UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
CIÊNCIAS DA COMPUTAÇÃO

Extensão da fábrica de jogos SharpLudus para desenvolvimento em J2ME

Trabalho de Graduação

Aluno: Geraldo Fernandes da Silva Filho
Orientador: André Luis Medeiros dos Santos

Assinaturas

Este Trabalho de Graduação é resultado dos esforços do aluno Geraldo Fernandes da Silva Filho, sob a orientação do professor André Luis Medeiros dos Santos, conduzido no Centro de Informática da Universidade Federal de Pernambuco. Todos abaixo estão de acordo com o conteúdo deste documento e os resultados deste Trabalho de Graduação.

Geraldo Fernandes da Silva Filho

André Luis Medeiros dos Santos

Agradecimentos

Agradeço primeiramente a minha família. A Geraldo Fernandes (o pai), que depois de três anos e meio, achou que eu estava apto a dirigir o carro. A minha mãe (Anunciada) e meus irmãos (Raquel e Gabriel) por serem bastante pacientes comigo.

Ao meu orientador, André Santos, por ter confiado em mim para a realização desse trabalho. Esteve sempre tranqüilo e disposto a me ajudar em todas as horas possíveis.

Ao criador do SharpLudus, André Furtado, que deu um suporte fundamental para o desenvolvimento deste trabalho, discutindo soluções e sugerindo melhorias.

A interface humana com o Google, Letícia, que conseguiu informações importantes para o enriquecimento deste trabalho e me ajudou com o Word.

Aos membros do extinto grupo mobili, Carlinhos e BPE, por me ajudarem discutindo problemas referentes à implementação deste trabalho.

A empresa Meantime Mobile Creations, pela compreensão que este trabalho demandou bastante tempo e eu precisava de flexibilidade nos horários de trabalho.

Finalmente a turma de computação 2006.2, por ter tornado a minha passagem pelo CIn algo suportável, com muitas idas para churrascos e barzinhos. Agradeço a todos: Laís, minha irmã de faculdade; Carol, que vai ter cinco filhos; Leo, Zé Carlos, Allan, Renan, Cabelinho, Livar, Jeane, Thiago, Gustavo Veio, Forro, Joabe, Acerola, Aninha, Arthur, Rodrigo, Carina, Augusto, Lira, Martinelli, Thiérry, René, Pablo, Bengt, Tadeu, Marcio e Dudu do Pagode; A galera de engenharia Farley, Vitão, Syl, Milena, Beto, Lauro, Adriano, Adelma, Seba.

Resumo

A industria de jogos é uma das mais vitoriosas e promissoras do mundo de entretenimento digital. Entretanto, não há paradigmas de desenvolvimento de jogos os quais estejam aptos a atender a demanda que vem crescendo em ritmo exponencial. A industrialização desse processo através da construção de fábricas de software como o SharpLudus é uma opção a qual vem ganhando forçar. Todavia, este paradigma não possui tanto força entre os desenvolvedores de jogos para celular, uma vez que para atender as restrições existentes atualmente nos celulares, é necessário ir na contramão da engenharia de software. Este trabalho, portanto, explora a integração entre o desenvolvimento de jogos para celular com o conceito de fábricas de software, as quais são focadas em transformar o atual paradigma artesanal de desenvolvimento em um processo de manufatura.

Palavras-chave: desenvolvimento de jogos, automação, fábricas de software, jogos para celular, SharpLudus.

Índice

Índice de Figuras

Índice de Tabelas

1. Introdução

1.1 Motivação e Objetivos

O mercado de entretenimento está em crescimento. Dentre as mais diversas formas de suprir a demanda desse mercado estão os jogos digitais. Espera-se que, em 2010, o faturamento de jogos para todas as plataformas atinja a inacreditável marca de 51 bilhões de dólares[1]. Com isso, a indústria de jogos tem investido bastante em seus novos produtos, afim de oferecer a seu público-alvo jogos cada vez melhores.

Jogos para dispositivos móveis vêm se mostrando bastante promissor neste mercado. Entretanto, assim como no desenvolvimento de jogos para PC e consoles, altos custos também atingem os jogos para dispositivos móveis.

Esse problema é causado, principalmente, pela falta de automação no processo de desenvolvimento e a necessidade de realizar o *porting* para diversos aparelhos. Uma das possíveis soluções para esse problema está na construção de fábricas de software, focadas em trazer ao universo do desenvolvimento de software conceitos industriais visando automação e produtividade, consequentemente reduzindo os custos de uma linha de produção.

