusabilidade.

Isso porque para uma classe com n métodos há múltiplas formas de rearranjar esses métodos em classes distintas. Para cada configuração de classes extraídas, há um conjunto de medições CK correspondente que nem sempre são melhores do que as medições originais. Esse caso acontece quando a minoria das classes extraídas contém a maioria dos métodos mais acoplados ou menos coesos. Esse fato comprova que as métricas de coesão propostas por (BAVOTA et al., 2014) têm uma função otimizadora, porque ajudam a encontrar a configuração de métodos que é mais reutilizável.

Base de Códigos Original	
Métrica	**Valor**
DIT	0.1489
NOC	0.1915
CBO	2.9433
LCOM	0.0319
WMC	6.8582
RFC	8.5745

Tabela 1 – Medições da base de códigos original.

BC Ext.								
mC	**w-ssm**	**w-cdm**	**dit**	**noc**	**cbo**	**lcom**	**wmc**	**rfc**
0.0000	0.0000	1.0000	0.000000	0.7873	2.5318	0.0000	2.8567	3.5716
0.0000	0.2000	0.8000	0.000000	0.7870	2.5325	0.0000	2.8609	3.5769
0.0000	0.5000	0.5000	0.000000	0.8125	2.4596	0.0000	2.5182	3.1484
0.0000	0.8000	0.2000	0.000000	0.7715	2.4617	0.0063	3.0266	3.7840
0.0000	1.0000	0.0000	0.001555	0.7729	2.4495	0.0062	3.0078	3.7605
0.1000	0.0000	1.0000	0.000000	0.7530	2.6261	0.0000	3.3173	4.1475
0.1000	0.2000	0.8000	0.000000	0.6408	2.8908	0.0073	4.6942	5.8689
0.1000	0.5000	0.5000	0.000000	0.4899	3.1520	0.0068	6.5338	8.1689
0.1000	0.8000	0.2000	0.000000	0.5065	3.1209	0.0065	6.3203	7.9020
0.1000	1.0000	0.0000	0.000000	0.5929	2.8142	0.0055	5.2842	6.6066
0.2000	0.0000	1.0000	0.000000	0.7584	2.6225	0.0000	3.2450	4.0570
0.2000	0.2000	0.8000	0.000000	0.7596	2.6093	0.0000	3.2287	4.0367
0.2000	0.5000	0.5000	0.000000	0.6074	2.9098	0.0106	5.1300	6.4138
0.2000	0.8000	0.2000	0.000000	0.6508	2.6865	0.0095	4.5938	5.7435
0.2000	1.0000	0.0000	0.000000	0.6294	2.7164	0.0075	4.8109	6.0149
0.3000	0.0000	1.0000	0.000000	0.7604	2.6106	0.0000	3.2180	4.0233
0.3000	0.2000	0.8000	0.000000	0.7677	2.5823	0.0000	3.1194	3.9000
0.3000	0.5000	0.5000	0.000000	0.6884	2.7495	0.0063	4.0716	5.0905
0.3000	0.8000	0.2000	0.000000	0.6920	2.6667	0.0105	4.0802	5.1013
0.3000	1.0000	0.0000	0.000000	0.6689	2.6892	0.0090	4.3559	5.4459
0.4000	0.0000	1.0000	0.000000	0.7700	2.5751	0.0000	3.0895	3.8626
0.4000	0.2000	0.8000	0.000000	0.7791	2.5475	0.0000	2.9663	3.7086
0.4000	0.5000	0.5000	0.000000	0.7121	2.6848	0.0058	3.7626	4.7043
0.4000	0.8000	0.2000	0.000000	0.7483	2.5414	0.0069	3.3345	4.1690
0.4000	1.0000	0.0000	0.000000	0.7120	2.6233	0.0099	3.8146	4.7692

Tabela 2 – Medições das bases de códigos extraídas.

