UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

INSTITUTO DE MATEMÁTICA

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FELIPE DE QUEIROZ BADEJO ALMEIDA

RACHINATIONS: Modelando a Economia Interna de Jogos

RIO DE JANEIRO

MAIO DE 2015

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

INSTITUTO DE MATEMÁTICA

CURSO DO BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FELIPE DE QUEIROZ BADEJO ALMEIDA

RACHINATIONS: Modelling the Internal Economy of Games

Trabalho de Conclusão de Curso apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Geraldo Bonorino Xexéo

RIO DE JANEIRO – RJ

MAIO DE 2015

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

INSTITUTO DE MATEMÁTICA

CURSO DO BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FELIPE DE QUEIROZ BADEJO ALMEIDA

RACHINATIONS: Modelando a Economia Interna de Jogos

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em __ de _____ de _____.

BANCA EXAMINADORA:

_____

GERALDO BONORINO XEXÉO, DSc

_____

GERALDO ZIMBRÃO DA SILVA, DSc

_____

FELLIPE RIBEIRO DUARTE, MSc

RIO DE JANEIRO – RJ

MAIO DE 2015

RESUMO

Um jogo tende a ter propriedades complexas que precisam ser modeladas e representadas de alguma forma caso se queira entender seu comportamento e identificar fatores que o possam influenciar. Usando o framework *Machinations* de autoria do Dr. J. Dormans como base, criamos o *Rachinations,* uma engine baseada em uma linguagem específica de domínios (DSL) para descrever e simular o funcionamento de jogos simples e também complexos. Com ela podemos definir precisamente a estrutura interna de jogos e descobrir padrões de comportamento que emergem após um certo tempo. O objetivo é criar um ambiente onde se possa validar hipóteses e testar estratégias relacionadas à mecânica interna de jogos, tudo isso sem precisar construir um protótipo ou mesmo escrever uma linha de código do jogo em si.

Palavras-chave: jogos, jogos complexos, sistemas complexos**,** dinâmica de sistemas.

ABSTRACT

Games are sometimes complex and display properties that need to be represented and modelled if one wants to understand their long-term behaviour and identify factors that influence them. Using Dr. J. Dormans' *Machinations* game modelling framework as reference we have created *Rachinations,* a simple DSL-based engine to define and simulate the execution of simple as well as complex games. It allows for simple defining and precise validation of a game's internal mechanics, in order to observe long-term patterns that emerge during the course of the game, providing a cheap and easy environment to test hypotheses and validate strategies without the need to construct a game prototype or write a single line of application code.

Keywords: games, game-playing, complex games, complex systems, system dynamics.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Uma das coisas que mais nos fascina e deslumbra sobre o universo em que vivemos é sua complexidade. Ela é responsável por toda a beleza do mundo natural que observamos na estrutura de um floco de neve, na estrutura do corpo humano e até na dinâmica dos corpos celestes.

Uma das áreas da ciência que se propõem a estudar o comportamento de sistemas que exibem certa complexidade, ou seja, de sistemas complexos, é a **dinâmica de sistemas** (do inglês, *system dynamics*). (SYSTEMDYNAMICS.ORG 1997)

Dá-se, aqui, o nome de sistemas complexos a sistemas que podem ser descritos e modelados via conceitos como *loops* retroalimentados (do inglês *feedback loops*) estoques e fluxos (do inglês *stocks and flows*), entre outros. Esses termos serão mais precisamente definidos em breve. (DORMANS 2011)

Central à análise do comportamento de sistemas complexos é a ideia de **emergência**, que é o nome que se dá ao surgimento de padrões de comportamento e crescimento bastante complicados em tais sistemas, mesmo aqueles definidos por regras bem simples. Esses padrões podem ser difíceis de prever e comumente apresentam comportamento vezes não linear.

A quantidade de fenômenos e processos que podem ser modelados em um sistema complexo é potencialmente infinita. As estruturas internas de jogos, como por exemplo jogos comuns de tabuleiros podem ser vistas como sistemas complexos onde o fluxo de recursos como pontos, unidades monetárias, e vidas se dá entre jogadores a partir de regras de formação em geral bem definidas.

A evolução destes recursos, assim como padrões recorrentes relacionados aos mesmos observados durante um jogo, sofrem efeitos de emergência e podem ser fatores que definem a *jogabilidade* de um jogo. A essa evolução dinâmica no estado de um jogo é comumente dado o nome de **mecânica do jogo** ou economia interna do jogo. (DORMANS 2011)

Assim como a humanidade deu um salto de progresso a partir do momento em que se começou a representar de forma permanente o conhecimento até então tácito ou perpetuado apenas através da tradição oral, também se espera que, para que a ciência do estudo de jogos, chamada de *ludologia*, cresça e se desenvolva é necessário, antes de tudo, criar formas, possivelmente variadas e complementares, de traduzir a estrutura interna de jogos em padrões formais que cruzem as fronteiras geográficas e temporais entre as pessoas que se dedicam a este trabalho. (DORMANS 2011)

O interesse no estudo acadêmico da mecânica de jogos é relativamente recente e, até bem pouco tempo, carecia-se de uma **linguagem de descrição** que capturasse a ótica necessária para que pessoas pudessem analisar, simular e comparar mecânicas diferentes de jogos. Isso poderia ajudar a introduzir um pouco de formalidade científica no processo, bem como entender quais características, do ponto de vista da mecânica, tornam um jogo divertido, estimulante e imprevisível. (DORMANS 2011)

Apesar de jogos existirem há muito tempo (HUIZINGA 1938), a revolução dos vídeo-games que começou na segunda metade do século XX deu novo fôlego ao mundo dos jogos e o interesse acadêmico nos mesmos vem crescendo desde então.

Talvez a interação entre programadores e líderes criativos envolvidos no desenvolvimento de vídeo-games (já que um vídeo-game é, antes de tudo, software) tenha tido influência no fato de algumas linguagens visuais de descrição de jogos terem surgido nos últimos tempos.

Alguns exemplos relevantes são os métodos criados por Raph Koster (2005), Stéphane Bura (2006), além de adaptações de linguagens visuais já existentes, como UML, Redes de Petri, máquinas de estado e também diagramas de *loops* causais (do inglês *Causal loop diagrams*). Além disso temos o *framework* Machinations, que surge em 2008 (DORMANS 2011).

O *framework* **Machinations**, desenvolvido pelo professor holandês Joris Dormans, se dedica a descrever jogos do ponto de vista dos recursos que se movimentam ao longo de um jogo e é bastante útil para se analisar e simular o comportamento dos mesmos. Através de diagramas com elementos como nós, conectores e recursos, ele representa os estoques e os fluxos que caracterizam a economia interna da maioria dos jogos baseados em regras.

O Machinations parece, em comparação com as outras abordagens citadas e após estudo da parte de quem lhes escreve, a forma mais elegante e útil para se representar fluxos, estoques e *loops* dentro de um jogo. Partindo-se do pressuposto, baseado em suposição do Dr. Dormans, que essa *mecânica* interna de um jogo é um dos principais fatores que determinam o quão bom ou divertido um jogo é (DORMANS 2011), é natural que nos debrucemos mais sobre o Machinations na tentativa de evoluir o entendimento sobre o tema.

Apesar de o framework Machinations ter se mostrado satisfatório para se modelar jogos de complexidade pequena e média, a evolução da complexidade e a impossibilidade de se construir abstrações (módulos, subdiagramas e outros) sobre os diagramas, além das ineficiências inerentes a linguagens visuais (por exemplo, o fato de não serem boas para programação em alta-escala) suscitaram rapidamente sugestões de como aproveitar a boa expressividade do Machinations em uma escala maior.

Sobre o uso de linguagens de programação visuais e a dificuldade em usá-las para projetos grandes, Bansal (BANSAL 2013) afirma:

> O uso do paradigma de programação visual para programação de uso geral em larga escala permaneceu adormecida por causa do seguinte: (1) a dificuldade de analisar, *parse*, uma linguagem em duas dimensões, (2) a dificuldade de compreensão humana de programas grandes em duas dimensões, e (3) dificuldade em apresentar grandes programas visuais para programadores humanos. (BANSAL 2013, tradução livre)

Na tentativa de criar uma nova abordagem para o *framework* Machinations que seja de mais fácil uso quando quisermos simular diagramas maiores e mais complexos,

vislumbrou-se a possibilidade de criar uma outra forma de representar algo que já sabemos que funciona bem, mas de uma forma mais escalável e formal.

Uma ideia clara foi criar uma forma de definir diagramas Machinations a partir de código em alguma linguagem de programação, através de uma linguagem específica de domínio (DSL, sigla em inglês *para Domain-specific language*), assim como uma *engine* que executasse estes diagramas de forma próxima àquela usada no aplicativo Machinations.

Decidimos, então, atacar o problema através de uma DSL interna baseada na linguagem de programação Ruby, pois esta tem uma sintaxe relativamente liberal, não requer o uso de marcadores sintáticos como parênteses e ponto e vírgula, suporta a definição de palavras reservadas com semântica própria e possui amplo suporte a metaprogramação.

O Rachinations (Machinations em Ruby) nasceu desta proposta. Ele possui dois componentes principais: uma DSL que permite que o usuário, mesmo com pouca experiência em programação defina um diagrama similar a um diagrama Machinations; e uma *engine*, responsável pela execução do diagrama, que possibilitará que o usuário execute o diagrama, faça experimentos, comprove hipóteses e simule mecânicas de jogos.

2 DINÂMICA DE SISTEMAS

A Dinâmica de Sistemas (do inglês, *System Dynamics*) é uma forma de definir, entender e modelar sistemas complexos para simulá-los e identificar quais fatores influenciam no seu comportamento ao longo do tempo. O início do estudo da Dinâmica de Sistemas é comumente atribuído ao professor Jay W. Forrester, do Massachusetts Institute of Technology (MIT), na década de 50. (SYSTEMDYNAMICS.ORG 1997)

> Tradicionalmente, a ciência explora fundamentalmente o muito pequeno e o muito grande, ambos os quais estão além da percepção do homem médio. A peculiaridade de sistemas complexos é que eles têm a ver com uma classe de fenômenos de fundamental importância, nos quais tanto o sistema quanto o observador se situam em escalas comparáveis de tempo e de espaço. (NICOLIS et al 2007, tradução livre)

O que torna a Dinâmica de Sistemas diferente de outras abordagens para estudo de sistemas complexos dentro da teoria de sistemas é principalmente o uso de *feedback loops* e de *stocks* e *flows*. Cabe citar um princípio, chamado **Princípio da Acumulação**, segundo o qual todo o comportamento dinâmico no mundo inteiro ocorre quando fluxos (flows) acumulam em estoques (stocks). (SYSTEMDYNAMICS.ORG 1997)

Esta abordagem pode ser aplicada a qualquer problema que possa ser modelado de forma adequada, com o objetivo de analisar os efeitos de decisões ou políticas (públicas ou privadas) específicas dentro de um contexto. Alguns exemplos são as áreas de estudos demográficos, estudos do meio-ambiente (estudos de sustentabilidade) ou simulações relativas à situação econômica de um país ou empresa.

2.1 REPRESENTAÇÃO

Há pelo menos duas formas bastante difundidas que são usadas para representar sistemas complexos para facilitar seu estudo do ponto de vista da dinâmica de sistemas: **diagramas de estoques e fluxos** e **diagramas de *loops* causais**. (SYSTEMDYNAMICS.ORG 1997)

2.1.1 Diagramas de estoques e fluxos

Diagramas de estoques e fluxos (do inglês, *stock and flow diagrams)* ajudam a estudar um sistema complexo na medida em que pode-se usá-los para identificar e descrever os pontos que representam fluxos e os pontos que representam estoques.

Pelo princípio da acumulação citado anteriormente, todo o comportamento dinâmico do mundo pode ser representado através destas duas entidades mas, como um modelo perde parte de sua utilidade caso se torne muito complexo e detalhado, na maior parte dos casos modelados cada estoque tem entre 4 e 6 fluxos de entrada ou de saída. (SYSTEMDYNAMICS.ORG 1997)

Na Figura 1 temos um exemplo simples de um diagrama de estoques e fluxos representando a dinâmica da obtenção e manutenção do conhecimento por uma pessoa;

retângulos representam estoques e arestas duplas representam fluxos.

"Nuvens" são também elementos frequentes neste tipo de diagramas e representam uma fonte ou sumidouro ideais – para simplificar o sistema e abstrair partes que não se deseja estudar no momento.

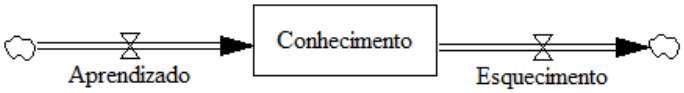

Figura 1 - O conhecimento é um estoque que cresce com o aprendizado e decresce com o esquecimento. Fonte: elaborada pelo autor

A Figura 2 representa um sistema um pouco mais complexo: uma população de galinhas é composta por indivíduos (no diagrama, "Galinhas") e por ovos, cujas dinâmicas influenciam um ao outro.

As arestas azuis (sintaxe não presente em todos os diagramas de estoques e fluxos) representam maneiras em que um elemento influencia o outro indiretamente: uma letra "R" representa uma influência positiva ("Reforço") e uma letra "B" representa uma influência negativa ("Balanço"). Estas influências são denominadas *loops* e são representadas com maior ênfase no tipo de diagrama discutido mais à frente.

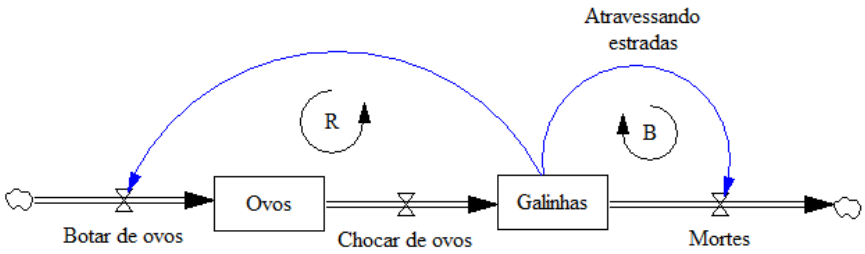

Figura 2 - Diagrama de estoques e fluxos representando uma população de galinhas que nascem, botam ovos e ocasionalmente morrem ao atravessar estradas. Uma maior quantidade de galinhas influencia o fluxo de postura de ovos e também influencia a taxa de mortes. Fonte: elaborada pelo autor

2.1.2 Diagramas de loops causais

Enquanto um diagrama de estoques e fluxos tende a focar mais na identificação e na representação de estoque e fluxos, um **diagrama de loop causal** (do inglês, *causal loop diagram*) foca mais nas influências indiretas que um elemento tem sobre outros.

Esta influência é chamada de loop retroalimentado (do inglês, *feedback loop*) e representam parte (ou a totalidade) do sistema em questão. Ele mostra como diferentes variáveis em um sistema estão inter-relacionadas.