O SharpLudus[2] é uma fábrica de software que gera código que consome um determinado *game engine* em C#. Alterar essa fábrica de software para a geração de código em J2ME é um dos objetivos desse trabalho de graduação.

Apenas gerar o código não é o bastante. Parte da tarefa consiste também em desenvolver um *game engine* a ser consumido pelo código J2ME. Isso envolve um certo grau de dificuldade, pois esse novo *game engine* precisaria ser re-projetado, dado que o *game engine* utilizado pelo SharpLuds é incompatível com os padrões de restrições para dispositivos móveis.

O desenvolvimento da nova *engine* utiliza alguns conceitos da Engenharia de Software. Muitos destes conceitos foram utilizados para adequar a *engine* ao gerador de código. Com isto, um antigo dilema no desenvolvimento de jogos é

reabordado: como ultrapassar as barreira de limitação de performance seguindo os padrões da Engenharia de Software.

1.2 Estrutura do Trabalho

Este trabalho está dividido da seguinte maneira:

O Capítulo 2 descreve um pouco da tecnologia J2ME e mostra alguns dados sobre o mercado de jogos para esta tecnologia.

O Capítulo 3 apresenta uma introdução sobre a fábrica de software SharpLudus, falando da estrutura básica da fábrica e como ele encapsula alguns de seus componentes.

O Capítulo 4 descreve as soluções encontradas para a mudança no SharpLudus para a geração de código em J2ME. Também descreve o funcionamento da *game engine* criada para este trabalho.

O Capítulo 5 apresenta um estudo de caso realizado em cima das mudanças no SharpLudus e da *game engine* criada.

O Capítulo 6, por fim, apresenta conclusões sobre o trabalho e indica possíveis mudanças para trabalhos futuros.

2. Jogos para celular

Não há dúvidas de que o mercado de jogos para celular está em larga expansão. Os números são bastante otimistas. A figura 1 mostra que em 2005, as vendas chegaram a $2,572 bilhões, e em 2010 espera-se alcançar a inacreditável marca de $11,186 bilhões, segundo estudo da Informa Telecoms and Media[3].

Esses números são bastante expressivos, visto que o mercado de jogos para *mobile* corresponde a 7,2% do mercado de jogos. Em 2010, esse número saltará para 21,8%. Jogos para celulares só não terão uma rentabilidade maior do que jogos desenvolvidos para console.

Worldwide Game Revenues, by Sector, 2000, 2005 & 2010 (millions)

	2000	2005	2010
Console hardware	$4,791	$3,894	$5,771
Console software*	$9,451	$13,055	$17,164
Handheld hardware	$1,945	$3,855	$1,715
Handheld software*	$2,872	$4,829	$3,113
PC software*	$5,077	$4,313	$2,955
Interactive TV	$81	$786	$3,037
Broadband	$70	$1,944	$6,352
Mobile	**$65**	**$2,572**	**$11,186**
Total	**$24,352**	**$35,248**	**$51,292**

*Note: *sales and rental*
Source: Informa Telecoms & Media, October 2005

067861 ©2006 eMarketer, Inc. www.eMarketer.com

Figura 1 - Revendas de jogos no mundo.

Este mercado ainda será dominado pela Ásia, tanto em número de vendas, quanto em número de usuários. Estima-se que 60% das vendas dos jogos para celulares acontecerão na Ásia, como podemos ver na Figura 2, liderados pelo Japão e Coréia do Sul.

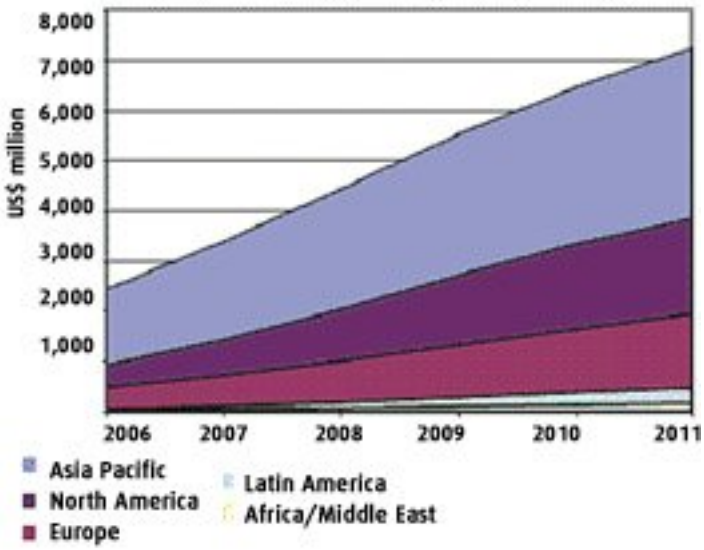


Figura 2 - Crescimento do mercado de jogos mobile no mundo.