B. Ext. Cont.								
mC	**w-ssm**	**w-cdm**	**dit**	**noc**	**cbo**	**lcom**	**wmc**	**rfc**
0.5000	0.0000	1.0000	0.000000	0.7867	2.5378	0.0000	2.8652	3.5822
0.5000	0.2000	0.8000	0.000000	0.7864	2.5386	0.0000	2.8694	3.5875
0.5000	0.5000	0.5000	0.000000	0.8123	2.4615	0.0000	2.5215	3.1525
0.5000	0.8000	0.2000	0.000000	0.7583	2.5348	0.0066	3.2020	4.0033
0.5000	1.0000	0.0000	0.000000	0.7595	2.5239	0.0066	3.1862	3.9835
0.6000	0.0000	1.0000	0.000000	0.7867	2.5378	0.0000	2.8652	3.5822
0.6000	0.2000	0.8000	0.000000	0.7879	2.5361	0.0000	2.8483	3.5611
0.6000	0.5000	0.5000	0.000000	0.8151	2.4519	0.0000	2.4827	3.1040
0.6000	0.8000	0.2000	0.000000	0.7660	2.5128	0.0064	3.0994	3.8750
0.6000	1.0000	0.0000	0.000000	0.7630	2.5195	0.0065	3.1396	3.9253
0.7000	0.0000	1.0000	0.000000	0.7879	2.5361	0.0000	2.8483	3.5611
0.7000	0.2000	0.8000	0.000000	0.7879	2.5361	0.0000	2.8483	3.5611
0.7000	0.5000	0.5000	0.873116	0.8191	2.4460	0.0000	2.4296	3.0377
0.7000	0.8000	0.2000	0.000000	0.7750	2.4838	0.0062	2.9800	3.7257
0.7000	1.0000	0.0000	0.007874	0.7701	2.5039	0.0063	3.0457	3.8079
0.8000	0.0000	1.0000	0.000000	0.7879	2.5361	0.0000	2.8483	3.5611
0.8000	0.2000	0.8000	0.000000	0.8127	2.4603	0.0000	2.5150	3.1443
0.8000	0.5000	0.5000	0.011111	0.8222	2.4457	0.0000	2.3877	2.9852
0.8000	0.8000	0.2000	0.000000	0.8220	2.4462	0.0000	2.3906	2.9889
0.8000	1.0000	0.0000	0.000000	0.7747	2.4815	0.0062	2.9846	3.7315
0.9000	0.0000	1.0000	0.000000	0.7879	2.5361	0.0000	2.8483	3.5611
0.9000	0.2000	0.8000	0.000000	0.8220	2.4462	0.0000	2.3906	2.9889
0.9000	0.5000	0.5000	0.000000	0.8222	2.4457	0.0000	2.3877	2.9852
0.9000	0.8000	0.2000	0.000000	0.8222	2.4457	0.0000	2.3877	2.9852
0.9000	1.0000	0.0000	0.000000	0.7754	2.4831	0.0062	2.9754	3.7200

Tabela 3 – Continuação das Medições das bases extraídas.

V. de Controle						
mC	**dit**	**noc**	**cbo**	**lcom**	**wmc**	**rfc**
0.00	0.000000	0.3260	3.9163	0.0264	8.5198	10.6520
0.10	0.000000	0.0253	5.0253	0.0506	12.2405	15.3038
0.20	0.000000	0.0723	4.8313	0.0361	11.6506	14.5663
0.30	0.000000	0.0778	4.8383	0.0539	11.5808	14.4790
0.40	0.000000	0.1250	4.6307	0.0341	10.9886	13.7386
0.50	0.000000	0.2165	4.3763	0.0361	9.9691	12.4639
0.60	0.000000	0.3108	3.9865	0.0315	8.7117	10.8919
0.70	0.003559	0.4733	3.5765	0.0320	6.8826	8.6050
0.80	0.002967	0.5608	3.2522	0.0267	5.7389	7.1751
0.90	0.006342	0.6913	2.9767	0.0085	4.0888	5.1121