As setas ligando uma variável a outra mostram como aquela influencia essa. Caso haja um sinal de menos (-) ao fim da seta, um aumento na variável de origem causa um decréscimo na variável de destino. Caso haja um sinal de mais (+), a influência é positiva.

Em geral, caso haja equilíbrio entre fluxos em um loop, este loop será chamado de **loop balanceado** (marcado com uma letra "B" no diagrama). Caso contrário, ou seja, se os fluxos relacionados a este loop se reforcem mutuamente (como é o caso do número de nascimentos e o total da população), este será um **loop reforçado** e será marcado com uma letra "R", como mostrado na Figura 3:

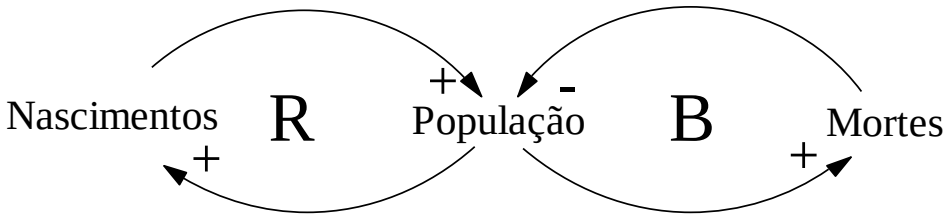

Figura 3 - Diagrama de loop causal relacionando nascimentos e mortes à população total (de uma espécie); um crescimento da população reforça a si mesmo através do relacionamento com a taxa nascimentos, mas também balanceia a si mesmo através do relacionamento com a taxa de mortes.

2.2 RELEVÂNCIA PARA O ESTUDO DE JOGOS

Apesar de não ser citada diretamente no principal livro que define o framework Machinations (DORMANS 2011), a dinâmica de sistemas usa um vocabulário bastante similar ao usado no framework e também é utilizada para estudar sistemas complexos, levando-nos a sugerir que, mesmo que indiretamente, ela pode ter servido como inspiração para a criação do framework.

Tendo isso em vista, é razoável que um estudioso de jogos mantenha-se minimamente ciente de novos desenvolvimentos na dinâmica de sistemas pois ela trata problemas análogos aos encontrados na área de ludologia e já existe há mais tempo, podendo ser fonte de *insights* e inspirações para o estudo de jogos.

3 O FRAMEWORK MACHINATIONS

Como já explicado anteriormente, o *framework* Machinations se propõe a modelar a localização e a transferência de recursos ao longo do tempo de duração de um jogo. Uma das intenções é tentar identificar padrões recorrentes na estrutura e no funcionamento de jogos diferentes, de modo a fornecer mais ferramentas para desenvolvedores de jogos e estudiosos do tema (DORMANS 2011).

O framework modela jogos usando principalmente os conceitos de **recursos, nós** e **conectores**.

3.1 EXEMPLO 1 – TETRIS

A Figura a seguir mostra um exemplo de diagrama Machinations. Ele representa uma modelagem possível da mecânica interna do jogo Tetris.

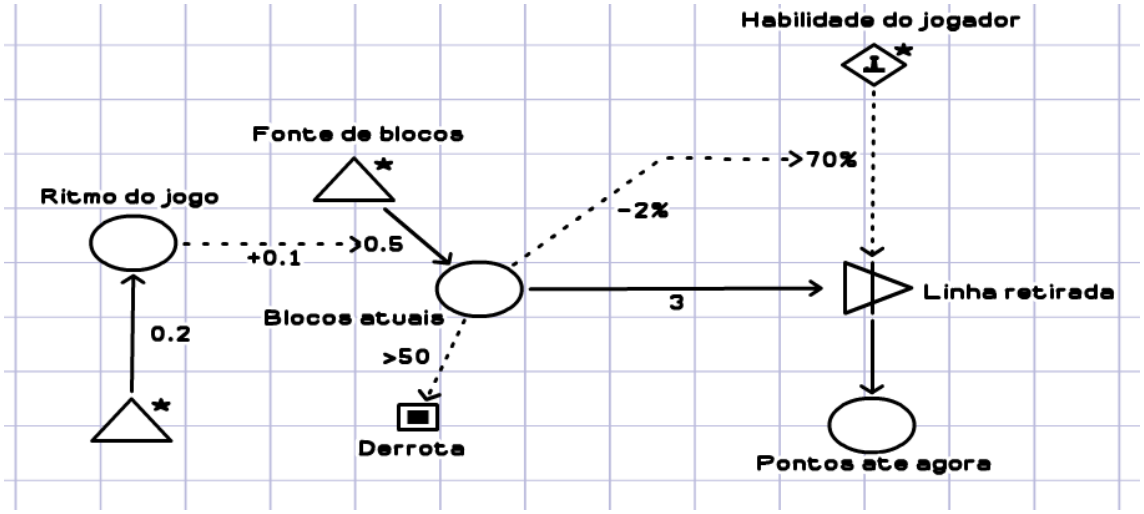

Figura 4 - Um diagrama Machinations representando o jogo Tetris. Fonte: elaborada pelo autor

No diagrama podemos ver que diferentes elementos influenciam na dinâmica do jogo; o "ritmo do jogo" representa a velocidade com que novos blocos são gerados a partir do topo da tela. Ele é representado por um depósito no diagrama e se conecta à aresta que une a fonte de blocos com o depósito que representa o número de blocos em jogo atualmente.

Pode-se ver que a variável que representa a habilidade do jogador, inicializada em 70% (em uma escala de 0% a 100%), governa a taxa com a qual blocos existentes são removidos do jogo, sendo convertidos em um ponto. O número de blocos em jogo afeta negativamente o nível de habilidade do jogador, como em um jogo real. Em outras palavras, faz com que erros sejam mais prováveis em tais condições.

Também pode-se perceber que neste diagrama há somente uma condição para término, com derrota. Isso corresponde ao fato do jogo Tetris não ter, efetivamente, um critério para vitória – o jogo continua enquanto o jogador conseguir se manter "vivo".

A Figura 5 mostra como o diagrama mostrado na figura anterior evolui, após algumas rodadas.

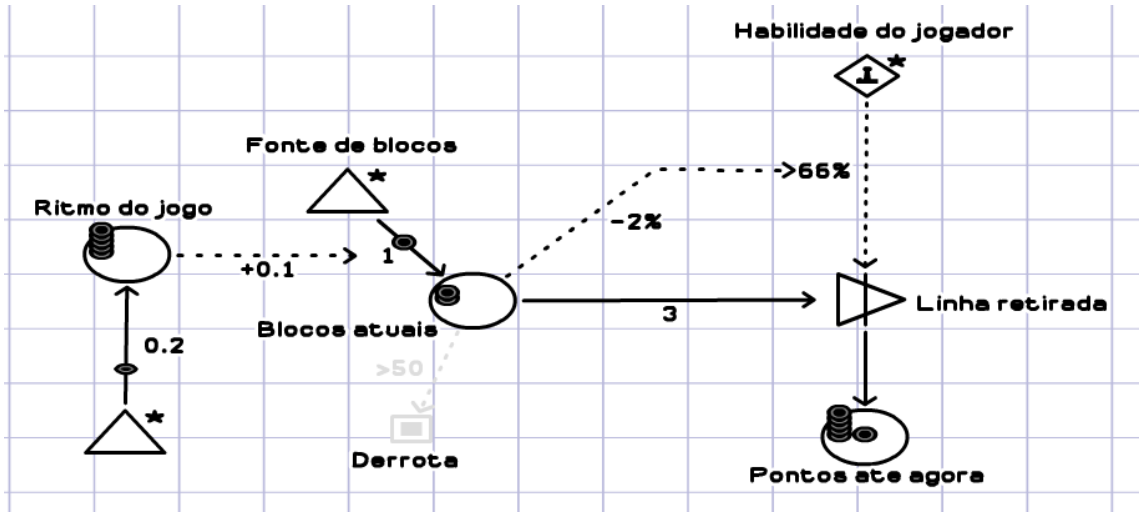

Figura 5 - O diagrama anterior, após algumas rodadas. Fonte: elaborada pelo autor

3.2 EXEMPLO 2 – SPACE INVADERS

A figura a seguir mostra uma das possíveis maneiras de se representar o jogo *Space Invaders*.

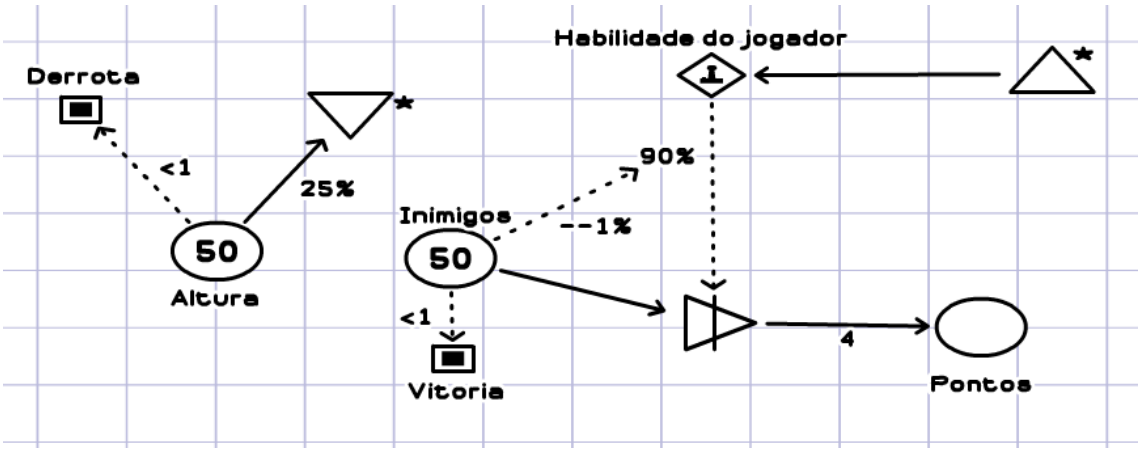

Figura 6 - Diagrama Machinations representando o jogo Space Invaders. Fonte: elaborada pelo autor

Representadas por um quadrado parcialmente preenchido estão as condições de término do jogo – derrota caso a altura dos inimigos chegue a zero (partindo de 50) e vitória caso o jogador consiga eliminar todos os inimigos (também 50) antes que eles o alcancem.

Como no jogo anterior, há elementos simulando a **habilidade do jogador** - que governa a taxa de eliminação dos inimigos, que são convertidos em pontos.

Note que o conjunto formado pelo nó representando a habilidade do jogador, o nó

representando a quantidade atual de inimigos em campo e o nó que faz a conversão de inimigos para pontos pode ser visto como um loop retroalimentado, do tipo balanceado. Isso ocorre porque um alto grau de habilidade do jogador fará com muitos inimigos sejam eliminados o que, por sua vez, tem o efeito de diminuir a habilidade do jogador. Isso é uma forma de simular o fato de que, quando há poucos inimigos na tela, é mais difícil para o jogador acertá-los.

A Figura a seguir mostra o estado do diagrama da figura anterior, após a execução de algumas rodadas.

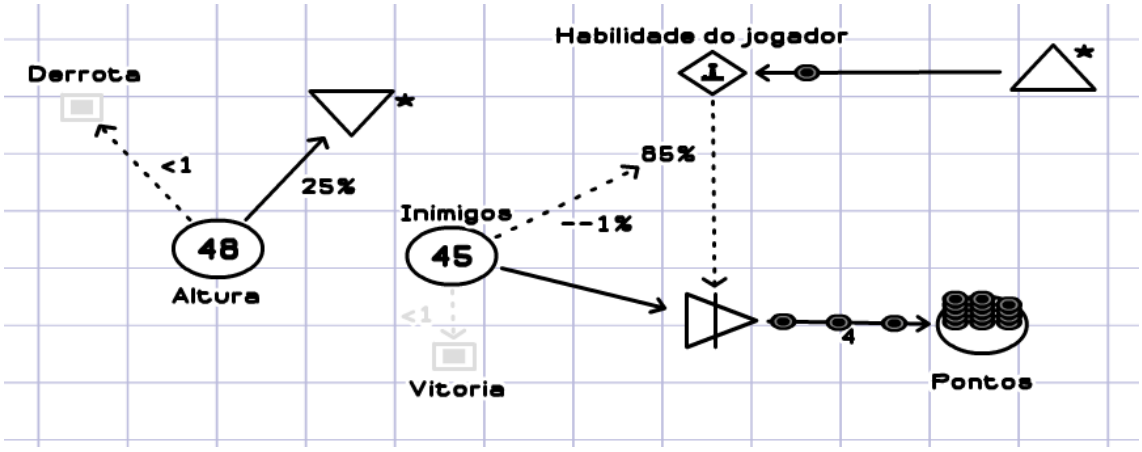

Figura 7 - O diagrama anterior, após alguns turnos. Fonte: elaborada pelo autor

3.3 RECURSOS

Recursos representam qualquer quantidade que varia durante um jogo e que pode ser usada para determinar a vitória de um jogador ou estimular a competição entre os mesmos.

Exemplos de coisas que podem ser modeladas como recursos são vidas (em um jogo de plataforma), moedas (em um jogo econômico), imóveis (em um jogo de tabuleiro como Banco Imobiliário) ou simplesmente pontos em um jogo esportivo.

As Figuras 8, 9 e 10 mostram três diferentes formas sob as quais recursos são mostrados no aplicativo Machinations: um recurso normal, recursos coloridos e recursos representados por um número.

Figura 8 - Um recurso dentro de um nó. Fonte: elaborada pelo autor

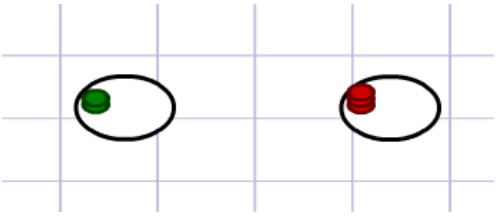

Figura 9 - Recursos podem ter diferentes cores. Fonte: elaborada pelo autor

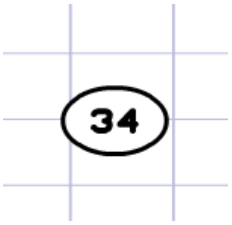

Figura 10 - Recursos são representados como números quando há muitos deles em um único nó. Fonte: elaborada pelo autor

Recursos não são muito úteis a menos que seja possível transportá-los para outros nós, de forma análoga a estoques em um diagrama de estoques e fluxos.

A figura a seguir mostra como é representado no aplicativo Machinations a passagem de um recurso de um nó para outro.