Mesmo com números tão expressivos, a indústria de jogos para *mobile* ainda tem um problema de aceitação entre os usuários. Apenas 6,7% dos usuários de celular baixam jogos. Em 2010, espera-se que esse número passe para 15,2% dos usuários.

Outro fator importante é o perfil do jogador. Eles não são "*hardcore users*", isto é, usuários que jogam intensivamente. Por outro lado, jogos para celulares tendem a ser jogos mais casuais. *Hardcore users* preferem outros tipos de dispositivos móveis, como Nintendo DS e PSP.

Entretanto o avanço dos celulares tornará possível o desenvolvimento de jogos mais elaborados, atraindo assim o público de *hardcore gamers*. Segundo o presidente da Gameloft, Michel Guillemot, não apenas o excelente avanço dos aparelhos possibilitará jogos mais realista, mas também a melhoria da rede, que agora permite jogos *multiplayer* em tempo real. Isso trará para o universo dos jogos para celular algo que já ocorre em outras plataformas através da internet: a possibilidade de desafiar pessoas de diferentes regiões do mundo.

Tais jogos *multiplayer* começarão a dar um retorno significativo para as operadoras, com trafego de dados, a partir de 2010. Esses jogos representarão um montante significativo das vendas de jogos para celular, com cerca de 20% das vendas.

2.1 J2ME

Java 2 Platform, *Micro Edition*[4], ou *J2ME*, é uma coleção de APIs de Java para o desenvolvimento de software para dispositivos com restrições, como PDAs, celulares e outros. J2ME foi desenvolvido pela Java Community Process como a JSR 68 (*Java Specification Request*). Entretanto, a evolução da plataforma exigiu que Java ME não fosse mais tratado como uma *Java Specification Request*. Atualmente, existem *Java Specification Request* próprias de J2ME.

J2ME, assim como J2SE e J2EE, foi desenvolvido pela Sun Micro-systems. Entretanto, a Sun apenas define as interfaces, o que não ocorre nas soluções da Sun J2SE e J2EE. As implementações das funções são de responsabilidade dos fabricantes dos aparelhos.

Java ME tornou-se popular para a criação de jogos para celulares devido à facilidade para desenvolver, emular (no PC) e baixar os jogos nos aparelhos. Tal abordagem contrasta com plataforma de desenvolvimento de jogos para dispositivos móveis, como as produzidas pela Nintendo e Sony. Nestas plataformas é necessária a compra de hardware e kits de desenvolvimentos caros.

2.1.1 Arquitetura J2ME

J2ME introduz dois conceitos de arquitetura: configurações (*configurations*) e perfis (*profiles*). Configurações são responsáveis por definir um conjunto de características de baixo-nível, como:

- Classes básicas de Java;
- Características de programação de Java;
- Características da máquina virtual.

Perfis são responsáveis por características de alto-nível do dispositivo, como componentes de interface gráfica, formas de armazenamento de dados etc.

Configurações e perfis são fornecidos separadamente para facilitar a portabilidade entre os diversos dispositivos. Configurações servem para aumentar a portabilidade entre muitos dispositivos diferentes. Já perfis definem as características de um dispositivo específico ou de um grupo de dispositivos similares.

A Figura 3 mostra que J2ME é composto de uma máquina virtual, uma configuração e um ou mais perfis.

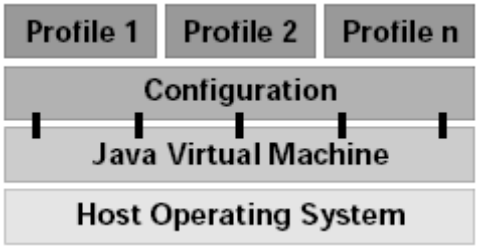


Figura 3 - Arquitetura alto-nível de J2ME[4].

Configurações são especificações definidas pela Java Community Process (JCP)[5] em cooperação com vários participantes da indústria. Atualmente, existem duas configurações:

- Connected Limited Device Configuration (CLDC)
- Connected Device Configuration (CDC)

Cada uma das configurações possui uma máquina virtual e um conjunto de dispositivos-alvo, como pode ser observado na Tabela 1.

