UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

VALIDAÇÃO E ADAPTAÇÃO DO FLATREDBALL GAME ENGINE PARA FÁBRICAS DE JOGOS ARCADE 2D

TRABALHO DE GRADUAÇÃO

Aluna: Laís de Mendonça Neves (lmn3@cin.ufpe.br)

Orientador: André Luís de Medeiros Santos (alms@cin.ufpe.br).

Co-Orientador: André Wilson Brotto Furtado (awbf@cin.ufpe.br).

Novembro de 2008

Universidade Federal de Pernambuco
Centro de Informática

Laís de Mendonça Neves

Validação e Adaptação do FlatRedBall Game Engine para Fábrica de Jogos Arcade 2D

Monografia apresentada ao Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Orientador: André Luís de Medeiros Santos
Co-orientador: André Wilson Brotto Furtado

Recife, 29 de novembro de 2008

A Jorge e Paula, pais e grandes incentivadores.

Agradecimentos

"Eis o meu segredo: só se vê bem com o coração.
O essencial é invisível aos olhos."
(Antoine de Saint-Exupéry)

A Deus.

Aos meus pais, que sempre me mostraram a grande importância da educação.

Aos meus irmãos, pelas coisas boas e ruins da convivência diária.

A minha enorme família, por estar sempre presente.

A Marcelo, por ter me encontrado tão longe e agora estar tão perto, e pela motivação nos momentos em que precisei.

Aos amigos de infância da escola, à turma CC2004.1 (G4 & agregados, comissão de formatura, Titanium e Commit Solutions) e aos grandes amigos que pude fazer nesses cinco anos de Centro de Informática e de UFPE.

Ao CIn e à UFPE pela oportunidade de conhecer um pouco do mundo lá fora.

Aos professores do CIn, os quais eu tive a honra de conhecer e com os quais convivi durante a graduação, por serem os responsáveis pela profissional que sou hoje.

À equipe do CIn/Motorola Research Lab pela compreensão e paciência durante os momentos que tive de me dedicar a este trabalho.

Ao professor André Santos e a André Furtado pela orientação prestada durante a realização deste trabalho.

A Tia Regina, pelas aulas de redação na época do vestibular e pela adequação deste trabalho à norma culta da língua portuguesa.

A Renato e a Marcelo, pela ajuda com as figuras e as músicas.

A todos os meus queridos testadores dos jogos.

Muito obrigada!

Resumo

O objetivo principal deste trabalho é o desenvolvimento de jogos para validar se o *game engine* FlatRedBall é capaz de suportar as funcionalidades (*features*) do domínio de jogos *arcade* 2D. Haverá um foco em permitir que tal *engine* seja consumido de maneira mais abstrata e intuitiva possível pelo código gerado a partir das linguagens de domínio especifico (DSLs) da fábrica ArcadEx, uma fábrica de software focada no desenvolvimento de jogos *arcade* 2D, integrante do projeto SharpLudus.

Dentro do escopo do trabalho, serão propostas melhorias na camada de abstração existente em cima do FlatRedBall, a qual facilita o consumo da *engine* pela ArcadEx. Por fim, este trabalho irá proporcionar *feedback* referente a similaridades e pontos de variabilidades encontrados no processo de desenvolvimento dos jogos, analisando como implementá-los de maneira mais fácil utilizando o FlatRedBall e permitindo que as DSLs visuais da fábrica ArcadEx sejam ajustadas e, eventualmente, novas DSLs sejam criadas.

Palavras-chave: fábricas de *software*, *domain-specific development*, dsl, *game engines*, jogos *arcade* 2D

Abstract

The main purpose of this work consists in the development of digital games to validate if the FlatRedBall game engine is able to support the features of the 2D arcade games domain. There will be an effort to enable the engine to be consumed in a more abstract and intuitive way by the code generated through the domain-specific languages of the ArcadEx factory, a game factory focused in developing 2D arcade games, as part of the SharpLudus project.

In the scope of this work, improvements will be proposed on the abstraction layer existing above the FlatRedBall, which makes it easy to consume the game engine from ArcadEx. Finally, this work will give feedback relative to the commonalities and variabilities found in the game development process, analyzing how to implement them in a easier way using FlatRedBall and allowing the factory visual DSLs to be adjusted, and, eventually, new DSLs to be created.

Key words: software factories, domain-specific development, dsl, game engines, 2D arcade games

Sumário

Lista de Figuras

Lista de Tabelas

1. Introdução

1.1. Motivação

Atualmente, o mercado de jogos eletrônicos tem se mostrado uma oportunidade de negócio extremamente lucrativa. Estudos recentes [IbisWorld] mostram que o setor espera crescer em 2008 cerca de 9,5%, gerando um total de lucros da ordem de 39 bilhões de dólares. Em 2007, foram vendidos, apenas nos Estados Unidos, atualmente o maior mercado mundial, mais de 34 milhões de consoles. Os investimentos nesse mercado chegam até mesmo a ultrapassar os investimentos destinados a outras áreas de entretenimento, como cinema, que movimentou 9,3 bilhões de dólares em 2007 e teve um crescimento de 5.4% nesse mesmo ano [MPAA 2007]. No Brasil, apesar dos prejuízos com pirataria e dos altos impostos pagos pelas indústrias de desenvolvimento de games nacionais, a previsão para 2008 é que o produto nacional bruto do setor de jogos seja de R$ 87,5 milhões [Abragames 2008].

O desenvolvimento de jogos tornou-se uma área interessante da computação por englobar diferentes áreas de conhecimento, como computação gráfica, redes de computadores e inteligência artificial. A evolução dos jogos eletrônicos, que estão cada vez mais complexos do ponto de vista do desenvolvimento, e também a modernização dos consoles têm trazido novos desafios aos desenvolvedores. Os usuários estão cada vez mais exigentes e acostumados a jogos com qualidade gráfica excepcional e a narrativas não lineares que dependem de variáveis quase infinitas de ações e levam em consideração os últimos avanços da Inteligência Artificial [Lopes 2007]. Na só os jogos, mas a maneira de desenvolvê-los também evoluiu. Atualmente, é possível que pessoas que possuam um mínimo conhecimento de programação possam criar seus próprios jogos. Isso se deve também ao surgimento da Web 2.0 [Web 2.0 Wikipedia] que, entre outras coisas, facilitou a criação e o acesso ao conhecimento para os seus usuários.

Por outro lado, pesquisas apontam que o desenvolvimento de software da maneira praticada atualmente é lento, caro e propenso a erros, o que freqüentemente gera produtos com grande número de defeitos, como problemas de usabilidade, performance, entre outros [Greenfield 2004]. O crescimento da indústria de jogos eletrônicos e o importante papel dos *games* na sociedade atual demandam uma mudança importante no paradigma de desenvolvimento atual. A fim de obter ganhos com produtividade e redução de custos, a industrialização no processo de desenvolvimento de *software* vem ganhando destaque.

Outras indústrias tiveram um grande aumento em sua capacidade de produção quando evoluíram de um processo artesanal, onde os produtos eram criados do zero, por um pequeno grupo de pessoas, para o processo de manufatura, no qual os produtos desenvolvidos por diferentes fornecedores são montados na fábrica para dar origem a vários produtos diferentes e onde máquinas automatizam grande parte do processo.

Fábricas de software constituem um paradigma de desenvolvimento que viabiliza a industrialização no processo de criação de *software*. Uma fábrica de *software* pode ser definida como sendo um ambiente de desenvolvimento configurado para suportar a construção de uma aplicação específica de forma mais rápida, utilizando padrões empregados na produção industrial [Greenfield 2004]. Neste ambiente, desenvolvedores encapsulam seus conhecimentos em bens reutilizáveis que podem ser aplicados em diferentes projetos. Para viabilizar o conceito de fábricas de *software*, faz-se o uso de modelos, linguagens, *frameworks* e ferramentas para automatizar ainda mais o processo de desenvolvimento de *softwares*.

No contexto de fábricas de *software*, a utilização de linguagens de domínio especifico (*Domain-Specific Languages* - DSLs) tem um papel importante. Uma DSL é uma linguagem pequena, normalmente declarativa, que oferece maiores funcionalidades ao desenvolvedor, sendo focada em um problema de domínio específico [Deursen 1999]. Atualmente, é sabido que uma abordagem genérica fornece uma solução geral para vários problemas em uma

certa área, mas tal solução pode não ser ótima. Porém uma abordagem específica fornece uma melhor solução para um conjunto limitado de problemas. As linguagens de programação mais antigas (*Cobol, Fortran, Lisp*) foram criadas para resolver problemas em uma determinada área. Elas foram evoluindo gradualmente para linguagens de propósito genérico que utilizam atualmente *frameworks* e bibliotecas para solucionar problemas específicos. Em uma fábrica de *software*, podemos utilizar uma DSL para o domínio e desenvolver ferramentas que suportam essa linguagem, como editores e compiladores, para automatizar o processo de montagem dos componentes.

Outro princípio igualmente importante, no qual estão fundamentadas as fábricas de *software,* consiste na utilização de *frameworks*, que encapsulam e permitem o reuso de pontos de similaridades compartilhados por uma mesma família de produtos, além de suportar pontos de variabilidade através de mecanismos como parametrização e herança, entre outros. Em se tratando do desenvolvimento de jogos, tais *frameworks* são realizados através de motores para jogos (*game engines*) [Furtado 2006].

A proposta deste trabalho, portanto, consiste em validar e adaptar o motor de jogos FlatRedBall [FlatRedBall] como um *framework* a ser utilizado em fábricas de *software* focadas no desenvolvimento de jogos *arcade* 2D. Tal trabalho será desenvolvido no contexto do projeto SharpLudus [Furtado 2006], cujo objetivo é definir processos e ferramentas para geração de fábricas de jogos.

1.2. Objetivo

O projeto SharpLudus tem como objetivo definir processos e ferramentas para fábricas de *software* aplicadas no desenvolvimento de jogos digitais. A fábrica de *software* ArcadEx é uma instância do projeto SharpLudus, focada no desenvolvimento de jogos *arcade* 2D. Entre os *assets* (“bens”) disponibilizados por essa fábrica aos desenvolvedores de jogos estão diferentes DSLs visuais focadas, cada uma, em um conjunto de características dos jogos a serem gerados, tais como: transição entre telas, máquina de estado para entidades do

jogo, *heads-up displays* (HUDs) para visualização de informações e estatísticas referentes ao jogo, etc.

A fim de permitir que os diagramas modelados através das DSLs visuais da fábrica ArcadEx sejam transformados no código final do jogo, é necessário validar e adaptar um framework (*game engine*) a ser consumido por tal código gerado. O objetivo principal deste trabalho consiste no desenvolvimento de jogos para validar se o motor FlatRedBall, um dos candidatos para a fábrica ArcadEx, é capaz de suportar as funcionalidades do domínio da fábrica, jogos *arcade* 2D. Ao mesmo tempo, haverá um foco em permitir que tal *engine* seja consumido de maneira mais abstrata e intuitiva possível pelo código gerado a partir das DSLs da fábrica ArcadEx. Dessa forma, este trabalho também realizará uma validação da camada de abstração já existente em cima do FlatRedBall, propondo possíveis pontos de extensão e melhorias na mesma, que irão facilitar o seu consumo pela fábrica. Por fim, procurará retornar à fábrica ArcadEx feedback referente a similaridades e pontos de variabilidades encontrados no processo de desenvolvimento dos jogos, permitindo que suas DSLs visuais sejam ajustadas e, eventualmente, novas DSLs sejam criadas pelos projetistas da fábrica.