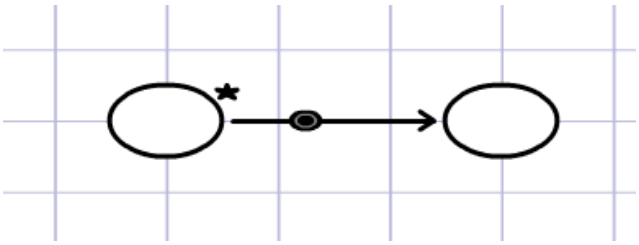

Figura 11 - Um recurso sendo passado de um nó a outro através de uma aresta. Fonte: elaborada pelo autor

3.4 NÓS

Nós são usados para representar a localização de um ou mais recursos em um dado instante de tempo (Depósito) ou para executar alguma ação específica quando um recurso lhes é fornecido (os outros tipos de nó).

Na figura a seguir pode-se ver os principais tipos de nó usados no framework Machinations, com os seus respectivos nomes, em inglês.

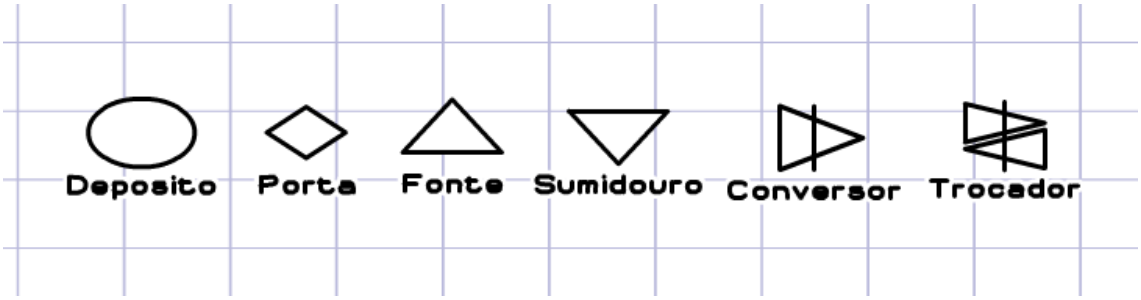

Figura 12 - Os principais nós do Machinations, com seus nomes em português. Fonte: elaborada pelo autor

A semântica de cada nó, ou seja, o significado que ele tem em um determinado jogo, depende obviamente da modelagem que se fez, mas em geral nós são usados para fins de contagem, - de pontos ou de qualquer outro valor relevante para o jogo - controle de fluxo, condições de parada, etc.

Em um jogo como War (*Risk*, na versão original em inglês), por exemplo, poderíamos usar nós (do tipo Depósito) para guardar recursos que representam o número de territórios que um jogador controla em um dado instante e também nós (do tipo Porta) para representar caminhos pelos quais um território passa de um jogador para outro. No caso do War, esta passagem se dá através de batalhas com dados.

Na Figura 13 o leitor pode ver uma modelagem do jogo War que mostra como vários loops governam o comportamento do jogo.

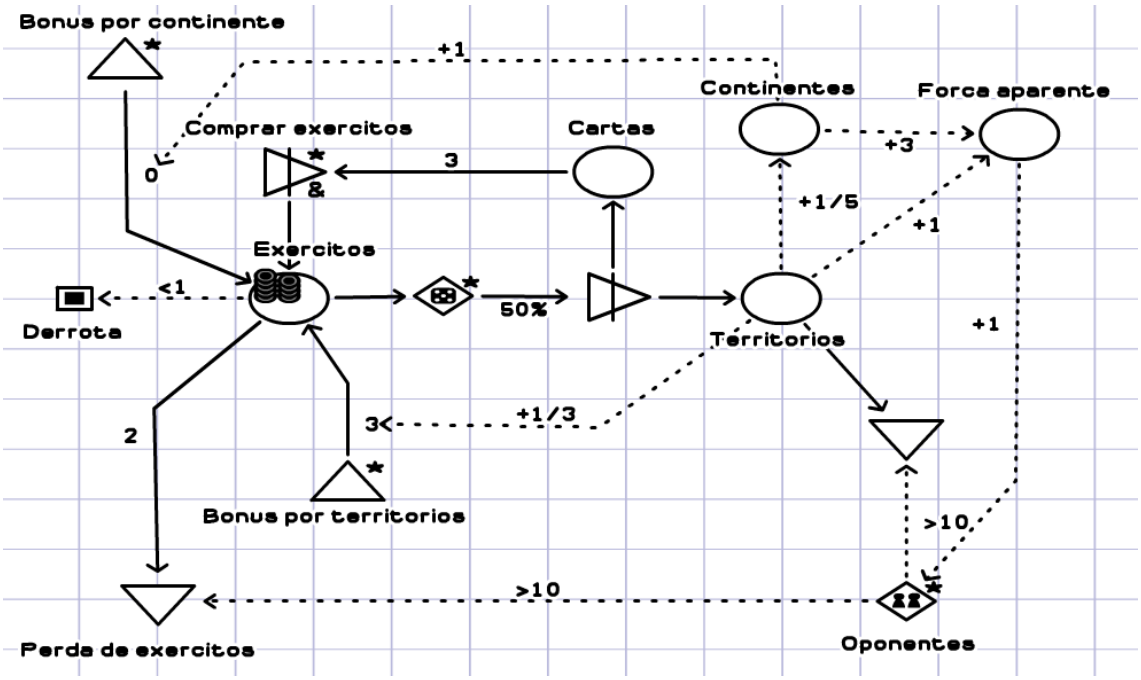

Figura 13 - Uma possível modelagem do jogo War (Risk, em inglês). Fonte: elaborada pelo autor

3.4.1 Principais tipos de nó

3.4.1.1 Depósitos (Pools)

São nós que possuem um número finito de recursos e podem enviá-los para outros nós e recebê-los de outros nós através de conectores. Nós do tipo depósito são usados para controlar qualquer tipo de quantidade que seja relevante ao jogo em questão.

As Figuras 14, 15 e 16 mostram três diferentes formas que um depósito pode apresentar.

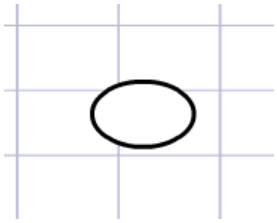

Figura 14 - Um Depósito vazio (sem recurso algum). Fonte: elaborada pelo autor

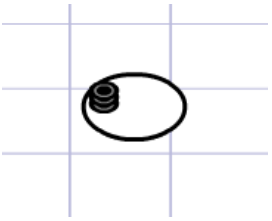

Figura 15 - Um Depósito com três recursos. Fonte: elaborada pelo autor

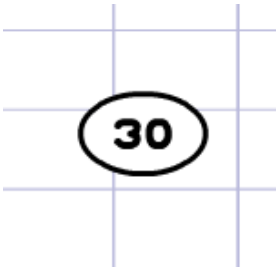

Figura 16 - Um Depósito com 30 recursos. Fonte: elaborada pelo autor

3.4.1.2 Fontes (Sources)

Fontes podem ser vistas como depósitos ilimitados. Fontes possuem infinitos recursos e podem ser usadas para simular ações constantes que ficam ativas enquanto o jogo estiver em andamento. Um exemplo simples é mostrado na figura a seguir.

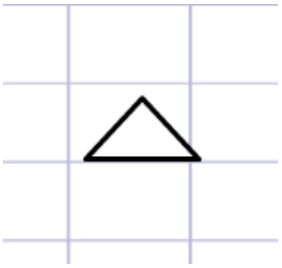

Figura 17 - Uma Fonte pode gerar infinitos recursos. Fonte: elaborada pelo autor

3.4.1.3 Sumidouros (Sinks)

Sumidouros são nós usados para servir de destino para recursos que não se deseje mais controlar. Pode-se ver na figura a seguir que um sumidouro é, do ponto de vista semântico assim como do ponto de vista visual, o oposto de uma fonte.

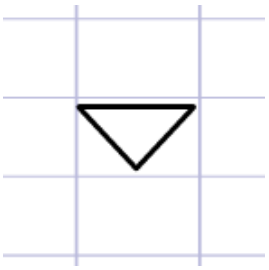

Figura 18 - Um Sumidouro pode receber infinitos recursos. Fonte: elaborada pelo autor

3.4.1.4 Conversores (Converters)

Conversores fazem a conversão de recursos em outros. Conversores têm, em geral, uma aresta entrando e uma aresta saindo (do tipo modificador de nó provavelmente) e podem ser usados, por exemplo, para significar que um certo número de recursos de um tipo pode ser transformado em outro.

Um exemplo pode ser visto no jogo Banco Imobiliário, onde o ato de se comprar uma casa poderia ser modelado através de um conversor, pois a compra de uma casa transforma um tipo de recurso (dinheiro) em outro tipo (propriedades).

A Figura 19 mostra um conversor na sua forma mais simples.

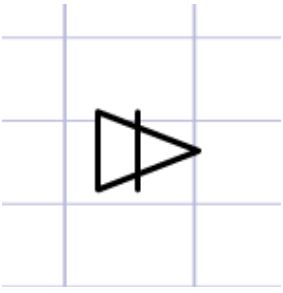

Figura 19 - Um Conversor recebe recursos de um lado e envia recursos do outro. Fonte: elaborada pelo autor

3.4.1.5 Trocadores (Traders)

A diferença entre um Trocador e um Conversor é que, no caso do Conversor, pode haver geração ou diminuição de recursos, ou seja, o total de recursos em jogo pode ser alterado – se, por exemplo, os rótulos das arestas que entram e saem do conversor forem diferentes. No caso do Trocador, a quantidade total de recursos é mantida constante.

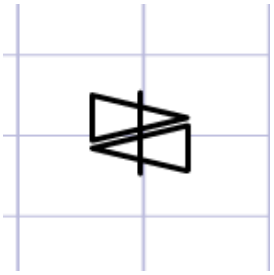

Figura 20 - Um Trocador faz trocas (possivelmente entre jogadores diferentes). Fonte: elaborada pelo autor

Segundo o criador do framework Machinations (DORMANS 2011), trocadores podem, por exemplo, ser utilizados para simular trocas de recursos entre dois jogadores. As duas estruturas na figura a seguir são, segundo o mesmo, equivalentes:

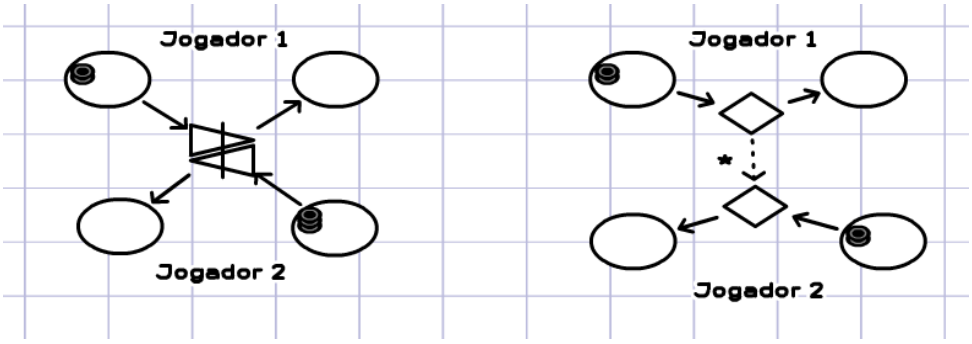

Figura 21 - Duas estruturas equivalentes, mostrando a troca de recursos entre dois jogadores. Fonte: elaborada pelo autor

3.4.1.6 Portas (Gates)

Portas são usadas para limitar, de alguma forma, a passagem de recursos no diagrama. Este tipo de nó não retém recursos (como um depósito faria) mas redistribui recursos que chegam imediatamente.

O primeiro tipo de porta é o mais simples e pode ser visto na Figura 22: representada por um losango vazio. Esta porta é chamada de porta determinística pois sempre distribui recursos na mesma ordem.

Ou seja, se uma porta determinística está conectada a dois nós através de conectores com o mesmo peso, os recursos irão, sempre, alternadamente para um nó e para o outro, para um e para o outro.

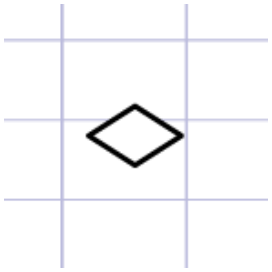

Figura 22 - Uma Porta determinística. Fonte: elaborada pelo autor

Os outros tipos de porta se diferenciam visualmente da primeira porque têm pequenos símbolos dentro e também por um fator muito importante: as outras portas são probabilísticas em vez de determinísticas.

Isto significa que, por exemplo, se uma destas portas estiver ligada a três nós (através de conectores), cada recurso será distribuído de forma aleatória a cada vez que a porta for gatilhada. Portanto, os diferentes símbolos mostrados nas figuras a seguir são usados apenas para ajudar o eventual leitor do diagrama entender o que ele representa para a modelagem em questão.

As Figuras 23, 24, 25 e 26 mostram as quatro formas que uma porta probabilística pode assumir.

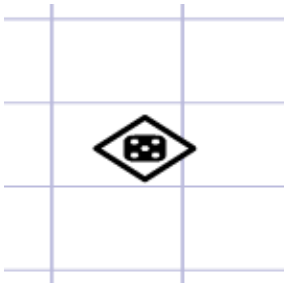

Figura 23 - Uma Porta probabilística pode ser usada para modelar os mais variados tipos de efeitos em um jogo. Fonte: elaborada pelo autor

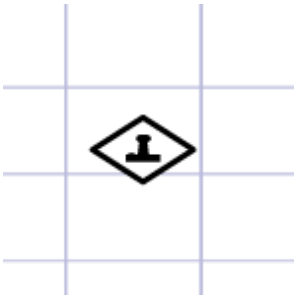

Figura 24 - Uma porta representando a habilidade de um jogador. Vários jogos têm aspectos que variam de acordo com a habilidade, ou *skill*, de um jogador. Fonte: elaborada pelo autor

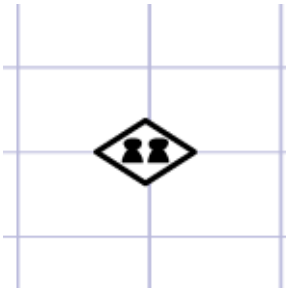

Figura 25 - Uma Porta representando interações sociais entre jogadores. Jogos com mais de um jogador comumente mostram este tipo de porta. Fonte: elaborada pelo autor

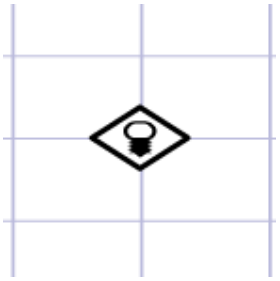

Figura 26 - Uma Porta representando a abstração de uma estratégia. Fonte: elaborada pelo autor

Na Figura 27 pode-se ver como portas podem ser (e frequentemente são) usadas para representar aspectos de um jogo que sejam variáveis ou não determinísticos.