Tabela 4 – Valores de Controle

6 Conclusão

Game engines são ferramentas importantes para a indústria de desenvolvimento de jogos. No entanto, o seu preço proibitivo é a causa da preferência de desenvolvedores pela reutilização do próprio código. Portanto, reusabilidade é um fator importante para esse setor do mercado. O papel da prototipação no processo de desenvolvimento de jogos é outro fator importante. Nossa pesquisa da literatura relacionada revelou que a prototipação é mais que uma ferramenta para comunicação de conceitos interativos. Ela assume um caráter construtivo ao ser utilizada para explorar um espaço de *design* que. até então, seria abstrato. Confirmamos esse caráter exploratório com trabalhos acadêmicos que demonstram a equivalência entre prototipação de jogos e submissão de jogos em *game jams*, que são competições com temas altamente experimentais. Uma consequência dessa equivalência é a alta disponibilidade de códigos-fonte de protótipos de jogos na internet. Essa disponibilidade, aliada à necessidade de reutilização de código, foi a motivação do nosso trabalho. Para saciá-la, construímos uma ferramenta para transformação de códigos fonte de protótipos de jogos em um conjunto de classes mais reutilizáveis.

Para isso, pesquisamos técnicas de refatoração automática que melhorem a reusabilidade. Os trabalhos que estudamos, no entanto, abordam o impacto dessas refatorações na qualidade como um todo. Mas também observamos que os fatores de qualidade explorados pelos autores são conectados. Por exemplo, a redução da complexidade de código reduz o seu número de defeitos, mas também aumenta sua reusabilidade. Baseado nesse fato, selecionamos o trabalho de Bavota et al. (2014) que apresenta uma técnica de refatoração automática baseada na extração de classes por coesão de métodos. A motivação dos autores era aumentar a qualidade geral do código, mas concluímos que a técnica tem impacto significativo na reusabilidade das classes.

Observamos esse mesmo efeito na utilização de métricas de código orientado a objeto que os autores utilizam para validar seus esforços. As métricas capturavam várias dimensões da qualidade ao mesmo tempo, incluindo reusabilidade. Exploramos esse efeito ao utilizarmos métricas de código orientado a objeto para validar nossa aplicação. Nossos resultados demonstram a eficácia da aplicação em identificar classes nos protótipos e dividi-las em várias classes mais reutilizáveis. Nossa principal conclusão, portanto, é que a ferramenta tem potencial econômico porque reaproveita subprodutos de desenvolvimento para reduzir custos. Esse fato se baseia no trabalho de Goel e Bhatia (2013) e na pesquisa de DeLoura (2011).

Entretanto, há vários melhoramentos desse trabalho a serem feitos em trabalhos futuros. Entre eles, estão o estudo do impacto do parâmetro *minLength* na aplicação. Tam-

bém podemos melhorar o método de refatoração e não só extrair classes, como organizá-las em uma arquitetura característica de *game engines*. Bavota et al. (2014) explorou métricas de coesão semânticas que optamos por não utilizar. O estudo do impacto dessas métricas é outro trabalho futuro. Por fim, podemos validar a utilidade da ferramenta com desenvolvedores de jogos.

Referências

ANDERSON, E. F. et al. The case for research in game engine architecture. In: KAPRALOS, B.; KATCHABAW, M.; RAJNOVICH, J. (Ed.). *Proceedings of the 2008 Conference on Future Play: Research, Play, Share, Future Play 2008, Toronto, Ontario, Canada, November 3-5, 2008*. ACM, 2008. p. 228–231. ISBN 978-1-60558-218-4. Disponível em: <http://doi.acm.org/10.1145/1496984.1497031>. Citado 2 vezes nas páginas 40 e 41.

ANDERSON, E. F. et al. Choosing the infrastructure for entertainment and serious computer games - a whiteroom benchmark for game engine selection. In: *Games and Virtual Worlds for Serious Applications (VS-GAMES), 2013 5th International Conference on*. [S.l.: s.n.], 2013. p. 1–8. Citado na página 25.