1.3. Organização da Monografia

Este trabalho de graduação está organizado da seguinte maneira:

- O capítulo 2 aborda o conceito de fábricas de software, as vantagens de se utilizar esse novo paradigma e como DSLs estão inseridas neste contexto.
- O capítulo 3 apresenta o projeto SharpLudus e a fábrica de jogos ArcadEx.
- O capítulo 4 descreve o *game engine* FlatRedBall e os jogos que foram implementados para realizar a sua validação no contexto da fábrica ArcadEx.
- No capítulo 5, encontram-se as conclusões deste trabalho e as propostas de trabalhos futuros.

2. Fábricas de Software e Domain-Specific Development

O conceito de fábricas de *software* visa a automatizar o desenvolvimento de sistemas computacionais com a finalidade de obter ganhos em produtividade. Embora a indústria dessa área ainda seja relutante em adotar paradigmas de desenvolvimento baseados em manufatura, a crescente pressão pela diminuição dos custos e dos prazos de entrega fez com que as empresas investissem em melhorias na qualidade de seus produtos, o que acabou gerando automatização de parte de seus processos.

O conceito de fábricas de *software* pode ser mal interpretado, alegando-se que o caráter criativo na elaboração de um software jamais seria compatível com um modelo de produção similar ao de uma fábrica. Porém acredita-se que progressos significativos podem ser alcançados fundamentando-se no paradigma de orientação a objetos e guiados pelas crescentes metodologias ágeis de desenvolvimento. [Greenfield 2003] Reconhecendo-se que o processo de desenvolvimento de *software* é uma atividade orientada a pessoas que não pode ser reduzida a processos puramente mecânicos e determinísticos, utiliza-se uma abordagem na qual é adotada um vocabulário próximo ao domínio do problema e que delega a maior parte dos aspectos mecânicos e determinísticos às ferramentas de desenvolvimento.

Nos últimos anos, a indústria de *software* passou a lidar com as demandas de uma sociedade que se torna cada vez mais automatizada, baseando-se nas habilidades individuais dos desenvolvedores, da mesma maneira como os artesãos tratavam as demandas de uma sociedade crescentemente industrializada no inicio da revolução industrial, baseando as atividades nas habilidades individuais de cada artesão. Porém os meios de produção de *software* atuais estão chegando próximo de suas capacidades máximas, pois a capacidade de uma indústria baseada em processos artesanais

é limitada por seus métodos e ferramentas e pela quantidade de trabalhadores qualificados.

Quando se depararam com desafios similares muitos anos atrás, outras indústrias evoluíram do processo artesanal para o processo de manufatura, aprendendo a customizar e montar diferentes componentes padronizados para produzir produtos similares, porém distintos. Esta mudança foi realizada graças a uma padronização e automação do processo de produção, no qual se desenvolveram diversas ferramentas que permitiram que tarefas repetitivas fossem automatizadas, proporcionando diminuição de gastos e ganho de tempo. A industrialização gerou também uma cadeia de fornecedores independentes e extremamente especializados, na qual os riscos e custos existentes na produção do produto final passaram a ser distribuídos por toda a cadeia produtiva.

2.1. Fábricas de *Software*

Atualmente, já é possível encontrar algumas ferramentas que permitem automatizar algumas etapas durante o processo de desenvolvimento de *software*. *Frameworks* para geração de componentes gráficos e editores visuais facilitam a construção e manutenção de *GUIs* (Graphical User Interfaces). A modelagem de banco de dados também oferece formas similares de automação. Analisando essas soluções, é possível encontrar um padrão recorrente:

- Após desenvolver vários sistemas para um mesmo domínio de problemas, é possível identificar um conjunto de abstrações reusáveis para este domínio e, então, documentar uma série de padrões para utilizar essas abstrações.
- *Frameworks* são desenvolvidos para codificar as abstrações e padrões encontrados. Isto permite construir sistemas no domínio do problema instanciando, adaptando, configurando e montando diferentes componentes. No caso de uma fábrica de *software* para o desenvolvimento de jogos, um *game engine* desempenharia este papel [Furtado & Santos 2006].

- Linguagens para o domínio são definidas e são construídas ferramentas para suportar a linguagem, como editores, compiladores e *debuggers*, que automatizam o processo de montagem do programa. Isto ajuda a responder rapidamente às mudanças de requisitos que acontecem durante o projeto, já que parte da implementação é gerada automaticamente e pode ser facilmente modificada.

Segundo [Greenfield 2004], uma fábrica de *software* pode ser considerada como sendo uma linha de produção que configura ferramentas de desenvolvimento extensíveis que são cuidadosamente criadas para construir aplicações específicas. Uma fábrica de *software* contém três idéias principais: um software *factory schema*, um *software factory template* e um ambiente de desenvolvimento extensível:

- O *software factory schema* lista os componentes da fábrica, como projetos, diretórios para código fonte, arquivos SQL e de configuração, e explica como eles devem ser combinados para criar um produto. Ele especifica também quais DSLs devem ser utilizadas e descreve como modelos baseados nessas DSLs podem ser transformados em código ou em outros artefatos. Finalmente, o *schema* descreve a arquitetura da fábrica e as relações entre os componentes e frameworks integrantes da fábrica.
- O *software factory template* implementa (realiza) os artefatos citados no *schema*. Ele provê padrões, orientação, *templates*, *frameworks*, exemplos, ferramentas customizadas, como editores visuais para as DSLs, *scripts* e outros ingredientes utilizados para desenvolver o produto.
- Um ambiente de desenvolvimento extensível, como o Visual Studio Team system (VSTS) [VSTS], que, quando configurado através do software *factory template,* torna-se uma fábrica de *software* para a família de produtos do domínio.

Fazendo uma analogia, os produtos são como as refeições servidas por um restaurante. Os *stakeholders* são como clientes do restaurante que fazem

pedidos de refeições existentes no cardápio. Os desenvolvedores de produto são como os cozinheiros que preparam as refeições descritas nos pedidos dos clientes e podem modificar as receitas das refeições criadas por outros cozinheiros, ou ainda criar uma nova refeição. Os desenvolvedores da fábrica são como os *chefs* de cozinha, que decidem quais pratos vão aparecer no cardápio e quais serão os ingredientes, processos, e equipamentos utilizados para a criação de cada prato.

A figura abaixo ilustra os bens disponíveis em uma fábrica de *software* para *web clients*.

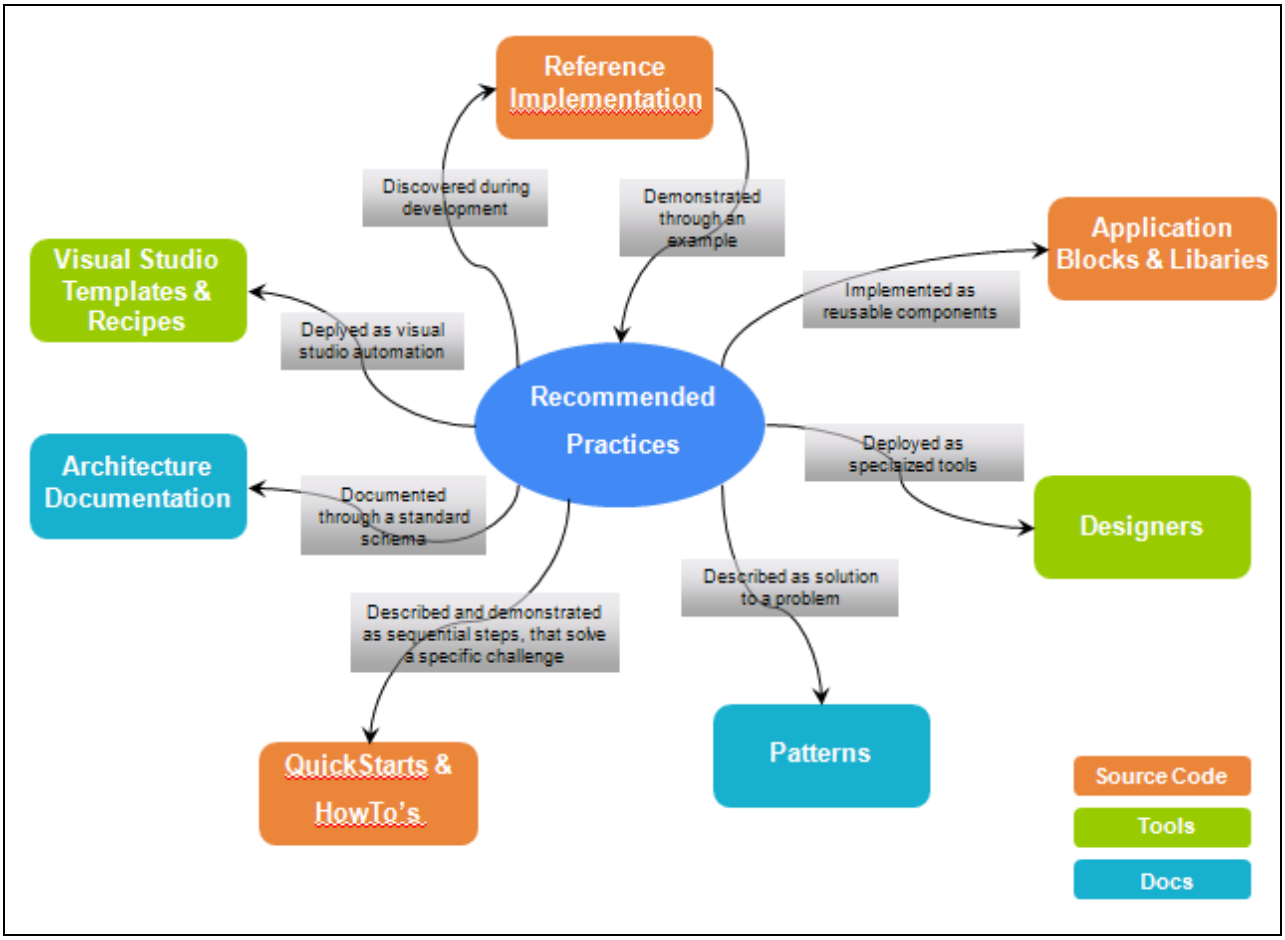


Figura 1 - Bens disponíveis em uma fábrica de *software* para *web clients*.

No contexto deste trabalho, será descrito no capítulo seguinte a fábrica de jogos Sharpludus [Furtado 2006], onde serão retomados os conceitos apresentados neste capítulo.

2.2. Domain-Specific Development

Domain-specific development é uma técnica baseada na observação de que muitos dos problemas de desenvolvimento de *software* podem ser mais facilmente resolvidos criando-se uma linguagem de propósito específico. Ao invés de utilizar apenas linguagens de propósito geral, para resolver esses problemas isoladamente, a prática de *domain*-specific *development* cria e implementa linguagens específicas, na qual cada uma delas pode resolver uma série de problemas similares. Essas linguagens, chamadas de *domain*-specific *languages* ou DSLs, podem ser tanto visuais quanto textuais.

Segundo [Deursen 1999], o desenvolvimento de uma DSL envolve basicamente as seguintes etapas:

1. Identificar o domínio do problema;
2. Reunir todo o conhecimento relevante para este domínio;
3. Agrupar este conhecimento em uma série de notações semânticas e operações;
4. Definir uma DSL que descreve concisamente as aplicações do domínio;
5. Construir uma biblioteca que implementa as notações semânticas;
6. Desenvolver e implementar um compilador que converte programas escritos na DSL para uma seqüência de chamadas à biblioteca.
7. Escrever programas na DSL para as aplicações desejadas e então compilá-los.