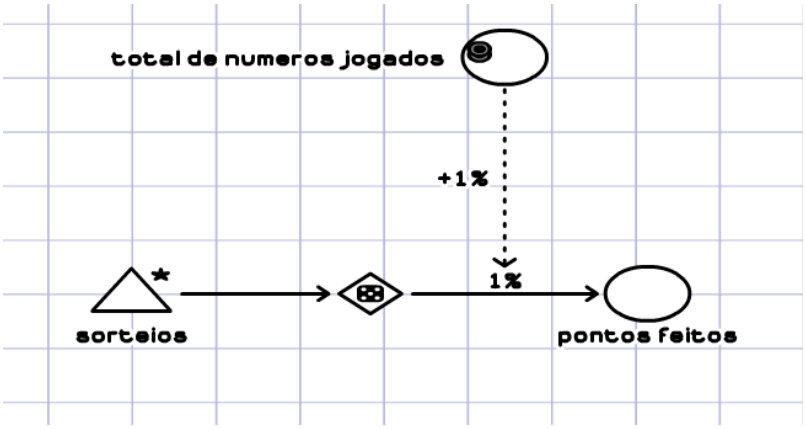

Figura 27 - Um exemplo do uso de Portas para representar aspectos de um jogo: considerando sorteios, quanto mais números um jogador jogar, mais chances ele tem de fazer pontos em cada sorteio. Fonte: elaborada pelo autor

Já na Figura 28 podemos ver como um jogo onde a habilidade de um jogador pode influenciar no comportamento do jogo.

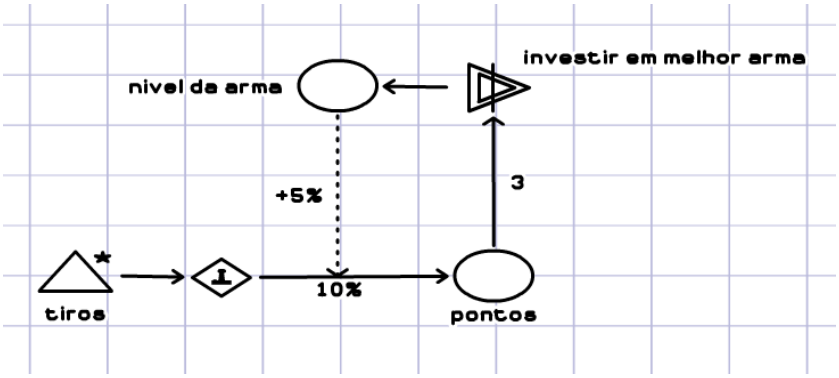

Figura 28 – Em um outro exemplo, neste jogo de tiro ao alvo, a Porta representa a chance de um tiro ser convertido em um ponto. Quanto maior o nível da arma, maior a chance de fazer pontos. Fonte: elaborada pelo autor

3.5 CONECTORES

Conectores são usados para conectar diferentes entidades dentro de um diagrama Machinations.

O uso mais comum é o de *modificador de* nó, que representa uma passagem de recursos entre um nó e outro. Em um uso mais avançado, eles podem também servir como *modificadores de rótulo*; neste caso, o modificador de rótulo é usado para mudar o valor de um outro conector (modificador de nó). Conectores são ainda usados na função de *gatilhos* e *ativadores*.

Vale salientar que apenas os conectores que são representados por uma aresta sólida nos diagramas (ou seja, somente os modificadores de nó) causam passagem de recursos entre um nó e outro. Todos os outros tipos de conectores, representados por arestas pontilhadas nos diagramas, apresentam uma *conexão virtual*, ou seja, onde um elemento do diagrama influencia outro mas não há efetiva passagem de recursos. Esta conexão pode ser vista como uma passagem de *sinais* em vez de recursos.

3.5.1 Principais tipos de conectores

3.5.1.1 Modificadores de nó (Node modifiers)

São representados por uma aresta sólida ligando um nó a outro. Modificadores de nó são um dos elementos mais comuns em um diagrama Machinations.

Modificadores de nó são os únicos conectores que permitem a passagem de recursos de um nó para outro.

Exemplos de casos de uso de modificadores de nó podem ser vistos nas Figuras 29, 30 e 31.

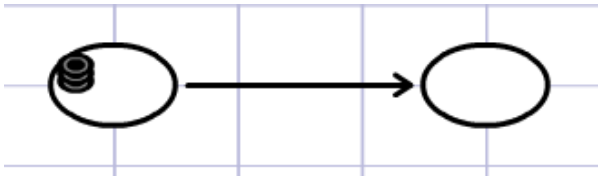

Figura 29 - Modificadores de nó permitem a passagem de recursos entre dois nós. Fonte: elaborada pelo autor

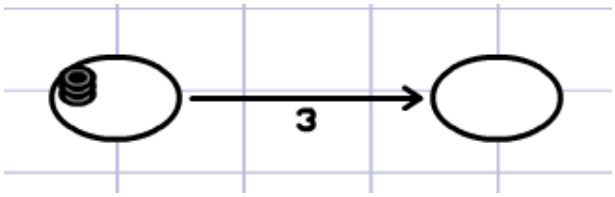

Figura 30 - Um conector que move 3 recursos por turno. Fonte: elaborada pelo autor

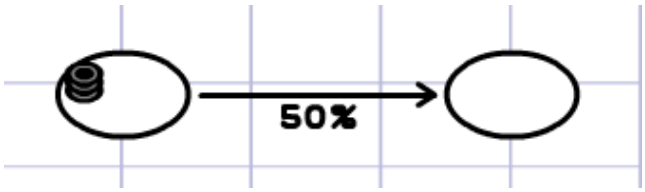

Figura 31 - A cada turno, este conector tem 50% de chance de mover 1 recurso. Fonte: elaborada pelo autor

3.5.1.2 Modificadores de rótulo (Label modifiers)

São representados por uma aresta pontilhada ligando um nó a um conector (do tipo modificador de nó).

No diagrama da figura a seguir, o modificador de rótulo mostrado faz com que o rótulo do modificador de nó (aresta sólida) seja incrementado de uma unidade para cada recurso no depósito à esquerda.

Neste exemplo, caso o depósito receba 5 recursos, o rótulo do modificador de nó valerá 6 (5+1) e causará, portanto, a passagem de 6 recursos em cada turno saindo da fonte e indo para o sumidouro.

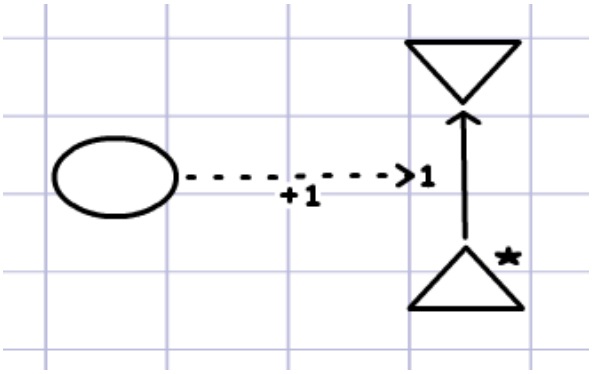

Figura 32 - Modificadores de rótulos alteram o valor de rótulos de conectores. Fonte: elaborada pelo autor

3.5.1.3 Gatilhos (Triggers)

São também representados por uma aresta pontilhada ligando um nó a outro nó. Um conector do tipo gatilho é sempre marcado por um asterisco (*).

Gatilhos fazem com que um nó seja disparado, ou *gatilhado*.

Na figura a seguir, por exemplo, toda vez que o depósito à esquerda for acionado (ou seja, quando receber ou enviar algum recurso) ele fará com que a fonte apontada pelo gatilho seja gatilhada. Isso fará com que um recurso saia da fonte e seja transportado para o sumidouro.

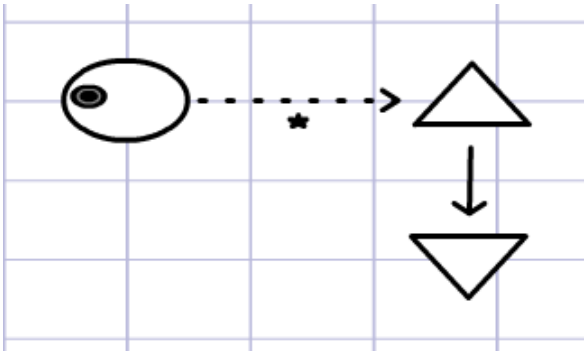

Figura 33 - Gatilhos fazem com que um nó seja 'gatilhado''. Fonte: elaborada pelo autor

3.5.1.4 Ativadores (Activators)

É representado por uma aresta pontilhada ligando um nó a outro nó. Diferentemente dos gatilhos, conectores do tipo ativador têm uma expressão associada. Somente quando esta expressão for satisfeita o nó apontado pelo ativador é ativado. Até que a expressão seja satisfeita o nó apontado fica desativado, ou seja, é como se não existisse no diagrama.

No exemplo da figura a seguir, somente quando o depósito tiver mais do que dois recursos a fonte será ativada. Note que pode ser que isso nunca chegue a acontecer.

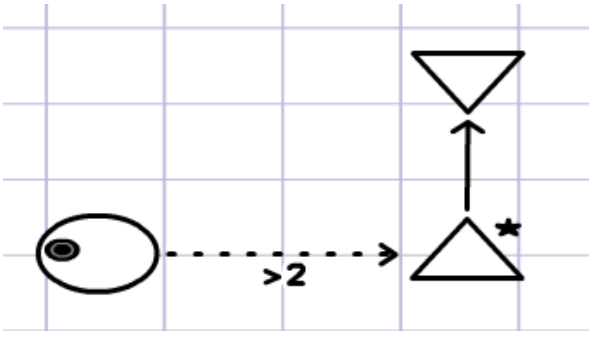

Figura 34 - Um Ativador com a condição ainda não satisfeita. Fonte: elaborada pelo autor

3.6 REGIME DE ATIVAÇÃO DOS NÓS

Os nós também se diferenciam de acordo com a sua ativação. Denominamos aqui *gatilhamento* (tradução livre do inglês *trigger*) como o momento em que um nó atua; do ponto de vista de uma Fonte, por exemplo, um gatilhamento causa um envio de recursos através de quaisquer conectores saindo de si; do ponto de vista de um Sumidouro, por outro lado, um gatilhamento fará com que arestas apontando para ele puxem recurso de nós a que estejam conectadas.

Nós podem ser criados com os regimes de ativação *Automático*, *Passivo*, *Inicial* ou *Interativo*, como descrito a seguir.

Note que qualquer nó será gatilhado se for alvo de um conector do tipo gatilho (representado por uma aresta pontilhada com um asterisco), não importando seu regime de ativação, podendo inclusive ser gatilhado mais de uma vez por turno.

3.6.1 Automático (Automatic)

Um nó no regime de ativação automático é gatilhado no início de cada turno. Nós neste regime são sinalizados com um asterisco, como mostrado na figura a seguir:

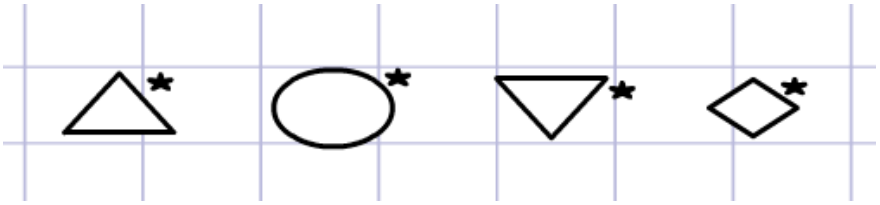

Figura 35 - Nós com ativação automática são sinalizados com um asterisco ('*'). Fonte: elaborada pelo autor

3.6.2 Passivo (Passive)

No modo de ativação passivo, o nó são executa ações independentemente. Um depósito passivo, por exemplo, só participará de interações iniciadas por outros nós (a ele conectados) ou se for explicitamente gatilhado através de um conector do tipo gatilho (representado por uma aresta pontilhada com um asterisco).

Nós passivos não têm nenhum detalhe visual que os diferencie; na verdade, a *falta* de detalhes é o que caracteriza um nó passivo, como se pode ver na Figura 36.

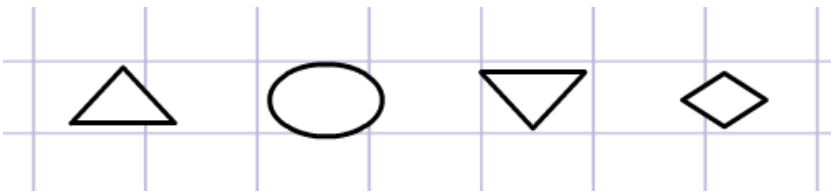

Figura 36 - Nós passivos não possuem detalhes extras. Fonte: elaborada pelo autor

3.6.3 Inicial (Starting action)

Nós com regime de ativação inicial são gatilhados somente uma vez, no primeiro turno, e depois se comportam como nós passivos até o fim da execução do diagrama.

Nós com este tipo de ativação podem ser usados para iniciar um processo ou simular o início de um jogo; eles são identificados por uma pequena letra ‘s’, como visto na figura a seguir.

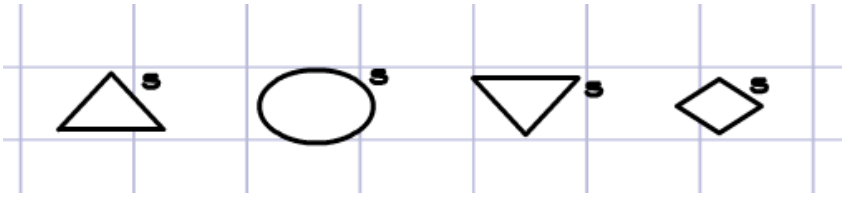

Figura 37 - Uma pequena letra 's' indica que um nó tem ativação inicial. Fonte: elaborada pelo autor

3.6.4 Interativo (Interactive)

O regime interativo (só relevante para aplicativos interativos), define que o nó será ativado quando um usuário clicar em cima dele. Nós com regime de ativação interativo são sinalizados por um segundo camada de sua borda. Eles estão representados na figura a seguir.

Vale lembrar que nós interativos não existem, a princípio, no Rachinations pois o mesmo se trata de código tipo texto e não suporta interação. A figura a seguir mostra como se parecem nós neste regime.

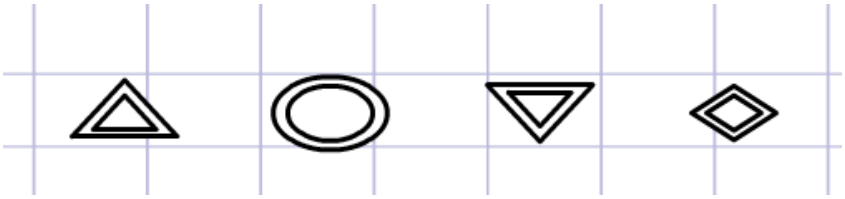

Figura 38 - Nós interativos têm borda dobrada e o usuário pode clicar nos mesmos (no aplicativo Machinations). Fonte: elaborada pelo autor

3.7 PASSAGEM DE RECURSOS

Uma outra forma de especializar o comportamento de certos nós é através da forma como a passagem (envio ou recebimento) de recursos se dá pelo mesmo. Este atributo será denominado simplesmente *modo* neste documento, para evitar ambiguidades e garantir a consistência do documento.