BANSIYA, J.; DAVIS, C. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, v. 28, n. 1, p. 4–17, jan. 2002. Citado 2 vezes nas páginas 28 e 29.

BAVOTA, G. et al. Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering*, v. 19, n. 6, p. 1617–1664, 2014. Disponível em: <http://dx.doi.org/10.1007/s10664-013-9256-x>. Citado 14 vezes nas páginas 26, 28, 29, 30, 46, 47, 48, 49, 51, 53, 54, 58, 61 e 62.

BAVOTA, G.; LUCIA, A. D.; OLIVETO, R. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, v. 84, n. 3, p. 397–414, 2011. Disponível em: <http://dx.doi.org/10.1016/j.jss.2010.11.918>. Citado 3 vezes nas páginas 27, 47 e 48.

BLOW, J. Game development: Harder than you think. *ACM Queue*, v. 1, n. 10, p. 28–37, 2004. Disponível em: <http://doi.acm.org/10.1145/971564.971590>. Citado 2 vezes nas páginas 42 e 43.

BUSCHMANN, F. et al. *Pattern-Oriented Software Architecture: A System of Patterns*. [S.l.]: John Wiley & Sons, New York, 1996. Citado na página 51.

CALLELE, D.; NEUFELD, E.; SCHNEIDER, K. Requirements engineering and the creative process in the video game industry. In: *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*. [S.l.: s.n.], 2005. p. 240–250. Citado 4 vezes nas páginas 26, 35, 36 e 37.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, v. 20, n. 6, p. 476–493, 1994. Citado 4 vezes nas páginas 30, 45, 51 e 55.

CRAWFORD, C. *Chris Crawford on Game Design*. [S.l.]: New Riders Publishing, 2003. Citado 2 vezes nas páginas 33 e 34.

DELOURA, M. Game engine survey 2011. *Game Developer Magazine*, v. 18, n. 5, p. 7–14, 2011. Citado 2 vezes nas páginas 25 e 61.

FOKAEFS, M. et al. Decomposing object-oriented class modules using an agglomerative clustering technique. IEEE, p. 93–101, 2009. Disponível em: <http://dx.doi.org/10.1109/ICSM.2009.5306332>. Citado 2 vezes nas páginas 27 e 28.

FOWLER, M. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999. (Addison Wesley object technology series). ISBN 978-0-201-48567-7. Disponível em: <http://martinfowler.com/books/refactoring.html>. Citado na página 26.

FREITAS, L. G. de et al. Gear2D: an extensible component-based game engine. In: EL-NASR, M. S.; CONSALVO, M.; FEINER, S. K. (Ed.). *FDG*. ACM, 2012. p. 81–88. ISBN 978-1-4503-1333-9. Disponível em: <http://dl.acm.org/citation.cfm?id=2282338>. Citado na página 30.

FULLERTON, T. *Game Design Workshop, Second Edition: A Playcentric Approach to creating Innovative Games*. [S.l.]: Morgan Kaufmann, 2008. Citado 4 vezes nas páginas 26, 35, 36 e 48.

GAMMA et al. *Design Patterns Elements of Reusable Object-Oriented Software*. Massachusetts: Addison-Wesley, 2000. ISBN 0-201-63361-2. Citado na página 30.

GOEL, B. M.; BHATIA, P. K. Analysis of reusability of object-oriented systems using object-oriented metrics. *ACM SIGSOFT Software Engineering Notes*, v. 38, n. 4, p. 1–5, 2013. Disponível em: <http://doi.acm.org/10.1145/2492248.2492264>. Citado 11 vezes nas páginas 26, 29, 30, 45, 46, 49, 51, 53, 55, 58 e 61.

GREGORY, J. *Game Engine Architecture*. Wellesley, Masachusetts: Transatlantic Publishers, 2009. ISBN 978-1-56881-413-1. Citado 11 vezes nas páginas 33, 34, 35, 36, 39, 40, 41, 42, 43, 44 e 45.