Tabela 1 - Comparação entre as diferentes configurações de J2ME

Configuração	Máquina Virtual	Exemplo de dispositivo
CLDC	KVM	Cellular phones PDAs
CDC	CVM	Pocket PCs Set-top boxes

Connected Limited Device Configuration (CLDC) é um mínimo denominador comum de tecnologia Java suportada em dispositivos móveis, com capacidade computacional limitada. Atualmente, existem duas versões de CLDC, a 1.0 e a 1.1.

Os dispositivos-alvo do CLDC 1.0 têm as seguintes características gerais:

- 160kb a 512kb de memória disponível para Java,
- Processadores de 16-bit ou 32-bit;
- Consumo de baixa potência, geralmente utiliza bateria;
- Conexão a algum tipo de rede, em geral *wireless*, com banda restrita.

CLDC é baseado em J2SE com a retirada de algumas funcionalidades. Algumas das funcionalidades suprimidas:

- *Java Native Interface* (JNI);
- *Reflection*;
- *Thread groups* e *daemon threads*;
- RMI e *serialization*;
- *Floating point* (*float* e *double*).

A retirada de funcionalidades de J2SE ocorreram por diversos motivos. Algumas dessas funcionalidades foram removidas apenas para reduzir o tamanho da API. Entretanto, algumas características foram explicitamente removidas por serem caras do ponto de vista de processamento ou de armazenamento na memória. Um bom exemplo disso são os tipos primitivos de ponto flutuante. Eles não foram inclusos no CLDC 1.0 porque foram considerados muito caros em termos de tamanho de código e de poder de processamento.

Existem casos em que a funcionalidade não foi excluída totalmente. É o caso da verificação de classes. Em virtude que realizar tal verificação é muito caro em termos de processamento e ocupa bastante memória da máquina virtual, essa funcionalidade foi reduzida. Em vez de ocorrer da forma tradicional, a verificação é feita em dois passos. O primeiro é realizado na estação de

trabalho do desenvolvedor. Nele ocorre uma pré-verificação, deixando o segundo passo, que ocorre no dispositivo, leve o suficiente para os padrões do CLDC1.0

Outras reduções ocorreram em virtude de CLDC não implementar todo o modelo de segurança de Java. Suprimir algumas características do modelo de segurança de Java, torna-se um risco potencial de segurança.

Uma das reduções mais simples que ocorreu na implementação de CLDC foi a retirada de classes que podem ser construídas depois pelo o usuário. Um exemplo disso é *ThreadGroup*. A retirada de alguns métodos também foi usada para diminuição do tamanho da biblioteca.

Outras reduções de características tais como o *finalization* e as referências fracas foram removidos do CLDC, primeiramente porque essas características não são utilizadas inteiramente ou necessárias.

Em março de 2003, a Sun lançou a CLDC 1.1. Tal versão não continha mudanças drásticas, uma vez que os desenvolvedores da CLDC estavam satisfeitos. Abaixo, algumas das características adicionadas na nova versão :

- Suporte a ponto flutuante:
    - Tipos primitivos;
    - Classes *Float* e *Double*;
    - Vários métodos foram adicionados a outras bibliotecas, agora com suporte a ponto flutuante.
- As classes *Calendar*, *Date* e *TimeZone* foram reprojetadas para se parecerem mais com J2SE;
- Várias das pequenas bibliotecas foram modificadas e tiveram seus bugs corrigidos.
- Memória mínima compartilhada para execução passou de 160kb para 192kb.

A *Kilobyte Virtual Machine* (*KVM*), nome dado para a máquina virtual J2ME, está presente em dispositivos com bastante restrição. A KVM possui uma CLDC, seja 1.0 ou 1.1. Essa máquina virtual adere ao máximo possível à

especificação da máquina virtual Java. Todavia, a capacidade é definida amplamente pela especificação CLDC. Basta ver o fato já citado anteriormente referente a ponto flutuante. Na CLDC 1.0, não há suporte a ponto flutuante, logo a KVM não reconhece o mesmo.

A KVM requer um pequeno espaço de memória para rodar no dispositivo, entre 40kb e 80kb, dependendo das opções de compilação e da plataforma alvo. A KVM foi projetada para ser mais rápida possível , atingindo de 30% a 80% da velocidade da JVM padrão.