Domain-specific development pode ser aplicado quando um determinado problema ocorre repetidas vezes [Cook 2007]. Cada ocorrência do problema contém muitos aspectos que são similares e que podem ser resolvidos uma única vez, sendo a solução replicada para as demais ocorrências. Os aspectos do problema, que são diferentes a cada ocorrência, podem ser representados

por uma DSL e cada ocorrência particular do problema pode ser resolvida criando-se uma expressão ou um modelo na DSL especificada, o qual será integrado com a parte fixa da solução.

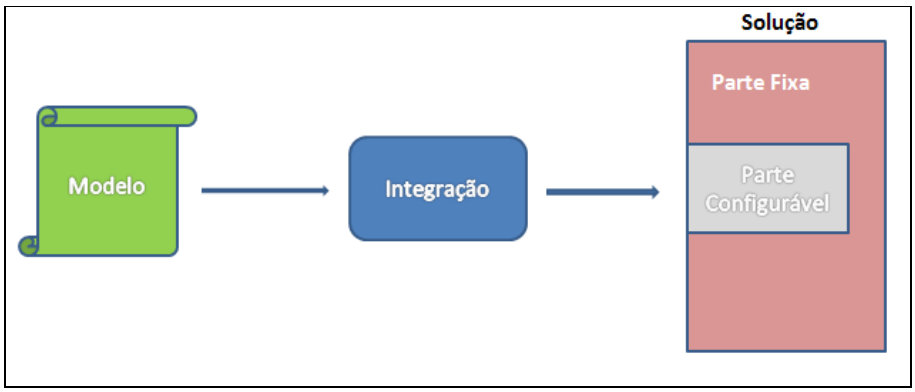


Figura 2 - ***Domain-specific development***

O desenvolvimento da parte fixa da solução também evoluiu baseado no contexto de *software product lines* (SPL), tendo como base itens diversos como *component-based development*, reuso, entre outros. Dependendo do tamanho e das características do problema, esta parte fixa da solução pode ser um *framework*, uma plataforma, um interpretador ou uma Application Programming Interface (API). A parte fixa captura os padrões arquiteturais que caracterizam o domínio e expõe pontos de extensão que permitem que ela seja utilizada por uma variedade de soluções. O que faz esta abordagem ser viável é o fato de que a parte variável da solução é desenvolvida utilizando-se DSLs. A esse conceito (parte fixa da solução sendo configurada por uma parte variável, eventualmente gerada por DSLs), dá-se o nome de *framework completion*.

Para criar uma solução funcional para o problema que se deseja atacar, é necessário integrar a parte fixa da solução com a parte variável, que é especificada pelo modelo. Existem duas abordagens para essa integração. A primeira é a abordagem interpretativa, na qual a parte fixa contém um interpretador para a DSL utilizada no desenvolvimento da parte variável do problema. Esta abordagem pode ser flexível, mas pode ter desvantagens como baixa performance e dificuldade de depuração. A outra abordagem consiste em

converter completamente o modelo criado na DSL em código que pode ser compilado juntamente com o restante da solução. Este procedimento pode ser mais complexo, porém proporciona vantagem em extensibilidade, performance e capacidade de depuração.

[Deursen 1999] e [Cook 2007] apontam uma série de benefícios e desvantagens na utilização de *domain-specific development*. Uma DSL bem projetada visa a encontrar o balanceamento adequado entre os riscos e as oportunidades.

Benefícios

- Uma DSL permite trabalhar em termos de domínio do problema, diminuindo os erros encontrados ao representar o problema através de uma linguagem de propósito geral;
- Trabalhar em termos do domínio do problema pode tornar os modelos mais acessíveis às pessoas que não estão familiarizadas com as tecnologias de implementação;
- Modelos criados baseados em DSLs podem ser validados no mesmo nível de abstração do domínio do problema, o que significa que os erros de concepção e representação podem ser descobertos mais cedo;
- Uma DSL provê uma API específica do domínio para manipular seus modelos, o que aumenta a produtividade do desenvolvedor;
- Uma vez que o conhecimento sobre o negócio é capturado por um modelo, torna-se consideravelmente mais fácil migrar a solução de uma tecnologia para outra, ou entre versões da mesma tecnologia, bastando realizar modificações nos geradores e interpretadores que consomem o modelo.

Desvantagens

- Custos relacionados à concepção, implementação e manutenção da DSL;

- Custos de aprendizado para os usuários da DSL;
- Dificuldade para encontrar o escopo adequado da DSL;
- Dificuldade de balancear o desenvolvimento em DSLs e em linguagens de propósito geral.
- Possível perda de eficiência quando comparada com software escrito à mão.

3. Projeto SharpLudus: Uma Fábrica de Software para Games

O projeto SharpLudus [Furtado 2006] surgiu como uma iniciativa de aplicar os conceitos de fábricas de software explicados anteriormente para o domínio de jogos digitais. Muitos desafios surgem nesta área, uma vez que o desenvolvimento de jogos é uma disciplina inerentemente criativa, talvez mais do que qualquer outra na computação, e onde originalidade e inovação são atributos intrínsecos dos jogos de sucesso. A idéia inicial do projeto focava no domínio de jogos de aventura, definindo uma DSL visual para a modelagem dos games criados pela fábrica.

Atualmente, a idéia do SharpLudus evoluiu para ser um ambiente onde são definidos processos para a construção de fábricas de *software* para games, cada qual focada em um domínio específico. Dentro do escopo deste trabalho de graduação, será analisada a fábrica de *games* ArcadEx, voltada para o domínio de jogos *arcade* 2D.

3.1. ArcadEx Game Software Factory

A fábrica de jogos ArcadEx consiste em uma abordagem utilizada para aprimorar a automatização do processo de desenvolvimento de jogos digitais para o domínio arcade bidimensional (2D), utilizando técnicas de *domain-specific development*. Tal abordagem é focada na criação de fábricas de jogos que possuem linguagens de domínio específico visuais como seus bens mais importantes.

A primeira tarefa na criação de fábricas de jogos digitais, baseados em *domain-specific development* e DSLs, é propor uma visão de alto nível e um entendimento comum do domínio do problema a ser atacado. A seguir, será abordada a visão e o escopo da ArcadEx, bem como as *features* identificadas para o seu domínio.

3.1.1. Visão

A grande diversidade dos jogos, criados até então, tornaram o desenvolvimento de jogos uma atividade de domínio bastante abrangente. Porém a criação de fábricas de games e DSLs, que tentam focar em um amplo domínio, como jogos de plataforma 2D até simuladores de vôo 3D, por exemplo, constituem um esforço muito grande e ineficaz. Neste cenário, DSLs e outros bens disponibilizados pela fábrica podem acabar não conseguindo explorar completamente os benefícios desta abordagem, como reuso e montagem de componentes. Para solucionar este problema, recomenda-se a escolha de um subdomínio de jogos a ser atacado.

Uma das maneiras mais comuns de classificar os jogos é definir alguns gêneros, que juntos vão compor sua taxonomia. [Crawford 1984]. Alguns dos gêneros mais populares são, por exemplo, jogos de tabuleiro, jogos de luta, jogos de aventura, RPGs e jogos de estratégia.

Outro gênero existente agrupa os jogos *arcade*, baseados nas antigas máquinas de fliperama das décadas de 70 e 80. Esses jogos são caracterizados por possuírem níveis bastante curtos, controles simples e intuitivos e um grau de dificuldade que cresce de modo constante. A figura abaixo ilustra as antigas máquinas de fliperama de jogos *arcade*.

Figura 3 – Máquina de jogos *arcade*

Baseada nesse domínio, a fábrica ArcadEx terá como foco a geração de jogos *arcade* casuais e bi-dimensionais para PC, com níveis curtos e simples, e com um grau de dificuldade que cresce de modo constante. Os jogadores controlam personagens principais e interagem com outras entidades como personagens não-jogáveis (*non-player characters*, NPC) e itens. A condição de vitória é definida pelo designer do jogo como um conjunto de eventos, por exemplo: inimigos são derrotados, um objeto é coletado, o jogador atinge uma pontuação máxima, etc.

3.1.2. Escopo

A definição do escopo do *game factory* refina a visão estabelecida anteriormente, provendo mais detalhes sobre as funcionalidades (*features*) que estarão presentes nos jogos gerados pela fábrica. O escopo proporciona aos designers da fábrica a primeira oportunidade de vislumbrar quais DSLs serão úteis à fábrica, já que esta atividade exercita a identificação de similaridades e variabilidades das *features* dos jogos.

Para definir o escopo da ArcadEx, [Furtado *et al*. 2008] utilizou um processo iterativo e incremental, no qual os designers da fábrica definem um escopo base (*baseline scope*), o qual é muito provável de evoluir em futuras iterações após a análise do domínio. Os projetistas da fábrica observam quais são as *features* que pertencem ao domínio estudado e em um determinado momento, o escopo base da fábrica acaba sendo, de fato, a soma dos escopos de cada *feature* juntamente com algumas resoluções técnicas.

O escopo base da fábrica é bastante primitivo e passível de mudanças, podendo ser simplesmente uma descrição textual de cada *feature* identificada. Quando um maior conhecimento sobre o domínio é adquirido, este modelo pode ser mais aprofundado. O quadro a seguir mostra as *features* inicialmente identificadas para a fábrica ArcadEx.

Feature	Escopo Básico
Fluxo	O jogos ArcadEx são compostos por uma série de telas, que podem mostrar apenas informações ou conter ações do jogo.
Gráficos	Mundo bi-dimensional (2D). As telas de ação mostram o mundo visto de cima. *Heads-Up Displays* (HUDs), como indicadores do estado de saúde ou tempo do jogo também podem ser mostrados.
Entidades	Cada jogador controla um personagem principal. Outros tipos de entidades são: itens e NPCs. Atributos de uma entidade incluem: posição, velocidade, direção e rotação. Animações também são suportadas.
Eventos	Criação e destruição de entidades, detecção de colisão, transição entre telas, mudança do valor do atributo de uma entidade.
Entrada	Teclado, *mouse* e *joystick*
Som	Efeitos de som como reações aos eventos. Músicas de *background* associadas às telas e tocadas em *looping*.
Modelagem Física	Detecção de colisão e algumas forças de atração.
Inteligência Artificial	Conceitos primitivos de IA são esperados.

Tabela 1 – Escopo base da ArcadEx

3.1.3. Análise do Domínio

A atividade de análise do domínio pode ser definida como um processo no qual as informações utilizadas no desenvolvimento de *software* são identificadas, capturadas e organizadas com a finalidade de torná-las reusáveis no desenvolvimento de novos sistemas [Prieto-Diaz 1990]

No contexto da fábrica ArcadEx, foi proposto um processo pragmático de análise de domínio, englobando as seguintes atividades: definição das

funcionalidades a serem analisadas, seleção de amostras de produtos do domínio, execução da análise, identificação de similaridades e pontos de variabilidade encontrados a partir da atividade de *feature modeling*.

Dentro do escopo deste trabalho, serão mostradas apenas as similaridades e os pontos de variabilidades encontrados, os quais são modelados segundo técnicas especificadas por [Riebisch 2003]. O *feature model* da ArcadEx, dividido em diagramas menores, será mostrado abaixo. Os diagramas foram gerados utilizando-se a DSL visual FeatureModelDSL [Furtado 2008].