O modo de um nó governa o seu comportamento quando há mais de um conector saindo de um nó ou entrando no mesmo.

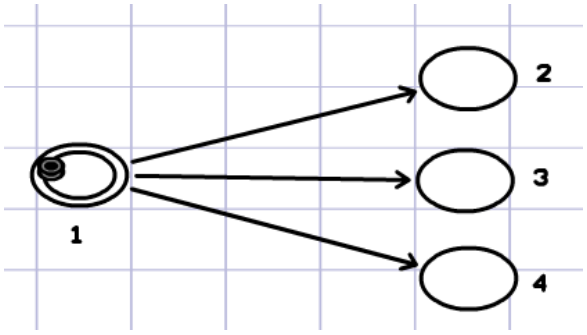

Figura 39 - O que deve acontecer quando o depósito 1 for gatilhado: recursos devem ser passados mesmo que nem todos os outros nós sejam atendidos ou nada deve acontecer até que todos os conectores possam ser satisfeitos?

3.7.1 Empurrar qualquer (push any)

Este é o modo padrão de um nó que empurra recursos, isto é, um nó que tem conectores saindo de si e o ligando a outros nós.

Neste modo, quando houver recursos insuficientes para satisfazer todos os conectores, há deslocamento de recursos para alguns nós e para outros não. A decisão sobre quais conectores favorecer parece ser feita de acordo com a ordem de adição dos mesmos. Conectores adicionados depois têm precedência sobre os demais.

As Figuras 40 e 41 mostram, respectivamente, os estados inicial e final de um pequeno diagrama com um nó neste modo.

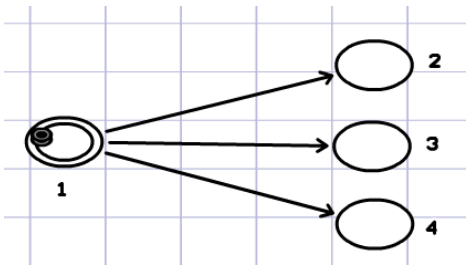

Figura 40 - Estado inicial: modo empurrar qualquer

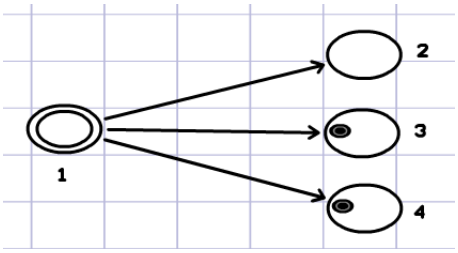

Figura 41 - Estado final: recursos foram movimentados

3.7.2 Empurrar todos (push all)

No caso em que um nó possui mais de um conector apontando para outros nós e está no modo *empurrar todos*, a passagem de recursos só é feita se todos os conectores puderem ser satisfeitos. Caso não seja possível satisfazer todos os conectores, nenhum recurso é movimentado.

Nós no modo empurrar todos são sinalizados com um pequeno símbolo “&”, como pode-se ver nas duas figuras a seguir, que mostram, também, os estados inicial e final de uma diagrama.

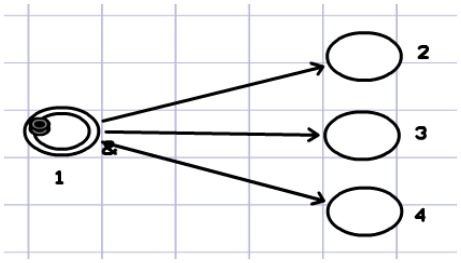

Figura 42 - Estado inicial: modo empurrar todos

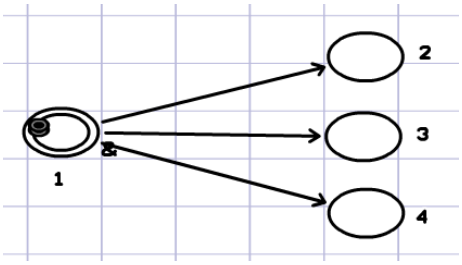

Figura 43 - Estado final: recursos não foram movimentados

3.7.3 Puxar qualquer (pull any)

Analogamente ao caso em que o nó empurra recursos para outros nós, há também modos que governam os casos em que um nó tem vários conectores apontando para si.

Como pode-se ver nas duas figuras a seguir, o modo **puxar qualquer** tenta puxar recursos mesmo que nem todas os conectores sejam satisfeitos:

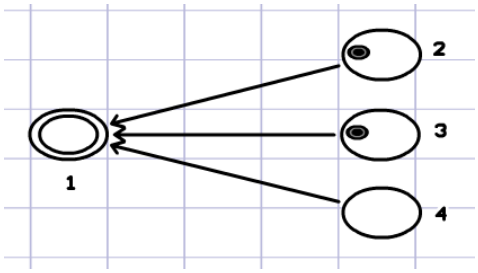

Figura 44 - Estado inicial: modo puxar qualquer

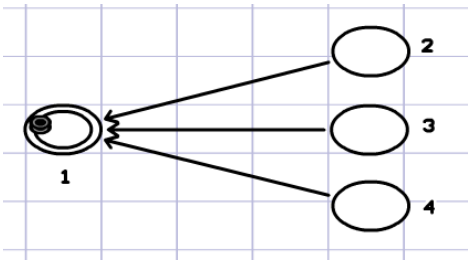

Figura 45 - Estado final: houve passagem de recursos

3.7.4 Puxar todos (pull all)

Quando um nó está no modo *puxar todos* e possui mais de um conector apontando para si, ele só puxa recursos dos nós conectados se todos os conectores puderem ser satisfeitos.

Nós neste modo também são representados por um símbolo “&”, como se pode ver nas duas figuras a seguir.

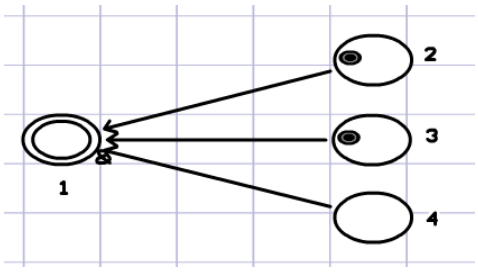

Figura 46 - Estado inicial: modo puxar todos

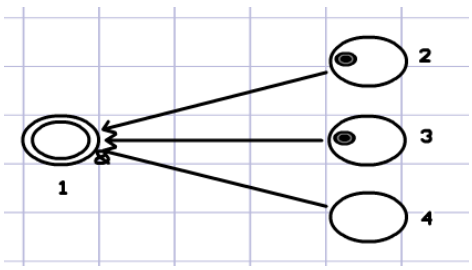

Figura 47 - Estado final: não houve passagem de recursos

4 A LINGUAGEM DE PROGRAMAÇÃO RUBY

A linguagem de programação Ruby foi usada para o desenvolvimento do sistema. Ruby tem amplo uso – ficou, de acordo com (BARD 2014), entre as três linguagens com mais repositórios criados no site de hospedagem de projetos open-source Github nos últimos três anos.

Esta linguagem foi escolhida pois (além do autor já ter alguma experiência com a mesma e conhecer alguns e seus pontos fortes) combina algumas características que facilitam o desenvolvimento de uma DSL interna, como sintaxe liberal (sem ponto e vírgula, com parênteses opcionais), programação funcional, tipagem dinâmica e extensas capacidades de metaprogramação.

4.1 EXEMPLOS

A seguir são mostrados alguns exemplos de uso da linguagem Ruby, para exemplificar algumas de suas características que a diferenciam de outras linguagens e, em particular, aquelas que a tornam boa para ser usada como DSL e que foram, portanto, usadas neste projeto.

4.1.1 Definição de métodos

```
# usa-se def/end para definir métodos
def write_to_stdout(str)
  puts str
end
#as duas linhas a seguir são equivalentes:
write_to_stdout("foo") # imprime "foo"
write_to_stdout "foo" # imprime "foo"
```

4.1.2 Blocos

Blocos são cidadãos de primeira-classe e um método pode passar o controle para um bloco através da cláusula `yield`. Blocos são delimitados por caracteres `'{'` e `'}'` ou por cláusulas `do` e `end`.

```
# um método que imprime um número se o bloco associado
# retornar true:
def print_if(number)
  if yield number
    print number
  end
end
```

```
# EXEMPLOS DE USO:
# não imprime nada pois 4 não é igual a 5
print_if(4){ |el| el == 5 }

# imprime 4 pois 4 é par
print_if(4){ |el| el % 2 == 0 }

# imprime 2 pois o bloco é sempre verdadeiro
print_if(2){ true }
```

4.1.3 Coleções e operações baseadas em programação funcional

```
array1 = ["foo","bar","baz"]
# map cria uma nova coleção a partir de uma antiga
# retorna ["Foo","Bar","Baz"]
array1_mod = array1.map{|str| str.capitalize }

array2 = [1,2,3,4,5,6]
# select (também conhecido como filter) filtra elementos
# que validam uma expressão
# retorna [2,4,6]
array2_mod = array2.select{|el| el % 2 == 0 }

# arrays podem conter elementos heterogêneos
array3 = [1,"foo",4.7]
```

4.1.4 Tudo é um objeto – até mesmo tipos primitivos

```
# chamando um método em um inteiro
resultado = 10.div 5 # o resultado é 2
# vetores são classes
arr = Array.new(4) # retorna [4]
```

4.1.5 Mixins

Mixins são conjuntos de métodos que podem ser associados a classes e/ou objetos. Eles podem ser usados para reusar código através de composição em vez de por herança. A frase "favoreça a composição à herança" (do inglês "*favor 'object composition' over 'class inheritance'*") é uma das principais recomendações contidas no livro "Design Patterns" (GAMMA et al 1994).

```
# módulos servem para várias coisas, como namespaces ou
# para definir mixins
module MeusMetodos
  def meu_metodo
    print("módulo MeusMetodos!")
  end
end

class MinhaClasse
  include MeusMetodos
end


objeto = MinhaClasse.new
# imprime "módulo MeusMetodos"
objeto.meu_metodo


# podemos também incluir este mixin em classes já prontas,
# até mesmo em um Array (do núcleo de Ruby)
class Array
  include MeusMetodos
end

arr = Array.new
# imprime "módulo MeusMetodos"
arr.meu_metodo
```

4.1.6 Metaprogramação

Um metaprograma é um programa que manipula outros programas (ou a si mesmo) como se fossem dados. O principal exemplo de metaprograma é um compilador, pois ele opera sobre código-fonte para gerar, em geral, um arquivo binário. (adaptado de C2.COM)

Deste ponto de vista, a alteração, em tempo de execução, do funcionamento de elementos (por exemplo, classes) é um exemplo de metaprogramação:

```ruby
# podemos "reabrir" uma classe já existente e modificá-la
class Array
  def dizer_oi
    puts "Oi!"
  end
end


arr = [1,2,3,4]
arr.dizer_oi # imprime "Oi!"
```

Outro exemplo de metaprogramação também possível em Ruby é a introspecção e modificação de elementos da linguagem (como métodos e classes), como ilustrado nos seguintes trechos de código:

```ruby
# listar os métodos de uma instância de Array e imprimir
# os que casam com a expressão regular /find/
arr = Array.new
arr.methods.grep(/find/).each{ |m| puts m }
# imprime "find_index", "find" e "find_all"


# podemos adicionar um método à instância arr
def arr.meu_metodo
  puts "meu método"
end

arr.meu_metodo # imprime "meu método"

# mas este método não está disponível para outras
# instâncias:
Array.new.meu_metodo # undefined method `meu_metodo'


# que tal modificar um método já existente em uma classe?
meu_array = [ "a", "b", "c", "d" ]
meu_array.size # retorna 4

Array.class_eval do
  def size
    5 # use com responsabilidade
  end
end
```

```
meu_array.size # retorna 5
```

4.2 DSLS BASEADAS EM RUBY

A seguir mostramos alguns exemplos de sistemas baseados em DSLs que foram exitosamente implementados em Ruby.

4.2.1 Sinatra

Sinatra é um *microframework* para desenvolvimento de aplicativos web feito em Ruby. Apesar de muitas outras opções estarem disponíveis, é possível definir um aplicativo inteiro ou um *web service* baseado em operações REST (Representational State Transfer) usando-se somente uma DSL baseada em Ruby. (SINATRARB.COM 2014)

Abaixo podemos ver um exemplo (conceitual, pois a implementação das operações na entidade `Usuario` não estão incluídas) de um conjunto de web services REST baseado em Sinatra, que disponibiliza operações de criação, visualização, edição e remoção de um usuário de um sistema:

```
get '/users/:id' do
  # mostra o usuário com o id dado
  Usuario.find(params[:id])
end


post '/users' do
  # cria um usuário com o id dado e o nome dado
  Usuario.new(params[:id],params[:name])
end


patch '/users' do
  # edita um usuário com o id dado e o nome dado
  Usuario.find(params[:id]).name = params[:name]
end


delete '/users' do
  # remove um usuário com o id dado
  Usuario.find(params[:id]).delete!
end
```

4.2.2 Chef

Chef é um sistema (CHEF 2014) de gerência de configuração (do inglês,

configuration management) que ajuda organizações e empresas a organizar todo o software em uma máquina ou conjunto de máquinas. Um dos principais casos de uso de um sistema deste tipo é o de definir a exata configuração de um servidor – instalação de pacotes, acionamento de serviços, configurações do sistema operacional, criação de arquivos, etc.

Esta configuração pode, então, ser compartilhada com outras máquinas e aplicada automaticamente. Isto diminui o custo de manutenção (instalação, atualização e configuração de software em cada máquina) de máquinas, a chance de erros e promove a homogeneização de um conjunto de máquinas.

A seguir temos um exemplo baseado nos códigos encontrados em (LEARNCHEF 2014) que mostra como podemos configurar um servidor Linux com o servidor web Apache usando Chef. (Comentários adicionados pelo autor)

```
# o pacote httpd será instalado com o gerenciador de
# pacotes existente no servidor
package 'httpd' do
  action :install
end


# iniciar o serviço httpd (instalado pelo passo anterior)
service 'httpd' do
  action [ :enable, :start ]
end


# criar um arquivo html sob /var/www/html/
# mostrando a string "Olá Mundo!" em seu corpo
file '/var/www/html/index.html' do
  content "<html><body>Olá Mundo!</body></html>"
  mode '0644'
  action :create
end
```

5 O SISTEMA RACHINATIONS

Como já mencionado anteriormente, um dos objetivos deste trabalho é definir e apresentar o sistema Rachinations, que se dispõe a oferecer um ambiente baseado em código para criação e execução de diagramas Machinations. A estrutura do sistema pode ser melhor compreendida se for dividida entre a) as classes necessárias para a representação do funcionamento do sistema ou, em outras palavras, o domínio ou regras de negócio e b) o código relacionado à linguagem de definição de diagramas.