GUI, G.; SCOTT, P. D. New coupling and cohesion metrics for evaluation of software component reusability. In: *ICYCS*. IEEE Computer Society, 2008. p. 1181–1186. Disponível em: <http://dx.doi.org/10.1109/ICYCS.2008.270>. Citado na página 47.

KOSTER, R. *A Theory of Fun for Game Design*. Phoenix, AZ: Paraglyph, 2004. Citado na página 33.

LEWIS, M.; JACOBSON, J. Game engines in scientific research. *Communications of the ACM*, v. 45, n. 1, p. 27–29, 2002. Citado 2 vezes nas páginas 40 e 41.

LLOPIS, N. Data oriented design: Or why you might be shooting yourself in the foot with object oriented programming. *Game Developers Magazine*, v. 16, n. 8, p. 43–45, 2009. Citado na página 44.

LLOPIS, N. Data oriented design: Now and in the future. *Game Developers Magazine*, v. 17, n. 8, p. 31–33, 2010. Citado na página 44.

MANKER, J. Game design prototyping. *Games and Innovation Research Seminar 2011 Working Papers*, v. 1, n. 1, p. 41–48, 2011. Citado 3 vezes nas páginas 25, 26 e 36.

MIAO, Q. et al. A game-engine-based platform for modeling and computing artificial transportation systems. *Intelligent Transportation Systems, IEEE Transactions on*, v. 12, n. 2, p. 343–353, June 2011. ISSN 1524-9050. Citado na página 41.

MUSIL, J. et al. Synthesized essence: what game jams teach about prototyping of new software products. In: KRAMER, J. et al. (Ed.). *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, (ICSE) 2010, Cape Town, South Africa, 1-8 May 2010*. ACM, 2010. p. 183–186. ISBN 978-1-60558-719-6. Disponível em: <http://doi.acm.org/10.1145/1810295.1810325>. Citado 5 vezes nas páginas 36, 37, 38, 39 e 48.

O'KEEFFE, M. K.; CINNÉIDE, M. Ó. Search-based refactoring for software maintenance. *Journal of Systems and Software*, v. 81, n. 4, p. 502–516, 2008. Disponível em: <http://dx.doi.org/10.1016/j.jss.2007.06.003>. Citado 2 vezes nas páginas 28 e 29.

POSHYVANYK, D. et al. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, v. 14, n. 1, p. 5–32, fev. 2009. Citado na página 47.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. Eighth edition. New York, NY: McGraw-Hill, 2014. Citado 2 vezes nas páginas 25 e 48.

RABIN, S. *Introduction to Game Development*. [S.l.]: CharlesRiver Media, 2005. 978 p. (Game Development Series). Citado 2 vezes nas páginas 35 e 36.

SALEN, K.; ZIMMERMAN, E. *Rules of Play: Game Design Fundamentals*. [S.l.]: The MIT Press, 2004. Citado 2 vezes nas páginas 33 e 35.

SIMON, F.; STEINBRCKNER, F.; LEWERENTZ, C. Metrics based refactoring. set. 14 2001. Disponível em: <http://citeseer.ist.psu.edu/445787.html;http://kastanie.informatik.tu-cottbus.de/Papers/metrics_based_refactoring_full.pdf>. Citado na página 28.

WANG, J.; LEWIS, M.; GENNARI, J. A game engine based simulation of the nist urban search and rescue arenas. In: CHICK, S. et al. (Ed.). *Proceedings of the 2003 Winter Simulation Conference*. IEEE, 2003. p. 1039–1045. ISBN 978-1-60558-218-4. Disponível em: <http://doi.acm.org/10.1145/1496984.1497031>. Citado na página 41.

ZERBST, S.; DUVEL, O. *3D Game Engine Programming*. Boston, MA, USA: Thomson Course Technology, 2004. Citado 2 vezes nas páginas 40 e 43.