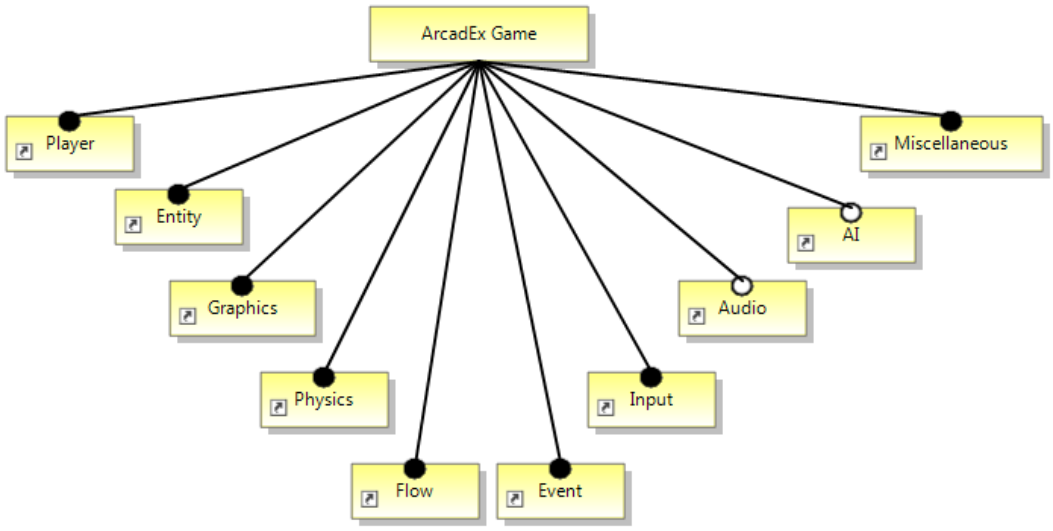


Figura 4 – *Feature model* do jogo ArcadEx

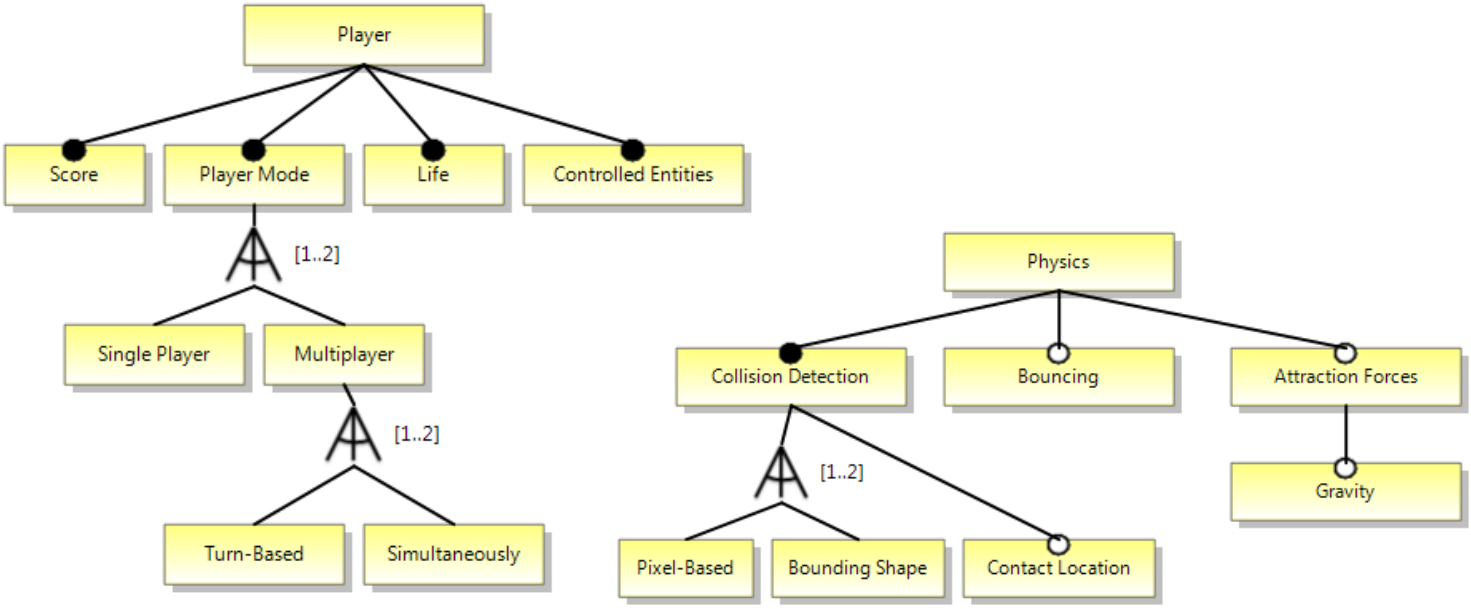
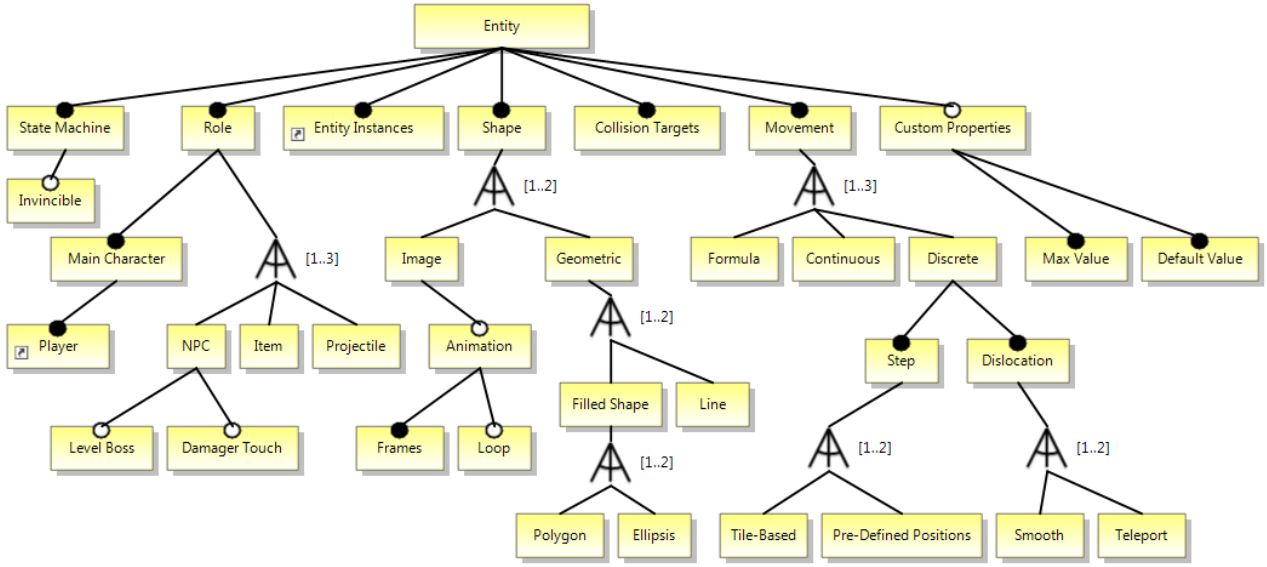