O domínio é composto de uma classe principal, `Diagram`, que representa a totalidade de um diagrama e usa todas as classes que representam as partes que o compõem, a saber, nós, conectores, recursos, etc.

A outra parte do sistema, responsável por definir o funcionamento da linguagem específica de domínio (em inglês, DSL), é composta basicamente pelo módulo `DSL`.

O sistema foi desenvolvido com o apoio da ferramenta de versionamento GIT e está hospedado na comunidade Github. Ele pode ser acessado através do seguinte endereço: http://github.com/queirozfcom/rachinations.

5.1 PROJETO DO SISTEMA

Uma ferramenta como o Rachinations é, em última instância, um projeto de software (não-trivial) e, como tal, pode ser conduzido de várias formas diferentes. A literatura de engenharia de software das últimas décadas está cheia de exemplos de projetos que não deram certo bem como de pequenas práticas que, juntamente, diminuem o risco de um projeto de software falhar por perda de controle sobre a complexidade bem como por outras razões.

Nós tentamos nos valer da experiência coletada em tais obras e citamos a seguir algumas das mais importantes práticas que nortearam o desenvolvimento do Rachinations e sem dúvida contribuíram para que o mesmo pudesse ser completado a contento.

5.1.1 Conceitos trabalhados

Aqui serão citados alguns conceitos e formas de trabalhar que nortearam ou foram relevantes para o desenvolvimento do Rachinations.

5.1.1.1 Domain-driven design (projeto orientado ao domínio)

O domínio é a parte de um sistema que representa a parte do mundo que se deseja modelar. O DDD (na sigla em inglês) busca reduzir erros e garantir a consistência do domínio como meio para domar a complexidade e aumentar a qualidade do software escrito principal mas não exclusivamente de sistemas de informação e sistemas corporativos.

O principal livro que trata deste tema é provavelmente (EVANS 2003).

5.1.1.2 Desenvolvimento top-down

Top-down (de cima para baixo) e bottom-up (de baixo para cima) são estratégias que podem ser usadas quando se escreve sistemas. Respectivamente, elas se referem a pensar o sistema primariamente em alto nível e depois codificar os detalhes e a pensar o sistema em termos de seus componentes de granularidade mais fina e depois juntar as partes para formar os fluxos de alto nível do sistema. (C2.COM 2014)

No Rachinations foi usado desenvolvimento top-down sempre que possível, partindo da premissa pessoal que, este estilo de programação facilita que humanos pensem o sistema e entendam o que está por baixo do mesmo.

Se fosse usado desenvolvimento *bottom-up* (isto é, pensar nos elementos como nós e conectores), é possível que, uma vez que os principais fluxos do programa fossem feitos, houvesse necessidade de muitos *hacks* e atalhos para fazer com que o nível *macro* se adaptasse ao que já foi pensado a nível *micro*, tornando o entendimento da dinâmica do sistema bastante difícil para eventuais leitores do código.

5.1.1.3 Programação funcional

A programação funcional é um paradigma de programação que, entre outras coisas, enfatiza funções sem efeitos colaterais (i.e., sempre retornam o mesmo valor dados os mesmos parâmetros de entrada e saída) no ambiente onde são executadas, promovendo assim uma maior previsibilidade e transparência no código criado. (HUGHES 1990)

Apesar de Ruby (a linguagem-base do sistema) não ser uma linguagem de programação funcional propriamente dita, ela ainda sim põe à disposição do programador algumas construções que são características do estilo funcional (C2.COM 2014), como funções como cidadãos de primeira classe (via blocos e `Proc`s), *list comprehensions* (via funções de ordem superior como `map`, `filter`, `reduce`), etc.

5.1.1.4 Baixo acoplamento

Tendo por base o documento em (C2.COM) e a lista de tipos de acoplamentos disponível em (UOTTAWA 2001), fica claro que o acoplamento entre dois componentes de um sistema (ou seja, o quanto um precisa do outro para funcionar corretamente) é uma das métricas mais importantes de qualidade em um sistema.

Apesar que certo nível de acoplamento entre os componentes do sistema (sejam eles classes, objetos ou módulos) seja indispensável pois eles precisam se comunicar, várias medidas foram tomadas com o objetivo de reduzir o acoplamento total do sistema (ou substituindo tipos mais severos por outros menos), como injeção de dependências (onde tudo que é externo a um objeto é passado como parâmetro – injetado – para o mesmo), parâmetros nomeados (para que a ordem de parâmetros não seja compartilhada) entre outros.

5.2 MANUTENÇÃO DA QUALIDADE

5.2.1 Desenvolvimento orientado a testes

É uma estratégia de desenvolvimento de sistemas originada em 1999 e que foi uma das origens do movimento de desenvolvimento ágil.

Esta técnica sugere que testes sejam escritos antes da codificação em si. Um livro razoavelmente antigo que trata do tema é (BECK 2002).

Isso ajudou a manter a qualidade do sistema, principalmente quando um só programador faz os testes e escreve o código; se o teste for feito pela mesma pessoa após a codificação do trecho de código a ser testado é difícil para a mesma não ser influenciada pelo conhecimento de já ter escrito o código em questão.

5.2.2 Integração contínua

Todo o processo de desenvolvimento foi acompanhado por um sistema de integração contínua. Integração contínua se refere, aqui, a um sistema que, a cada novo *commit* ou atualização do código, executa todos os testes e verifica que a nova modificação não fez com que as funcionalidades preexistentes deixassem de funcionar.

5.3 ESTRUTURA DO SISTEMA

A programação orientada a objeto busca limitar o acoplamento entre as entidades do sistema e agrupar estruturas de dados (objetos) junto com os comportamentos dos mesmos (métodos). Um dos principais conceitos da programação orientada a objeto são classes, que definem como serão os objetos gerados em tempo de execução.

A Figura a seguir mostra um diagrama de classes mostrando a relação entre as principais classes que compõem o domínio do problema (não mostrando classes utilitárias ou que, de outra forma, não representem uma entidade do diagrama).

Alguns métodos e atributos, que podem ajudar a entender como as diferentes partes se comunicam, também foram adicionados. Note que a ausência de um método, classe ou atributo deste diagrama não implica a ausência do mesmo do sistema em si.

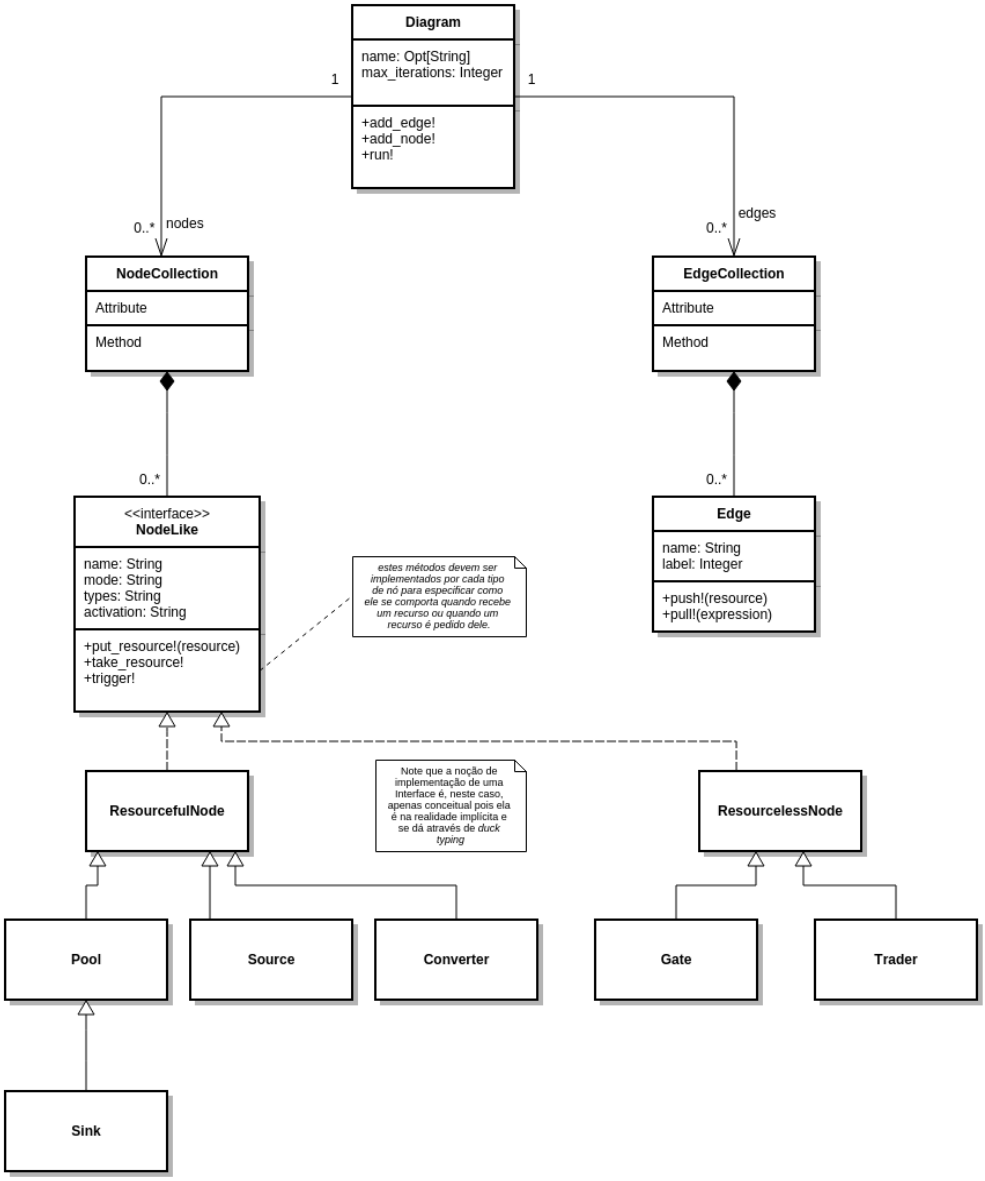

Figura 48 - A estrutura de classes do domínio mostra como alguns elementos do sistema se relacionam do ponto de vista de sua estrutura. Fonte: elaborada pelo autor

5.4 FUNCIONAMENTO DO SISTEMA

Um eventual revisor do código do sistema verá que o funcionamento do sistema pode ser divido em a) a montagem do diagrama (definição de quais elementos farão parte do mesmo) e b) a execução do diagrama.

Todo o processo de montagem do diagrama se dá antes do início da execução do mesmo. Isso é feito principalmente através dos métodos `add_edge!` e `add_node!` usados para adicionar um conector e um nó ao diagrama, respectivamente.

O código a seguir mostra um exemplo básico da criação e execução de um diagrama Rachinations:

```
d = Diagram.new 'simples'

d.add_node! Source, {
  :name => 'fonte',
  :activation => :automatic
}

d.add_node! Pool, {
  :name => 'depósito',
  :initial_value => 0
}

d.add_edge! Edge, {
  :name => 'conector',
  :from => 'fonte',
  :to => 'depósito'
}

# este diagrama é executado por 10 turnos
d.run!(10)
```

O trecho anterior equivale ao diagrama Machinations representado na figura a seguir:

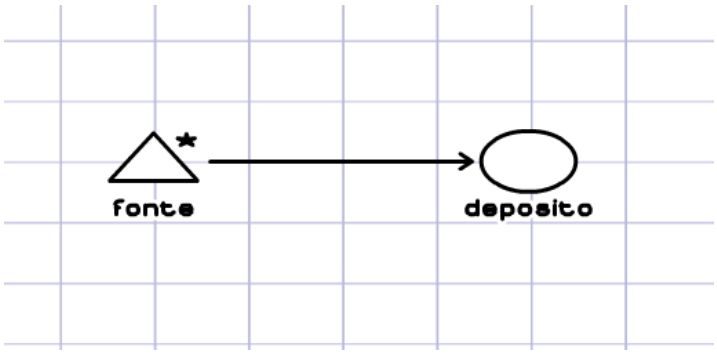

Figura 49 - Diagrama Machinations equivalente àquele descrito no trecho de código anterior. Fonte: elaborada pelo autor.

O processo de execução, por sua vez, é representado na interface pública da classe `Diagram` pelo método `run` que executa um certo número de turnos no diagrama ou até que uma das condições de término seja verificada.

Uma vez que o cliente chame o método `run`, a classe `Diagram` é responsável por consultar cada nó para saber se ele deve ser gatilhado (nós com ativação automática, por exemplo, devem ser gatilhados todos os turnos).

Em termos de código, o loop principal do sistema se encontra dentro do método `run_round!`, que equivale à execução de um turno no diagrama. No trecho a seguir,

adaptado do original, vemos a forma como o método `run!` se relaciona com o método `run_round!`

```
# arquivo diagram.rb (classe Diagram)

# ... outros métodos

def run!(rounds)

  # executa um número de turnos igual ao parâmetro passado
  rounds.times{ run_round! }

end
```

O método `run_round!` é chamado pelo método `run!` e é responsável, entre outras coisas, por consultar cada nó e, caso estejam ativos, ativá-los, através do método `trigger!` nos mesmos. O método `trigger!` é o principal método de um nó. Todos os nós implementam este método da sua maneira.