Figura 5 – *Feature model* do jogador e modelagem física



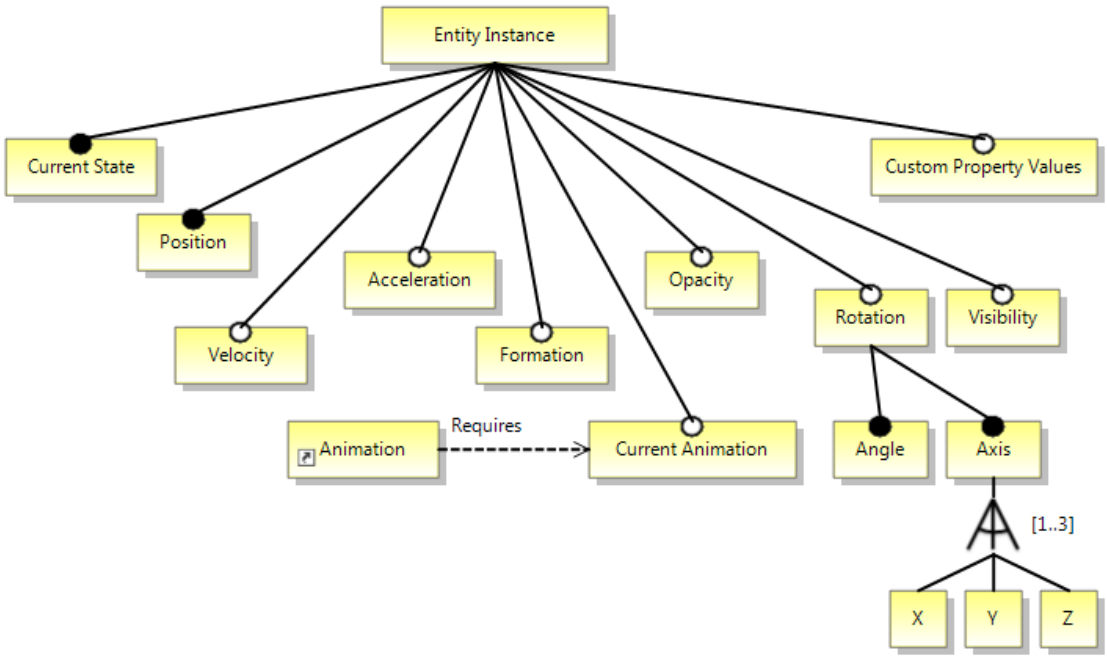


Figura 6 – *Feature model* de uma entidade e de uma instância da entidade

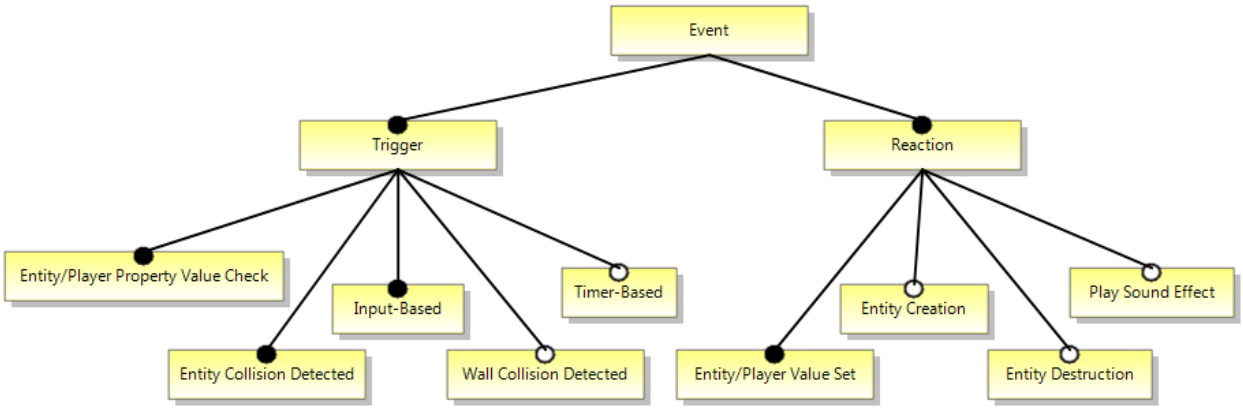


Figura 7 – *Feature model* de um evento

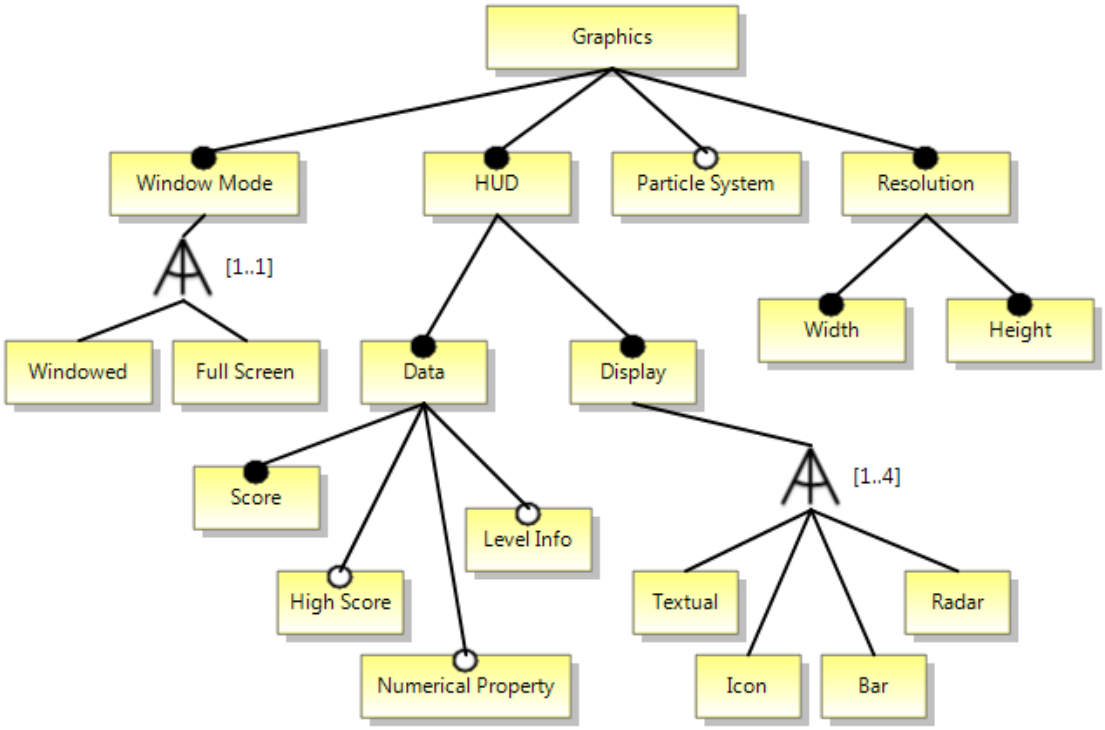


Figura 8 – *Feature model* dos gráficos

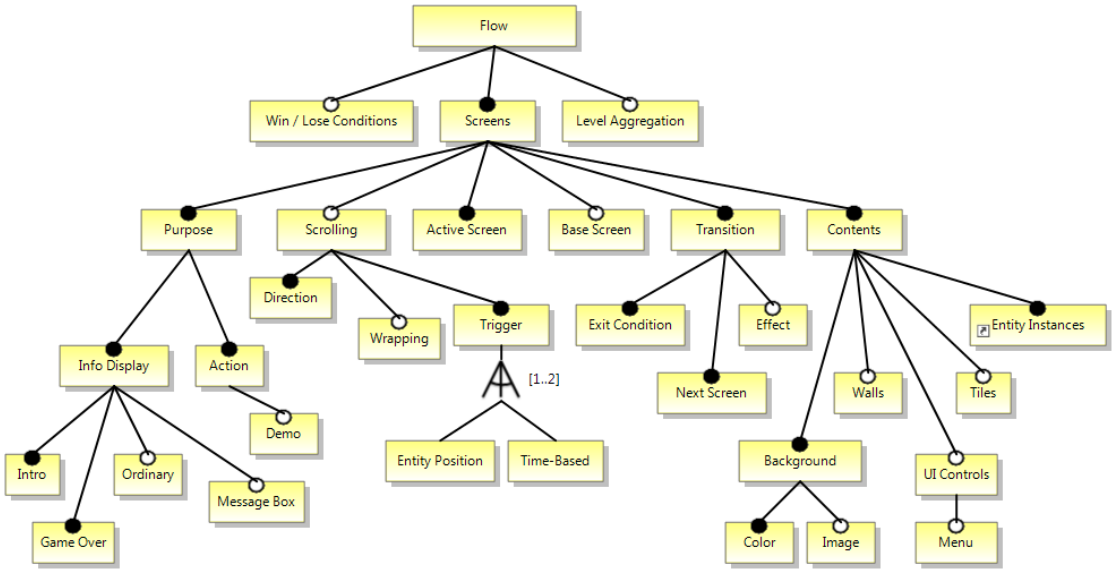


Figura 9 – *Feature model* do fluxo de jogo

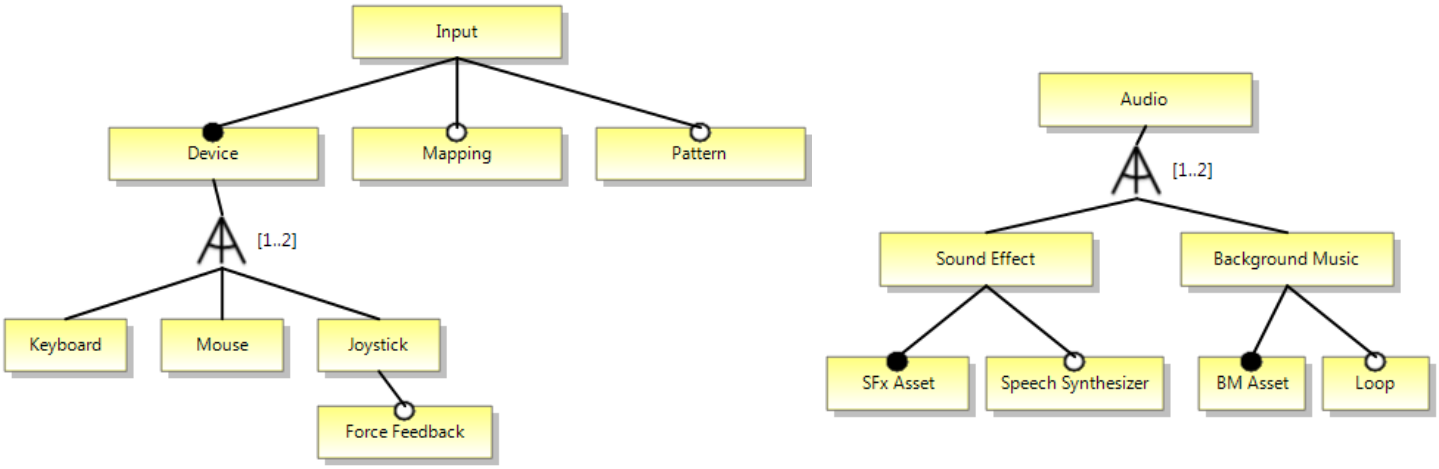


Figura 10 – *Feature model* dos dispositivos de entrada e do áudio

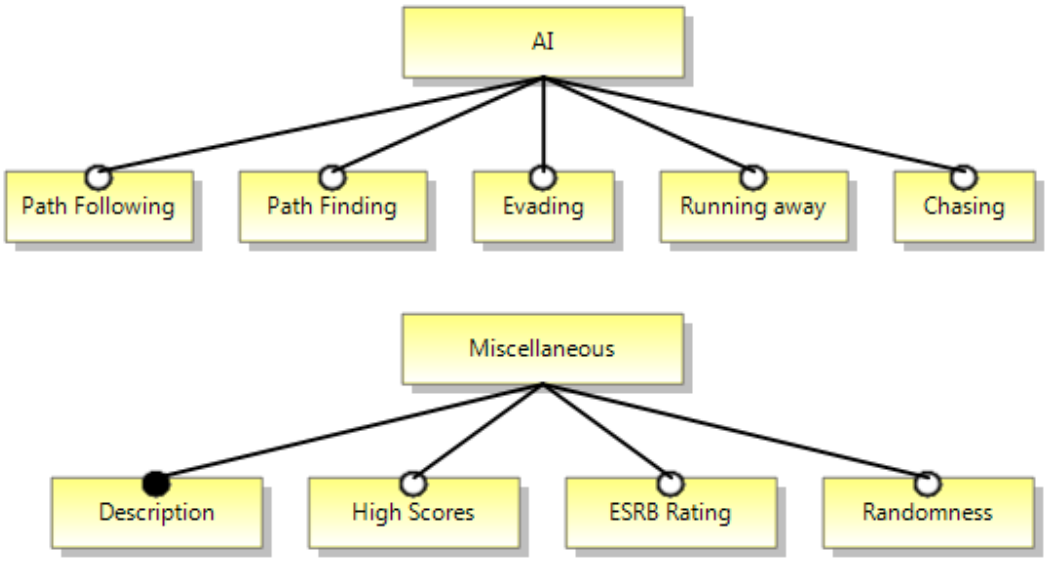


Figura 11 – *Feature model* da IA e aspectos variados

Após a atividade de análise do domínio, o primeiro passo para a construção dos *assets* (bens) da fábrica é encapsular as *features* comuns em bens reusáveis, como componentes ou APIs. No contexto de uma fábrica de games, aconselha-se a utilização de *game engines* como o framework do domínio da fábrica [Furtado & Santos, 2006]. Esta abordagem deve garantir que o *engine* final seja passível de *framework completion*, ou seja, o *engine*

disponibilizará uma interface que é expressiva e concisa o suficiente de forma que o código que o consome possa ser gerado por uma ou mais DSLs.

Para o framework do domínio da ArcadEx, foi escolhido o *game engine* FlatRedBall [FlatRedBall] voltado para a plataforma de jogos XNA, da Microsoft [XNA]. Tal *engine* mostrou-se como um potencial candidato para cobrir as *features* analisadas, com uma baixa curva de aprendizado. O trabalho de pesquisa atual do SharpLudus realizou a criação de uma camada de adaptação em cima do *game engine* com o intuito de facilitar o processo de *framework completion*. Tal camada foi denominada *ArcadEngine*, e sua estrutura será explicada na próxima seção.

4. Validação do Game Engine FlatRedBall

Esta etapa do trabalho consiste na validação do *game engine* escolhido através da implementação de jogos contendo diferentes conjuntos das *features* estudadas na etapa de análise do domínio. Como citado no capítulo anterior, durante o desenvolvimento foram utilizadas as seguintes ferramentas:

- FlatRedBall *game engine*;
- Microsoft XNA 3.0;
- Microsoft Visual Studio 2008;
- Linguagem de programação C# para a plataforma .NET.

Para a implementação, foram escolhidos alguns dos jogos *arcade* mais conhecidos do público em geral, como *Space Invaders,* e também jogos populares disponíveis em celulares, como o *Rapid Roll* Tais jogos foram escolhidos por conterem um grande número das *features* levantadas durante a etapa de análise do domínio.

Como citado na seção anterior, foi utilizado também a camada de abstração *ArcadEngine*, até então desenvolvida como parte do projeto SharpLudus.. Esta camada contém classes que modelam o comportamento das telas, das entidades do jogo e gerenciam os efeitos de som e a transição entre as telas, utilizando as funcionalidades disponibilizadas pelo *game engine*. Abaixo, segue o diagrama de classes da *ArcadEngine*.