O trecho de código comentado a seguir (adaptado do original) mostra como estas etapas são efetivamente executadas:

```
# arquivo diagram.rb (classe Diagram)

# ... outros métodos

def run_round!

  # para cada nó neste diagrama
  nodes
  # se ele estiver ativo
  .select { |node| node.enabled? }
  # e for do tipo automático
  .select { |node| node.automatic? }
  # chame o seu método trigger!
  .each { |node| node.trigger! }

end
```

A figura a seguir (com mais detalhes) apresenta um diagrama de sequência conceitual para explicar melhor os conceitos explicados nos trechos de código:

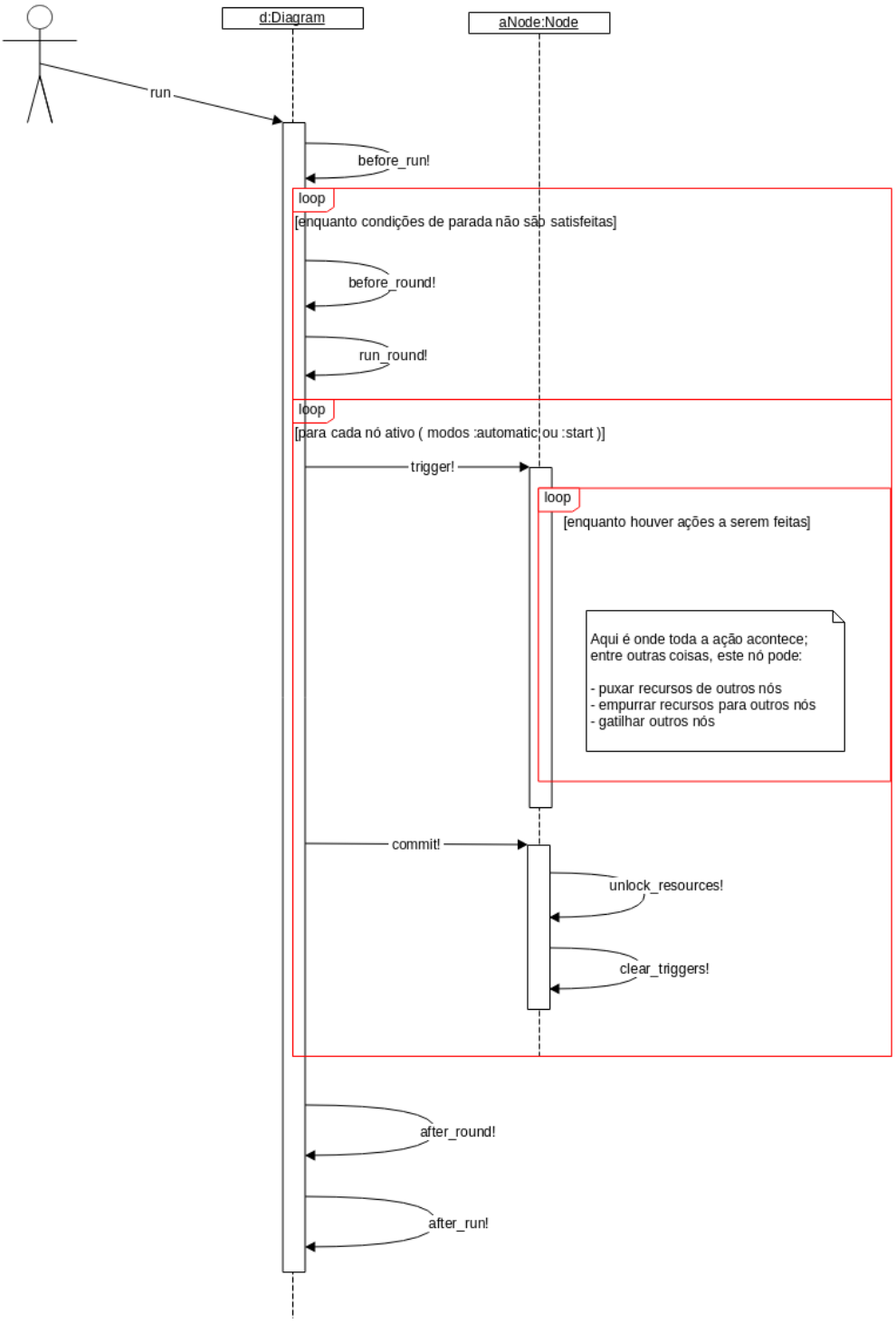

Figura 50 – Um diagrama de sequência (conceitual) que mostra como, uma vez que o método run! é chamado, outras ações são executadas no sistema. Fonte: elaborada pelo autor

6 A LINGUAGEM DE DEFINIÇÃO E EXECUÇÃO DE DIAGRAMAS

Uma das razões da dificuldade de usar o software Machinations para simular jogos e sistemas maiores é, sem dúvida, a dificuldade, inerente a todas as linguagens visuais, de serem usadas em escala.

Como já mencionado anteriormente, uma das motivações para a criação do Rachinations foi a de aperfeiçoar o aplicativo Machinations pois este, baseado em interfaces gráficas com elementos visuais, parece não escalar muito bem quando desejamos criar diagramas com muitos elementos ou criar abstrações de elementos.

O sistema até aqui descrito já contempla o problema da escala, pois permite a definição, execução e a extensão de diagramas Machinations através de código fonte. Ainda assim o sistema é uma biblioteca da linguagem Ruby e exige, desta forma, algum conhecimento da mesma e dificilmente poderá ser usado por não programadores.

Como forma de facilitar ainda mais o uso do sistema por não programadores e também de expor a usuários somente a complexidade necessária, criamos uma DSL (sigla para domain-specific language, em inglês) que fornece uma interface mais amigável para criação e execução de diagramas mas que não muda o funcionamento do mesmo; a DSL é somente uma *fachada* para o sistema previamente descrito.

A DSL fornece, portanto, meios para que um usuário leigo possa definir e executar diagramas mais facilmente do que se utilizasse o sistema diretamente.

Em sua forma mais simples, um diagrama Rachinations definido através da DSL tem a seguinte estrutura conceitual:

```
require 'rachinations'

diagram 'meu_diagrama' do

  # nós e conectores são adicionados aqui

end
```

6.1 ESTRUTURA DA DSL

A criação da DSL utilizou fortemente várias funcionalidades da linguagem ruby, em especial metaprogramação, composição (através de mixins) e sintaxe liberal.

A principal característica da DSL é o método `diagram`, como pode-se ver no exemplo acima. Este método instancia um diagrama e também define um bloco (delimitado pelas palavras-chave `do` e `end`) e executa os comandos no escopo do diagrama criado:

```
diagram 'meu_diagrama' do
```

```
  # mensagens enviadas aqui são executadas no escopo do
  # diagrama criado

end
```

Entretanto, o objeto diagrama criado (em cujo escopo os métodos são executados) não é simplesmente um diagrama normal; vários métodos da classe `Diagram` são sobrescritos, removidos e adicionados, em tempo de execução, de modo a blindar usuários de qualquer complexidade desnecessária para criar diagramas.

A seguir é mostrado um trecho de código adaptado mostrando como isso é feito:

```
# arquivo diagram_shorthand_methods.rb

module DSL
  module DiagramShorthandMethods

    # reabrindo a classe Diagram usando metaprogramação
    class ::Diagram

      # um método curto e com parâmetros livres é
      def pool(*args)

        hash = Parser.parse_arguments(args)

        # note que o controle é passado ao método original,
        # responsável pela adição do nó ao diagrama
        add_node! Pool, hash

      end

      # um método análogo, para fontes
      def source(*args)
        # conteúdo omitido
      end

    # resto do conteúdo omitido

  end
end
```

6.2 EXEMPLOS DE USO

Os exemplos a seguir servem não somente para mostrar o uso da DSL como

também para proporcionar ao leitor a oportunidade de comparar as duas formas de definição e execução de diagramas Rachinations (usando vis-à-vis não usando a DSL) e concluir como a adoção de uma DSL torna este problema mais palatável, principalmente do ponto de vista de um usuário – que não necessariamente tem experiência com programação.

A seguir vemos um diagrama simples com uma fonte e dois depósitos conectados por dois conectores. O diagrama Machinations correspondente é mostrado na Figura 51.

```
require 'rachinations'


diagram 'exemplo_1' do
  source 's1', :automatic
  pool 'p1'
  pool 'p2', :automatic
  edge from: 's1', to: 'p1'
  edge from: 'p1', to: 'p2'
end
```

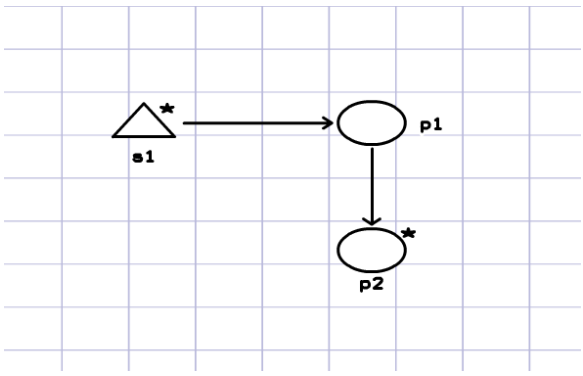

Figura 51 - Um simples diagrama Machinations equivalente ao diagrama Rachinations mostrado. Fonte: elaborada pelo autor

Já este diagrama Rachinations é um pouco mais elaborado e mostra outros elementos, como um conversor.

O diagrama Machinations equivalente é mostrado na figura 52.

```
require 'rachinations'

diagram 'exemplo_2' do
  source 's1'
  pool 'p1'
  converter 'c1', :automatic
  pool 'p2'
  pool 'p3'
```

```
  edge from: 's1', to: 'p1'
  edge from: 'p1', to: 'c1'
  edge from: 'c1', to: 'p2'
  edge from: 'c1', to: 'p3'
end
```

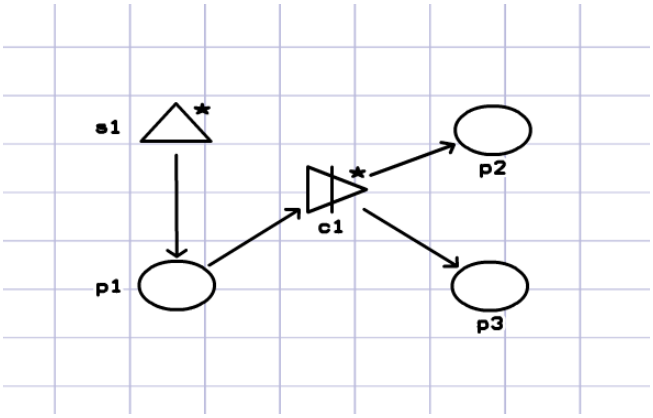

Figura 52 - Um diagrama Machinations um pouco mais elaborado mostrando outros tipos de nó. Fonte: elaborada pelo autor

Este diagrama Rachinations mostra outras características do sistema como portas e também a definição de condições por meio de expressões ad hoc, ou *lambdas*.

O diagrama Machinations correspondente se encontra na figura 53.

```
require 'rachinations'


diagram 'exemplo_3' do
  source 's1'
  gate 'g1', :probabilistic
  pool 'p1'
  pool 'p2'
  pool 'p3'
  sink 's2', :automatic, condition: expr{ p2.resource_count
> 30 }
  edge from: 's1', to: 'g1'
  edge from: 'g1', to: 'p1'
  edge 2, from: 'g1', to: 'p2'
  edge from: 'g1', to: 'p3'
  edge from: 'p3', to: 's2'
end
```

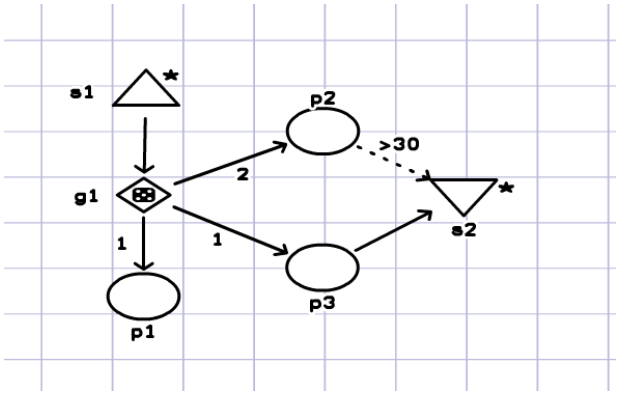

Figura 53 - Um diagrama Machinations mostrando uma porta probabilística e condições. Fonte: elaborada pelo autor

O trecho de código a seguir mostra o uso de condições de parada. O diagrama Machinations equivalente está na Figura 54:

```
require 'rachinations'


diagram 'exemplo_3' do
  source 's1', :automatic
  pool 'p1'
  edge from: 's1', to: 'p1'
  stop expr{ p1.resource_count > 5 }
end
```

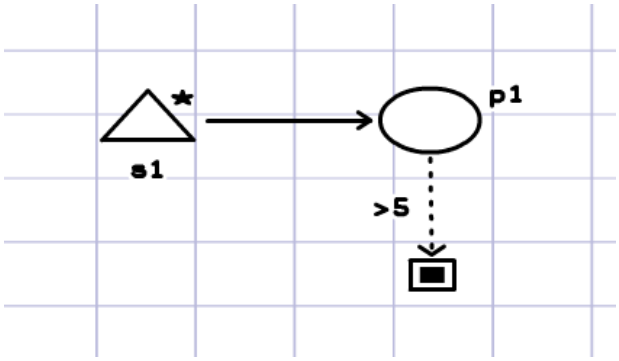

Figura 54 - Condições de parada podem ser usadas para testar hipóteses ou avaliar o estado de um diagrama

A seguir temos um diagrama Rachinations mostrando o uso da opção `triggers:` para definir gatilhos; o diagrama Machinations correspondente está na Figura 55.

```
require 'rachinations'


diagram 'exemplo_4' do
  source 's1'
  pool 'p1', triggers: 's2'
  source 's2', :passive
  pool 'p2'
  edge from: 's1', to: 'p1'
  edge from: 's2', to: 'p2'
end
```

Este trecho a seguir também gera o mesmo resultado do trecho anterior. A diferença é que o gatilho foi definido através da opção `triggered_by:`. Ele também corresponde à Figura 55.

```
require 'rachinations'


diagram 'exemplo_4_modificado' do
  source 's1'
  pool 'p1'
  source 's2', :passive, triggered_by: 'p1'
  pool 'p2'
  edge from: 's1', to: 'p1'
  edge from: 's2', to: 'p2'
end
```

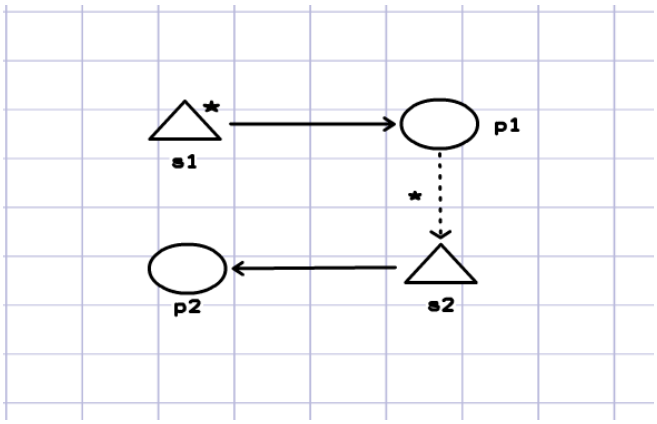

Figura 55 - Diagrama Machinations correspondente aos dois trechos de código anteriores. Fonte: elaborada pelo autor

6.3 ESPECIFICAÇÃO COMPLETA

Por se tratar de um documento estático, esta versão da especificação pode se tornar

obsoleta. Ver versão atualizada (em inglês) em http://github.com/queirozfcom/rachinations/blob/master/README.md#full-dsl-specification

A seguir serão mostradas, em detalhes, as opções e configurações que podem ser usadas na construção de diagramas.

Para facilitar a exposição, definimos os seguintes *tipos* de opções, que serão referenciados em cada item:

• IDENTIFICADOR: Um identificador ser qualquer string que possa ser usada como variável ruby, ou seja, que dê match na seguinte expressão regular: `/^[a-z_][a-zA-Z_0-9]*$/`. Por exemplo: `foo`, `_foo`, `foo123`.