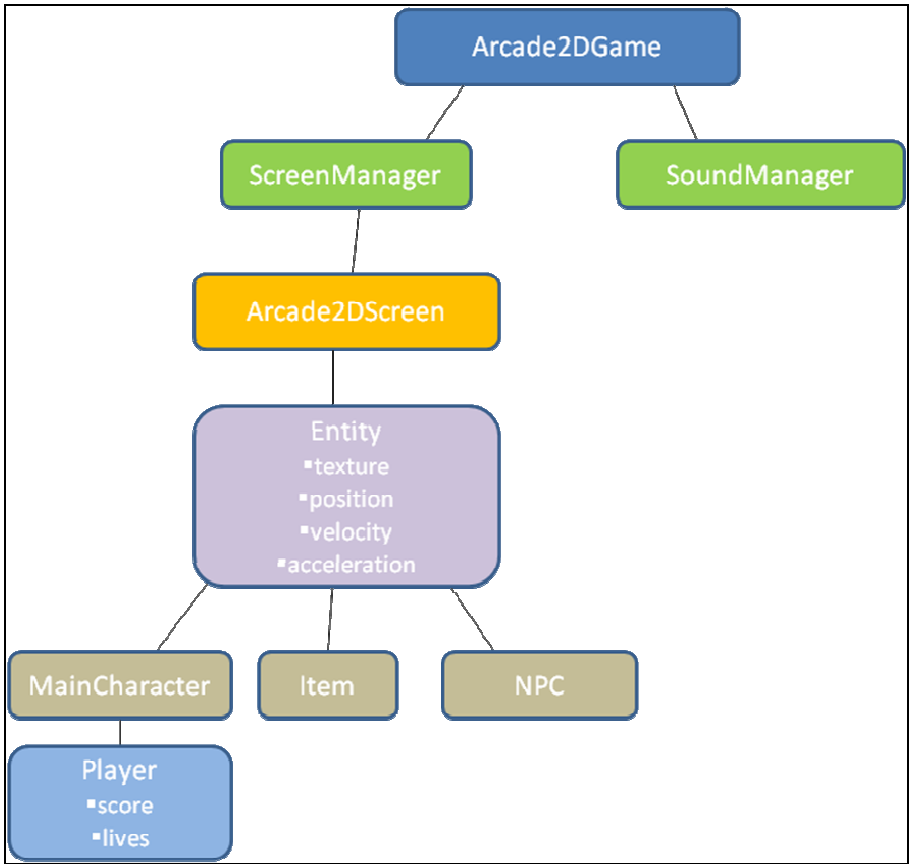


Figura 12 – Organização de um jogo na *ArcadEngine*

A classe *Arcade2DGame* representa um jogo *arcade,* herda de *Microsoft.Xna.Framework.Game* e contém uma classe *ScreenManager* e uma classe *SoundManager*, responsáveis pelo gerenciamento da transição das telas e dos efeitos sonoros, respectivamente. Uma classe *ScreenManager* contém uma referência para a primeira tela do jogo, que referencia a próxima tela a partir dela, e assim sucessivamente. As telas do jogo são do tipo *Arcade2DScreen* e podem conter várias entidades do tipo *MainCharacter*, representando o jogador principal, *Item*, representando entidades que podem ser coletadas pelo jogador, ou *NPC*, que representam os personagens não jogáveis, como os inimigos. As entidades herdam da classe *FlatRedBall.Sprite*. Cada classe do tipo *MainCharacter* possui uma referência a um objeto do tipo *Player*, que contém, entre outras coisas, a pontuação obtida pelo jogador ao longo do jogo.

4.1. Jogos Desenvolvidos

Os jogos criados no contexto deste trabalho utilizam as classes definidas pela ArcadEngine. O projeto de pesquisa SharpLudus já havia desenvolvido previamente o jogo MoneyHunt para validar algumas features do feature model. Alguns dos jogos desenvolvidos neste trabalho basearam-se no MoneyHunt como ponto de partida.

Esta seção contém uma breve descrição dos jogos desenvolvidos e das *features* que foram implementadas em cada um deles, bem como possíveis pontos de melhoria para a camada *ArcadEngine*. O Apêndice A, no final deste trabalho, contém um relatório que lista todas as features do domínio e explica quais foram implementadas em cada jogo.

4.1.1. Space Invaders

Jogo bastante popular lançado na década de 70, onde o jogador controla os movimentos de um canhão que se movimenta horizontalmente na parte inferior da tela. Da parte superior, *aliens* vão em direção ao canhão. O objetivo do jogo é eliminar o maior número de *aliens* possíveis sem ser atingido. Para isso o jogador dispõe de balas infinitas e unidades de energia que aparecem esporadicamente durante o jogo para aumentar a “saúde” do jogador. O jogo termina quando uma pontuação máxima pré-definida é alcançada. A seguir, uma imagem da tela principal do jogo:

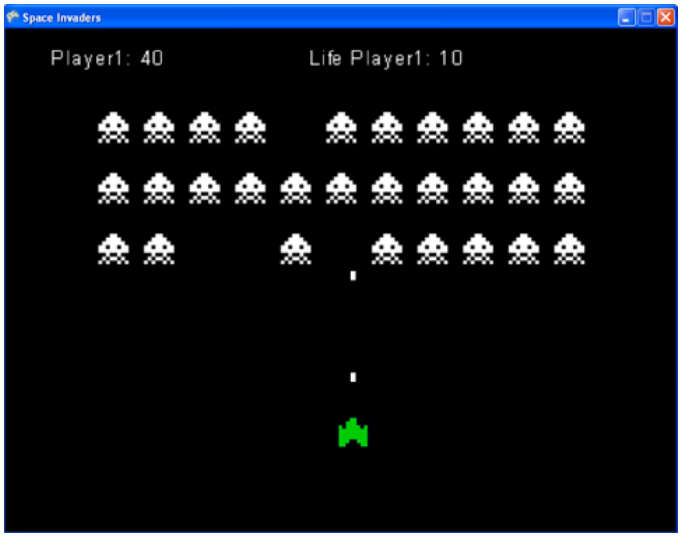


Figura 13 – Telas do jogo Space Invaders

Features Implementadas

Player

Representado pelo canhão.

Modelagem Física

Colisões

- do canhão com os *aliens*
- da bala com os *aliens*
- do canhão com os itens de HealthPowerUp
- do canhão com as paredes laterais

No momento da criação do jogo utilizando a ArcadEx, o desenvolvedor poderá configurar através da DSL visual qual o tipo de colisão desejado entre as instâncias das entidades que estão sendo modeladas. O código a seguir é um exemplo de código que poderia ser gerado pela DSL. Esse código estaria no construtor da classe que representa a entidade e é responsável por definir um circulo de colisão para a mesma.

```
Circle c = new Circle();
c.Radius = this.Texture.Width;
this.SetCollision(c);
```

Entidades

Canhão

- do tipo *MainCharacter*
- possui dois estados: atingido e normal
- vetor de duas dimensões para indicar posição
- movimento contínuo, movimentando-se horizontalmente.
- textura

Alien

- do tipo *NPC*
    - *DamagerTouch:* eventos de colisão determinam a diminuição do nível de "saúde" do jogador
- componente vertical da velocidade, com o sentido de cima para baixo
- vetor de duas dimensões para indicar posição
- movimento contínuo, se movimentando verticalmente
- textura

HealthPowerUp

- do tipo *Item*
- sem movimento
- textura

Bala

- do tipo *Item*
- movimento contínuo, vertical, de baixo para cima
- textura

A DSL visual da ArcadEx deve permitir configurar os atributos textura, posição, velocidade e a aceleração para as entidades de um jogo, bem como definir uma máquina de estados para as mesmas, caso desejado. Os códigos seguintes constituem um exemplo que poderia ser gerado pela DSL. O primeiro representa o construtor da classe do canhão, no qual estão sendo definidos os atributos de posição e o estado inicial.

```
public abstract class Ship : MainCharacter
    {
        public Ship(Texture2D texture, Player player, string name)
            : base(texture, player)
        {
            this.Name = name;
            this.Position = new Vector3(-10, -10, 0);
            this.SetState(Ship.State.Normal);
        }

    }
```

Este outro trecho de código mostra a definição dos estados da entidade canhão

```
public enum State { Normal, Stunned };
private State currentState;

public void SetState(State newState)
{
      if (newState == State.Stunned)
      {
            //Jogador fica impossibilitado de se
            //mover durante alguns segundoos
      }

       else if (newState == State.Normal)
       {
            //Jogador volta ao estado normal.
       }
}
```

Eventos

Triggers/Reaction

- a colisão da bala com o *alien* aumenta a pontuação do jogador
- a colisão do *alien* com o canhão diminui a saúde do jogador

- a colisão do *alien* com o *HealthPowerUp* aumentam a saúde do jogador
- os *aliens* somem da tela ao alcançar a margem inferior
- efeitos sonoros são executados na colisão da bala com o *alien*

Gráficos

- modo de exibição *Windowed*
- resolução 800x600 *pixels*
- *HeadsUp Displays* indicam *score* e estado de saúde.

Fluxo

Telas:

- Início
- Instruções
- Tela Principal
    - contém toda a ação e as entidades do jogo
    - não apresenta *Scrolling* e seus limites são definidos por paredes
- *Game Over*
    - saúde do jogador igual a zero
- Vitória do Jogador.
    - condição de vitória baseada no *score*

O código seguinte configura as paredes da tela principal e pode ser um exemplo de código a ser gerado pela DSL visual da fábrica:

```
PositionedObjectList<AxisAlignedRectangle> mBoardWalls;

    void CreateWalls() {
        mBoardWalls = new PositionedObjectList<AxisAlignedRectangle>();

        AxisAlignedRectangle topWall = ShapeManager.AddAxisAlignedRectangle();
        topWall.ScaleX = 100;
        topWall.ScaleY = 3;
        topWall.Y = SpriteManager.Camera.RelativeYEdgeAt(0) + 2;
        mBoardWalls.Add(topWall);

        AxisAlignedRectangle bottomWall = ShapeManager.AddAxisAlignedRectangle();
        bottomWall.ScaleX = 100;
        bottomWall.ScaleY = 3;
        bottomWall.Y = -SpriteManager.Camera.RelativeYEdgeAt(0) - 2;
        mBoardWalls.Add(bottomWall);

        AxisAlignedRectangle leftWall = ShapeManager.AddAxisAlignedRectangle();
        leftWall.ScaleY = 100;
        leftWall.ScaleX = 3;
        leftWall.X = -SpriteManager.Camera.RelativeXEdgeAt(0) - 2;
        mBoardWalls.Add(leftWall);

        AxisAlignedRectangle rightWall = ShapeManager.AddAxisAlignedRectangle();
        rightWall.ScaleY = 100;
        rightWall.ScaleX = 3;
        rightWall.X = SpriteManager.Camera.RelativeXEdgeAt(0) + 2;
        mBoardWalls.Add(rightWall);

        foreach (AxisAlignedRectangle rec in mBoardWalls) {
            rec.Visible = false;
            mAxisAlignedRectangles.Add(rec);
        }
    }
```

Entrada de Dados

A classe *SpaceInvadersGame* do tipo *Arcade2DGame* contém um método responsável por mapear as teclas do teclado em botões do *joystick* do Xbox 360.