• NÚMERO: Qualquer número não-negativo inteiro ou real, por exemplo: `1/3`, `0.3`, `0`, `999`.

• NATURAL: Qualquer número inteiro não-negativo, por exemplo: `0`, `1`, `999`.

• EXPRESSÃO: Uma expressão deve ser informada através da palavra `expr` e uma expressão entre chaves, por exemplo: `expr{ 30 > 10 }`, `expr{ p2.resource_count > 0 }`.

Vale lembrar que expressões são avaliadas de forma preguiçosa (*lazy*) somente quando são executadas. Além disso, é possível acessar elementos do diagrama (como nós e conectores) de dentro da expressão, como mostrado nos exemplos.

6.3.1 Criação de diagramas

A criação de um diagrama pode receber um nome e um modo, para controlar as informações que são mostradas na saída para o usuário (usualmente na linha de comando); as seguintes opções são suportadas:

- Nome
  - Cardinalidade: opcional
  - Tipo: IDENTIFICADOR
  - Valor default: nenhum
- Modo (mode)
  - Cardinalidade: opcional
  - Valores suportados: `:default`, `:silent` e `:verbose`
  - Valor default: `:default`

6.3.1.1 Exemplos

```
diagram 'diagrama_1' do
  # modo :default foi inferido
end
```

```
diagram mode: :silent do
  # diagrama sem node, modo silencioso
end
```

```
diagram 'diagrama_1', mode: :verbose do
  # modo :verbose imprime grande quantidade de informações
  # na saída
end
```

6.3.2 Depósitos, Fontes e Sumidouros

Depósitos, fontes e sumidouros (criados, respectivamente, pelos comandos `pool`, `source` e `sink`) podem recebem as seguintes opções:

- Nome
  - Cardinalidade: opcional
  - Tipo: IDENTIFICADOR
  - Valor default: nenhum
  - Obs.: caso presente, a opção nome deve sempre ser a primeira opção dada.
- Valor inicial
  - Cardinalidade: opcional
  - Tipo: NATURAL
  - Valor default: 0
  - Obs.: somente aplicável para depósitos
- Ativação (`activation`)
  - Cardinalidade: opcional
  - Valores suportados: `:passive`, `:automatic`, `:start`
  - Valor default: `:passive`
- Modo (`mode`)
  - Cardinalidade: opcional
  - Valores suportados: `:push_any`, `:pull_any`, `:push_all`, `:pull_all`
  - Valor default: `:pull_any` para depósitos e sumidouros, `:push_any` para fontes
- Condição (`condition`)
  - Cardinalidade: opcional
  - Tipo: EXPRESSÃO

- Gatilha (`triggers`)
  - Cardinalidade: opcional
  - Tipo: IDENTIFICADOR
  - Obs.: Deve ser um elemento existente do diagrama (ou que virá a ser criado)
- Gatilhado por (`triggered_by`)
  - Cardinalidade: opcional
  - Tipo: IDENTIFICADOR
  - Obs.: Deve ser um elemento existente do diagrama (ou que virá a ser criado)

6.3.2.1 Exemplos

```
# um depósito sem nome, ativação automática
pool :automatic
```

```
# um depósito nomeado, com ativação automática e
# com uma condição
pool 'p1', :automatic, condition: expr{ p2.resource_count >
30 }
```

```
# uma fonte que só é gatilhada uma vez, no início da execu
# cão do diagrama, no modo empurrar todos
source 's1', :start, :push_all
```

```
# um sumidouro passivo que gatilha um outro elemento 'p1'
sink 'sink1', :passive, triggers: 'p1'
```

6.3.3 Portas

Portas (criadas pelo comando gate) são usadas, entre outras coisas, para chavear recursos. Note que somente um recurso sai de uma porta por vez portanto rótulos de conectores saindo de uma porta têm semântica específica.

Portas podem receber as seguintes opções:

- Nome
  - Cardinalidade: opcional
  - Tipo: IDENTIFICADOR
  - Valor default: nenhum
  - Obs.: caso presente, a opção nome deve sempre ser a primeira opção dada.
- Tipo
  - Cardinalidade: opcional

    - Valores suportados: `:probabilistic`, `:deterministic`
    - Valor default: `:deterministic`
- Ativação (`activation`)
    - Cardinalidade: opcional
    - Valores suportados: `:passive`, `:automatic`, `:start`
    - Valor default: `:passive`
- Condição (`condition`)
    - Cardinalidade: opcional
    - Tipo: EXPRESSÃO
- Gatilha (`triggers`)
    - Cardinalidade: opcional
    - Tipo: IDENTIFICADOR
    - Obs.: Deve ser um elemento existente do diagrama (ou que virá a ser criado)
- Gatilhado por (`triggered_by`)
    - Cardinalidade: opcional
    - Tipo: IDENTIFICADOR
    - Obs.: Deve ser um elemento existente do diagrama (ou que virá a ser criado)

6.3.3.1 Exemplos

```
# uma porta nomeada, probabilística que gatilha outro nó
gate 'g1', :probabilistic, triggers: 'p1'
```

6.3.4 Conversores

Conversores (criados pelo comando converter) podem receber as seguintes opções:

- Nome
    - Cardinalidade: opcional
    - Tipo: IDENTIFICADOR
    - Valor default: nenhum
    - Obs.: caso presente, a opção nome deve sempre ser a primeira opção dada.
- Ativação (`activation`)
    - Cardinalidade: opcional
    - Valores suportados: `:passive`, `:automatic`, `:start`
    - Valor default: `:passive`
- Modo (`mode`)
    - Cardinalidade: opcional
    - Valores suportados: `:push_any`, `:pull_any`, `:push_all`, `:pull_all`
    - Valor default: `:push_all`

6.3.4.1 Exemplos

```
# um conversor nomeado, com ativação automática
converter 'c1', :automatic
```

6.3.5 Conectores

Conectores (criados pelo comando edge) podem receber as seguintes opções:

- Nome
  - Cardinalidade: opcional
  - Tipo: IDENTIFICADOR
  - Valor default: nenhum
  - Obs.: caso presente, a opção nome deve sempre ser a primeira opção dada.
- Nó de origem
  - Cardinalidade: obrigatório
  - Tipo: IDENTIFICADOR
  - Obs.: Deve ser um elemento existente do diagrama (ou que virá a ser criado)
- Nó de destino
  - Cardinalidade: obrigatório
  - Tipo: IDENTIFICADOR
  - Obs.: Deve ser um elemento existente do diagrama (ou que virá a ser criado)
- Rótulo (`label`)
  - Cardinalidade: opcional
  - Tipo: NÚMERO
  - Valor default: 1

6.3.5.1 Exemplos

```
# um conector básico
edge from: 'p1', to: 'p2'
```

```
# identificadores podem ser úteis se quisermos nos refe
# rir a este elemento mais tarde
edge 'conector1', from: 'p1', to: 'p2'
```

```
# conectores podem permitir a passagem de vários recursos
# por turno
edge 'grande_via', 5, from: 'p1', to: 'p2'
```

6.3.6 Condições de parada

- Nome
  - Cardinalidade: opcional
  - Tipo: IDENTIFICADOR
  - Valor default: nenhum
  - Obs.: Nomes são obrigatórios quando houver mais de uma condição de parada
- Condição
  - Cardinalidade: obrigatório
  - Tipo: EXPRESSÃO

6.3.6.1 Exemplos

```
# a expressão pode acessar elementos do diagrama como nós e
# conectores, assumindo que eles existam
stop expr{ p1.resource_count > 10 }
```

```
# quando mais de uma condição de parada é definida é neces-
# sário informar descrições
stop 'condição1', expr{ p1.resource_count > 20 }
stop 'condição2', expr{ p5.resource_count == 40 }
```

7 CONCLUSÃO

Com a noção da dinâmica de sistemas permeando nosso pensamento e usando como base o framework Machinations idealizado e implementado (em um aplicativo web) pelo professor holandês Joris Dormans, criamos o Rachinations, um pequeno sistema que permite a um usuário definir e executar diagramas baseados naquele framework.

O projeto nos permitiu identificar os pontos fortes do trabalho do professor Dormans e adaptá-lo para uma forma diferente de uso, não através de um GUI (pois interfaces gráficas não são facilmente usadas em escala) mas através de uma DSL criada especialmente para este fim.

Ficou clara para nós a aplicabilidade de conceitos de dinâmica de sistemas para a modelagem de jogos como os descritos neste documento. Jogos são sistemas complexos e parece que é possível usar todo o arcabouço de dinâmica de sistemas – já bem desenvolvido – para o estudo dos mesmos.

O uso de uma DSL para abstrair o uso de um sistema computacional complexo fornecendo uma interface menos complexa também mostrou bons resultados. Foi possível criar uma forma mais fluida e prática, o que pode encurtar os ciclos de hipótese e teste que eventuais usuários possam fazer.

O fato de o projeto inteiro ser código *open source* (código aberto) e de estar disponível na plataforma de compartilhamento Github (github.com) ajudou no desenvolvimento e ajudará também na disseminação do código caso outros desenvolvedores venham a se juntar a este projeto.

7.1 UTILIDADE

O sistema pode ser usado para fins similares àqueles do framework Machinations, ou seja, para testar hipóteses relacionadas a jogos, identificar padrões de comportamento nos mesmos, etc.

É possível também imaginar o Rachinations sendo usado para modelar outros fenômenos, não necessariamente jogos, que também possam ser descritos em termos de recursos, nós e conectores.

7.2 TRABALHOS FUTUROS

O sistema está apenas em uma de suas primeiras versões e pode ser estendido de muitas formas.

Dentre as formas que nos parecem mais promissoras estão a criação de algum mecanismo de abstração e componentização mais poderoso, de forma a permitir que diagramas sejam partes de outros diagramas, aumentando o nível da linguagem (pois as primitivas serão semanticamente mais ricas), por assim dizer.

Assim como um quadro no qual seu criador sempre acha que falta algo, um sistema nunca está realmente pronto e isso não deixa de ser verdade com o Rachinations. Deixando de lado as funcionalidades por um instante, há muito a se fazer no que tange à usabilidade do sistema, por exemplo; várias opções novas podem ser adicionadas para facilitar ações que se repitam muito frequentemente, de modo a facilitar o trabalho de futuros usuários.

Outro campo que nos parece ser digno de atenção é o do uso do sistema para sistemas complexos genéricos, não somente jogos. Pode ser proveitoso usar o Rachinations para outros sistemas que se queira estudar do ponto de vista da dinâmica de sistemas.

O Rachinations é, em última instância, um sistema complexo em si mesmo, assim como os sistemas que ele pode ser usado para modelar. Sendo assim, também mostrará, no seu tempo, um grau de imprevisibilidade e de comportamento emergente que é comum a todos os sistemas deste tipo.

Desta forma é impossível prever o uso que outras pessoas farão dele e é neste ponto que mora toda a beleza dos sistemas complexos.

8 REFERÊNCIAS

INTRODUÇÃO

SYSTEMDYNAMICS.ORG. **U.S. Department of Energy's Introduction to System Dynamics**. 1997. Disponível em <http://www.systemdynamics.org/DL-IntroSysDyn/start.htm>. Acessado em 10 de Janeiro de 2015.

DORMANS, J. **Engineering Emergence: Applied Theory for game Design.** 2011.

HUIZINGA, Johan. **Homo Ludens**. Londres. Routledge & Kegan Paul Ltd. 1949.

BANSAL, Arvind**. Introduction to Programming Languages.** 2013.

DINÂMICA DE SISTEMAS

SYSTEMDYNAMICS.ORG. **Origin of System Dynamics**. 1997. Disponível em <http://www.systemdynamics.org/DL-IntroSysDyn/origin.htm.> Acessado em 10 de Dezembro de 2014.

NICOLIS G, ROUVAS-NICOLIS, C. **Complex systems**. 2007. Disponível em <http://www.scholarpedia.org/article/Complex_systems>. Acessado em 10 de Dezembro de 2014.

SYSTEMDYNAMICS.ORG. **Stocks and Flows.** 1997. Disponível em <http://systemdynamics.org/DL-IntroSysDyn/stock.htm>. Acessado em 10 de Novembro de 2014.

SYSTEMDYNAMICS.ORG. **Feedback**. 1997. Disponível em <http://www.systemdynamics.org/DL-IntroSysDyn/feed.htm>. Acessado em 10 de Novembro de 2014.

O FRAMEWORK MACHINATIONS

DORMANS, J**. Engineering Emergence: Applied Theory for game Design.** 2011.

A LINGUAGEM RUBY

BARD. **Top Github Languages of 2014 (so far) - Adam Bard and his magical blog.** 2014. Disponível em <http://adambard.com/blog/top-github-languages-2014/>. Acessado em 2 de Dezembro de 2014.

GAMMA, Erich et al. **Design Patterns: Elements of Reusable Object-Oriented Software.** 1994. Addison-Wesley.

C2.COM. **Meta Programming.** 2015. Disponível em <http://c2.com/cgi/wiki?MetaProgramming>. Acessado em 25 de Janeiro de 2015.

SINATRARB.COM. **Getting Started.** 2014. Disponível em

<http://sinatrarb.com/intro.html>. Acessado em 2 de Dezembro de 2014.

CHEF. **Getting Started with Chef.** 2014. Disponível em <http://gettingstartedwithchef.com/first-steps-with-chef.html>. Acessado em 2 de Dezembro de 2014.

LEARNCHEF. **Create your first cookbook.** 2014. Disponível em <https://learn.chef.io/legacy/tutorials/create-your-first-cookbook>. Acessado em 2 de Dezembro de 2014.

O SISTEMA RACHINATIONS

EVANS, Eric. **Domain-Driven Design: Tackling Complexity in the Heart of Code.** 2003. Addison-Wesley.

C2.COM. **Top Down Design**. 2014. Disponível em <http://c2.com/cgi/wiki?TopDownDesign>. Acessado em 10 de Dezembro de 2014.

HUGHES, J. **Why Functional Programming Matters**. 1990. Em "Research Topics in Functional Programming", Addison-Wesley.

C2.COM. **Functional Programming.** 2014. Disponível em <http://c2.com/cgi/wiki?FunctionalProgramming>. Acessado em 2 de Dezembro de 2014.

C2.COM. **Coupling And Cohesion.** 2014. Disponível em <http://c2.com/cgi/wiki?CouplingAndCohesion>. Acessado em 2 de Dezembro de 2014.

UOTTAWA (Universidade de Ottawa). **Object Oriented Software Engineering Knowledge Base**. 2001. Disponível em <http://www.site.uottawa.ca:4321/oose/index.html#coupling>. Acessado em 2 de Dezembro de 2014.

BECK, Kent. **Test Driven Development: By Example.** 2002. Addison-Wesley Professional.