A DSL da *ArcadEngine* deve permitir definir qual tipo de entrada será utilizado e configurar a associação entre as opções existentes, caso desejado. O trecho de código seguinte mostra como seria o código gerado pela DSL. Ele é responsável por mapear as teclas do teclado nos botões do *joystick*.

```
private static void SetInputMappings() {
     if (!InputManager.Xbox360GamePads[0].IsConnected) {
          KeyboardButtonMap map = new KeyboardButtonMap();
          map.DPadLeft = Keys.A;
          map.DPadUp = Keys.W;
          map.DPadDown = Keys.S;
          map.DPadRight = Keys.D;
          map.Start = Keys.Enter;
          map.Back = Keys.Escape;
          InputManager.Xbox360GamePads[0].ButtonMap = map;
     }
}
```

Áudio

Cada tela possui uma música de fundo que é executada em *looping*. Efeitos sonoros são definidos carregando-se arquivos Wav a partir da classe *SoundManager*.

4.1.2. Rapid Roll

Rapid Roll é baseado em um jogo presente nos modelos mais simples dos celulares Nokia®, como o Nokia 1100 e o Nokia 2255 [Nokia]. Neste jogo, o jogador comanda uma bola que não deve tocar os espinhos que aparecem na tela. Para isso, ele deve pular nos blocos que surgem da parte inferior da tela e se movimentam verticalmente em direção à parte superior. Ao final do jogo, uma tela de *game over* é mostrada com a pontuação máxima alcançada pelo jogador. Abaixo segue uma imagem da tela principal jogo.

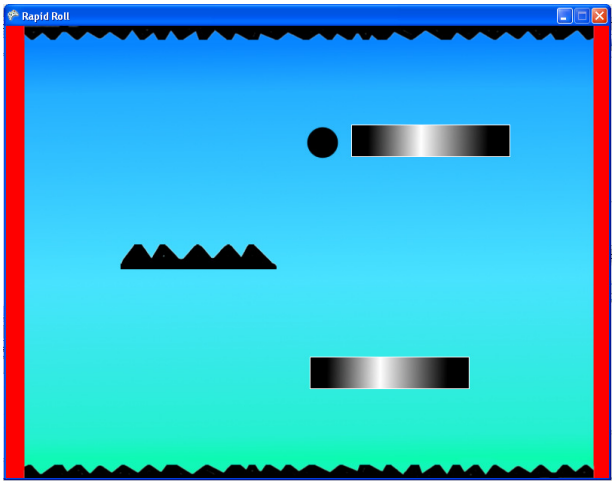


Figura 14 – Tela do jogo Rapid Roll

Features Implementadas

Player

- representado pela imagem uma bola
- atributo para indicar o *score*
- inicia o jogo com três vidas

Modelagem Física

A bola possui aceleração descendente que simula a força da gravidade. Existe a colisão da bola com os blocos ascendentes e com os espinhos, do tipo *CollideAgainstMove*, onde o quadrado acompanha o movimento do bloco durante a colisão. Todas as colisões são do tipo *BoundingShape*.

As features de gravidade e colisão do tipo *CollideAgainstMove* não haviam sido implementadas anteriormente no jogo *MoneyHunt*. Esse tipo de colisão não é previsto pelo fábrica, constituindo uma extensão do escopo original.

Entidades

- Bola
  - do tipo *MainCharacter*

    - aceleração e velocidade descendentes
    - possui dois estados
        - parado: quando está em cima do bloco
        - movimento: quando está em queda livre
    - texturas
    - movimento contínuo
- Bloco e Espinho
    - do tipo *Item*
    - velocidade ascendente
    - texturas
    - movimento contínuo

Eventos

Triggers/Reaction

- colisão da bola com o espinho ocasiona perda de vida e efeitos sonoros
- remoção dos blocos e espinhos ao alcançarem o topo da tela

Gráficos

O jogo consegue armazenar o *score* mais alto alcançado por um jogador. O *score* obtido pelo jogado e o *score* máximo do jogo são mostrados na tela de *Game Over* e se o jogador atingiu um novo recorde, o *score* máximo é atualizado.

Fluxo

Telas

- Tela principal
    - contém toda a ação e as entidades do jogo
    - geração randômica dos blocos e espinhos
- Introdução
- Instruções

- *Game Over*

Entrada de Dados e Áudio

Utilizam o mesmo esquema adotado no jogo *Space Invaders*.

4.1.3. HitMario

Hit Mario é um jogo criado utilizando imagens do jogo Super Mario World [Nintendo] onde o jogado tem que acertar a imagem do Mario, que fica mudando de posição no decorrer do jogo. Ao clicar na imagem incorreta, o jogador perde o jogo. A figura seguinte mostra a tela principal do jogo.

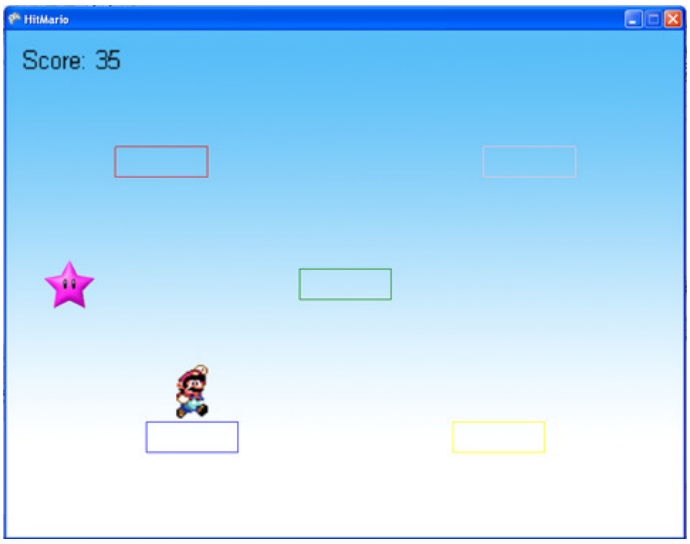


Figura 15 – Tela do jogo HitMario

Features Implementadas

Entidades

- Mario
  - do tipo *Item*
  - movimento discreto baseado em tele transporte (posições pré-definidas)

    - máquina de Estados
        - *normal*, representando o Mario que deve ser clicado
        - *badMario*, representando o Mario que faz o jogador perder
- Estrela
    - do tipo *Item*
    - animação
    - visibilidade, ficando invisível durante um intervalo de tempo pré-definido

Eventos

Triggers/Reaction

- ao clicar no Mario correto, o score aumenta
- ao clicar no Mario incorreto, o jogo acaba
- ao clicar na Estrela, a velocidade do jogo diminui

Gráficos

O jogo contém *HUDs* para indicar o *score* do jogador e informar se o jogo está no modo Star, onde a velocidade do Mario é diminuída.

Fluxo

Telas

- Tela principal
    - contém toda a ação e as entidades do jogo
- Introdução
- Instruções
    - seleção do nível de dificuldade
- *Game Over*

Entrada de Dados

Utiliza o *mouse* para saber se o Mario foi clicado ou não.

Áudio

Utiliza o mesmo esquema adotado nos demais jogos.

4.2. Sugestões para a *ArcadEngine*

Algumas *features* do domínio apresentaram dificuldades no momento de sua implementação. Porém, alguns desses problemas podem ser resolvidos se realizadas pequenas modificações na camada de abstração *ArcadEngine*.

Entre essas *features,* podemos citar a implementação de *Animation*/*Frames* para as entidades do jogo. A classe *Entity* da *ArcadEngine*, da forma como está implementada atualmente, possui um construtor que aceita apenas texturas. A solução encontrada foi acrescentar um novo construtor para a classe *Entity* que recebe uma *AnimationChain* (classe que representa uma animação) como parâmetro. Tal exemplo é mostrado no código a seguir:

```
public Entity(AnimationChain animationChain)
        {
            this.SetAnimationChain(animationChain);
            SpriteManager.AddSprite(this);
            this.PixelSize = 0.03f;
            this.Position = Vector3.Zero;
        }
```

Para a *feature* que define a resolução do jogo, foi criado um método que configura o tamanho da janela do jogo na classe *Arcade2DGame* da *ArcadEngine*, como mostra o código a seguir:

```
protected void SetWindowMode(int width, int height, bool isFullScreen)
        {
            this.graphics.PreferredBackBufferWidth = width;
            this.graphics.PreferredBackBufferHeight = height;
        }
```

Este outro método mostra como configurar o jogo para ser exibido em modo FullScreen:

```
protected void SetFullScreenMode()
        {
            this.graphics.IsFullScreen = true;
        }
```

5. Conclusão

Após a realização deste trabalho, foi possível concluir que tanto o *game engine* utilizado quanto a camada de adaptação implementada pela ArcadEx facilitaram o processo de desenvolvimento dos jogos. O game *engine* se mostrou capaz de suportar grande parte das *features* existentes no domínio de *games arcade* 2D, requerendo um baixo esforço de implementação. Este trabalho também trouxe contribuições para a FeatureModelDSL e a Confeaturator, ferramentas que estão sendo desenvolvidas no contexto do projeto SharpLudus.

Entre as *features* encontradas para o domínio em questão, os jogos que foram desenvolvidos demonstram que grande parte das funcionalidades já está sendo suportada. Porém, ainda existem *features* que não foram implementadas e validadas pelos jogos desenvolvidos, por questões de recursos limitados para este trabalho. Entre elas podemos citar:

- *Custom Properties* para as entidades, como quantidade de combustível, de munição, etc.;
- Background tiles;
- Elementos de IA, como perseguição de um entidade, *path finding*, path following, etc.;
- *Scrolling* das telas;
- *Input pattern,* onde um comportamento da entidade, como aceleração ou velocidade, é mapeado em um dispositivo de entrada de dados.

Durante a realização deste trabalho de graduação, surgiram algumas dificuldades. A primeira delas foi trabalhar com um projeto que ainda não está totalmente finalizado, como é o caso do SharpLudus e da fábrica ArcadEx. Podemos citar também a falta de experiência com o *game engine* utilizado e com a linguagem C#, utilizada na implementação dos jogos. Outra dificuldade foi lidar com ferramentas bastante novas, como o XNA 3.0 e o Microsoft Visual

Studio 2008, e com o fato de a versão atual do FlatRedBall não suportar oficialmente as últimas versões do XNA e do VS.

Entre as possíveis propostas de continuidade em trabalhos futuros, existe espaço para contribuir ainda mais com a pesquisa, implementando-se as features citadas anteriormente que ainda não foram validadas. Outra proposta interessante seria a realização de um *refactoring* no código dos jogos existentes a fim de encontrar alternativas para implementar mais facilmente as *features* e simplificar o código que pode ser gerado por uma DSL. Existe também a possibilidade de se utilizar a API de *Storage* do *framework* XNA para armazenar o *score* nos jogos.

Outra proposta de trabalho futuro encontrada é reflexão sobre o nível de acoplamento da Fábrica ArcadEx, a ArcadEngine e o FlatRedBall, no sentido de analisar a utilização de outra *game engine* ou até mesmo outra plataforma que não XNA. Da maneira atual, isso não é suportado, pois a camada de abstração ArcadEx é totalmente dependente do *game engine* FlatRedBall

Finalmente, este trabalho proporcionou a oportunidade de estudar conceitos bastante interessantes e extremamente importantes, como as fábricas de *software* e as DSLs, que estão ganhando cada vez mais importância no processo de desenvolvimento de software atual. O desenvolvimento dos jogos mostrou-se uma atividade que pode ser, ao mesmo tempo, divertida e desafiadora.

6. Apêndice A

O relatório abaixo foi gerado pelo *Confeaturator* e pela *FeatureModelDSL*, uma DSL visual do projeto SharpLudus para gerar *feature models*. Este relatório resume quais *features* da fábrica ArcadEx foram validadas pelos jogos desenvolvidos.

Configuration Report: HitMario, MoneyHunt, RapidScroll, SpaceInvaders

Summary

	Selected	Unselected	Total
Root / Mandatory	60	3	63
Optional / alternative options	58	36	94
Total	118	39	**157**

Details

Features	HitMario	MoneyHunt	RapidScroll	SpaceInvaders	Total
ArcadEx Game	•	•	•	•	4
Player	•	•	•	•	4
Score	•	•	•	•	4
Player Mode	•	•	•	•	4
Alternative [1..2]					
Single Player	•		•	•	3
Multiplayer		•			1
Alternative [1..2]					
Turn-					0

Based					
Simultaneously		•			1
Life	•	•	•	•	4
Controlled Entities	•	•	•	•	4
Entity	•	•	•	•	4
State Machine	•	•	•	•	4
Role	•	•	•	•	4
Main Character	•	•	•	•	4
Alternative [1..3]					
NPC		•	•	•	3
Level Boss					0
Damager Touch			•	•	2
Item	•	•		•	3
Projectile				•	1
Entity Instance	•	•	•	•	4
Current State	•	•	•	•	4
Position	•	•	•	•	4
Acceleration			•		1
Velocity		•	•	•	3
Formation				•	1
Opacity	•	•		•	3
Current Animation	•	•			2
Rotation		•			1
Angle	•	•	•	•	4
Axis	•	•	•	•	4

Alternative [1..3]					
X		•			1
Y		•			1
Z		•			1
Custom Property Values					0
Visibility	•				1
Shape	•	•	•	•	4
Alternative [1..2]					
Image	•	•	•	•	4
Animation	•	•			2
Frames	•	•	•	•	4
Loop	•	•	•	•	4
Geometric	•		•		2
Alternative [1..2]					
Filled Shape	•		•		2
Alternative [1..2]					
Polygon	•		•		2
Ellipsis					0
Line					0
Collision Targets	•	•	•	•	4
Movement	•	•	•	•	4
Alternative [1..3]					
Continuous		•	•	•	3
Discrete	•				1

Step	•	•	•	•	4
Alternative [1..2]					
Tile-Based					0
Pre-Defined Positions	•				1
Dislocation	•	•	•	•	4
Alternative [1..2]					
Smooth					0
Teleport	•				1
Formula			•		1
Custom Properties					0
Max Value					0
Default Value					0
Graphics	•	•	•	•	4
HUD	•	•	•	•	4
Data	•	•	•	•	4
Score	•	•	•	•	4
High Score			•		1
Level Info					0
Numerical Property					0
Display	•	•	•	•	4
Alternative [1..4]					
Textual	•	•	•	•	4
Icon					0
Bar					0

Radar					0
Window Mode	•	•	•	•	4
Alternative [1..1]					
Windowed		•	•	•	3
Full Screen	•				1
Resolution	•	•	•	•	4
Width	•	•	•	•	4
Height	•	•	•	•	4
Particle System					0
Physics	•	•	•	•	4
Attraction Forces			•		1
Gravity			•		1
Bouncing		•	•		2
Collision Detection		•	•	•	3
Alternative [1..2]					
Pixel-Based					0
Bounding Shape	•	•	•	•	4
Contact Location					0
Flow	•	•	•	•	4
Win / Lose Conditions	•	•	•	•	4
Screens	•	•	•	•	4
Purpose	•	•	•	•	4
Info Display	•	•	•	•	4
Intro	•	•	•	•	4
Game Over	•	•	•	•	4
Ordinary	•	•	•	•	4

Message Box					0
Action	•	•	•	•	4
Demo					0
Scrolling					0
Direction					0
Trigger	•	•	•	•	4
Alternative [1..2]					
Entity Position					0
Time-Based					0
Wrapping					0
Active Screen	•	•	•	•	4
Base Screen					0
Transition	•	•	•	•	4
Exit Condition	•	•	•	•	4
Next Screen	•	•	•	•	4
Effect					0
Contents	•	•	•	•	4
Entity Instances	•	•	•	•	4
Background	•	•	•	•	4
Color	•	•	•	•	4
Image	•	•	•	•	4
Walls		•	•	•	3
UI Controls					0
Menu					0
Tiles					0

Level Aggregation					0
Event	•	•	•	•	4
Trigger	•	•	•	•	4
Entity/Player Property Value Check	•	•	•	•	4
Input-Based	•	•	•	•	4
Entity Collision Detected	•	•	•	•	4
Wall Collision Detected		•	•	•	3
Timer-Based	•	•	•	•	4
Reaction	•	•	•	•	4
Entity Destruction		•	•	•	3
Entity Creation		•	•	•	3
Play Sound Effect	•	•	•	•	4
Entity/Player Value Set	•	•	•	•	4
Input	•	•	•	•	4
Device	•	•	•	•	4
Alternative [1..2]					
Keyboard		•	•	•	3
Mouse	•				1
Joystick		•	•	•	3
Force Feedback		•	•	•	3
Pattern		•		•	2
Mapping		•		•	2
Audio	•	•		•	3

Alternative [1..2]					
Sound Effect	•	•	•	•	4
SFx Asset	•	•	•	•	4
Speech Synthesizer					0
Background Music	•	•	•	•	4
Loop	•	•	•	•	4
BM Asset	•	•	•	•	4
AI					0
Path Following					0
Path Finding					0
Evading					0
Running away					0
Chasing					0
Miscellaneous	•	•	•	•	4
Description	•	•	•	•	4
High Scores			•		1
ESRB Rating					0
Randomness	•	•		•	3

Generated by **Feature Model DSL** on Dec 01, 2008
Implementation and extended design by: Andre Furtado
Original design by: Gunther Lenz and Christoph Wienands

Referências

[IbisWorld] IbisWorld.*The New American Players: baby boomers and women take on video gaming*. [Online] 2008. [Citado em 25 de agosto de 2008] Disponível em [http://www.ibisworld.com/pressrelease/pressrelease.aspx?prid=133].

[MPAA 2007] MPAA. 2007 *Entertainment Industry Market Statistics*. Motion Picture Association. [Online] 2007. [Citado em 25 de agosto de 2008] Disponível em [http://www.mpaa.org/USEntertainmentIndustryMarketStats.pdf]

[Greenfield 2004] Greenfield, J. *The Case for Software Factories* Microsoft Architect Journal. [Online] 2004. [Citado em 25 de agosto de 2008] Disponível em [http://msdn.microsoft.com/en-us/library/aa480032.aspx].

[Greenfield 2004] Greenfield, J. *Moving to Software Factories*. [Online] 2004. Citado em 21 de outrubro de 2008. Disponível em [http://www.softwarefactories.com/ScreenShots/MS-WP-04.pdf]

[Greenfield 2003] J Greenfield, K Short, S Cook, S Kent. *Assembling Applications with Patterns, Models, Frameworks and Tools*. ACM OOPSLA 2003.

[Deursen 1999] Arie van Deursen, Paul Klint, Joost Visser. *Domain-Specific Languages: An Annotated Bibliography*. [Online] 1999. [Citado em 25 de agosto de 2008]. Disponível em [http://homepages.cwi.nl/~arie/papers/dslbib/].

[FlatRedBall] FlatRedBall XNA & MDX Game Engine. [Online] 2008. [Citado em 25 de Agosto de 2008] Disponível em [http://www.flatredball.com/frb/].

[XNA] Microsoft. XNA.com. [Online] 2008. [Citado em 13 de Novembro de 2008]. Disponível em [http://www.xna.com/].

[Furtado 2006] Furtado, A. *SharpLudus: Improving Game Development Experience Through Software Factories and Domain-Specific Languages,* 2006. Dissertação de mestrado, Centro de Informatica – UFPE.

[Furtado & Santos 2006] Furtado, A. W. B.; Santos, A. L. M. *Software Factories Relevance to Digital Games: A Practical Discussion*, 3rd Brazilian Symposium on Computer Games and Digital Entertainment. 2006

[Abragames 2008] Abragames. *Indústria Brasileira de Jogos Eletrônicos: Um mapeamento do crescimento do setor nos últimos 4 anos*. [Online] 2008. Citado em 21 de outubro de 2008. Disponível em [http://www.abragames.org/docs/Abragames-Pesquisa2008.pdf]

[Lopes 2007] Lopes, J. *Games no Brasil – Criatividade e Desafios. 2007*. [Online] 2007. Citado em 21 de outubro de 2008. Disponível em [http://diariodonordeste.globo.com/materia.asp?codigo=468533]

[VSTS] *Visual Studio Team System 2008 Team Suit*. Microsoft Developer Network. [Online] 2008, Citado em 3 de novembro de 2008. Disponível em [http://msdn.microsoft.com/en-us/vsts2008/default.aspx]

[Udai@ Technology] *Microsoft Patterns & Practices – Web Client Software Factory.* [Online] 2007. Citado em 3 de novembro de 2008. Disponível em [http://bestofcyber.wordpress.com/2007/02/01/microsoft-patterns-practices-web-client-software-factory/]

[Cook 2007] Cook, S.; Jones, G.; Kent, S.; Wills, A. C., *Domain-Specific Development with Visual Studio DSL Tools*, Addison-Wesley Professional, 2007.

[Fowler 2005] Fowler, M. *Language Workbenches: The Killer-App for Domain Specific Languages?* [Online] 2007. Citado em 4 de novembro de 2008. Disponível em [www.martinfowler.com/articles/languageWorkbench.html]

[Fowler 2007] Fowler, M. *Domain Specific Languages.* [Online] 2007. Citado em 4 de novembro de 2008. Disponível em [http://martinfowler.com/dslwip/Intro.html]

[Web 2.0 Wikipedia] *Web 2.0*. Wikipédia, a enciclopédia livre. [Online] 2007. Citado em 5 de novembro de 2008. Disponível em [http://pt.wikipedia.org/wiki/Web_2.0]

[Prieto-Diaz 1990] Prieto-Diaz, R. *Domain Analysis: An Introduction*. ACM SIGSOFT Software Engineering Notes, Vol. 15, No. 02, pp. 47-54, April, 1990.

[Riebisch 2003] Riebish, M. *Towards a More Precise Definition of Feature Models. Modelling Variability for Object-Oriented Product Lines*, pp. 64-76, BooksOnDemand Publ. Co. Norderstedt, 2003.

[Nokia] *Nokia*. [Online] 2008. Citado em 15 de novembro de 2008. Disponível em [http://www.nokia.com.br/]

[Furtado 2008] Furtado, A. W. B. *Feature Model DSL*. [Online] 2008. Citada em 24 de novembro de 2008. Disponível em [www.codeplex.com/FeatureModelDSL]

[Nintendo] Nintendo. [Online] 2008. Citado em 24 de novembro de 2008. Disponível em [http://www.nintendo.com/]

Assinaturas

Este Trabalho de Graduação é resultado dos esforços da aluna Laís de Mendonça Neves, orientada pelo professor André Luís de Medeiros Santos, com o título "*Validação e Adaptação do FlatRedBall Game Engine para Fábrica de Jogos Arcade 2D*". Todos abaixo estão de acordo com o conteúdo deste documento e os resultados deste Trabalho de Graduação

Orientador

_____

Prof. Ph.D. André Luís de Medeiros Santos

Aluna

_____

Laís de Mendonça Neves