A *Connected Device Configuration* (CDC) é a outra configuração definida para J2ME. A CDC roda em cima da *C-Virtual Machine* (CVM). Essa configuração tem como alvo os dispositivos com mais de 512kb de memória, entretanto, foi projetado para plataformas com cerca de 2MB de memória. Os dispositivos que suportam CDC têm mais poder de processamento que os dispositivos CLDC e possuem suporte a conexões com redes de alto desempenho. Na figura 4 podemos ver a diferença das arquiteturas de J2ME para CDC e CLDC.

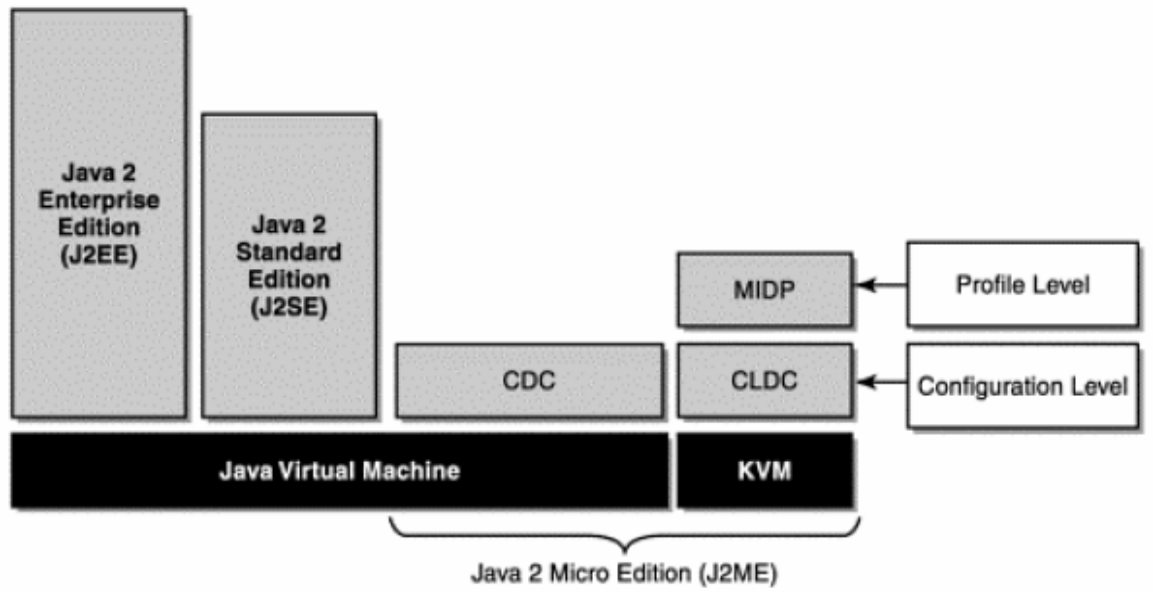


Figura 4 - Arquiteturas Java[6].

Assim como a KVM, a CVM é uma máquina virtual que atende às especificações da JVM. A CVM foi desenvolvida para aprimorar o desempenho para as plataformas-alvo e dispositivos de rede. Um dos aprimoramentos se refere ao *garbage collection*. Na CVM, ele não varre o *heap* em busca de objetos sem referência, uma vez que ele já conhece todos os ponteiros da aplicação, não sendo necessário consumir ciclos extras para fazer essa verificação.

Para aumentar a portabilidade entre as plataformas, a CVM implementa *multithreading*. Isso é extremamente vantajoso, já que, cada sistema operacional implementa *multithreading* à sua maneira. Todavia, é possível que a CVM possua *multithreading* nativo, basta o fabricante achar que seja necessário. A tabela 2 mostra uma comparação entre as duas configurações de J2ME[4].

Tabela 2 - Requisitos para as configurações J2ME.

	CDC	**CLDC**
Memória para executar Java (no mínimo)	512 kilobytes	128 nkilobytes
Memória para alocação de memória em tempo de execução	256 kilobytes	32 kilobytes
Outras características	Conectividade de rede alta, algumas vezes pode ser persistente.	Conectividade de rede intermitente e baixa largura de banda, geralmente dispositivos móveis.

2.1.2 Desenvolvendo em J2ME

Jogos desenvolvidos para o mercado de celulares em J2ME precisam atender à maior quantidade possível de aparelhos. Isso implica em lançar o mesmo jogo tanto para os aparelhos mais modestos quanto para os mais modernos.

Lançar jogos para os modelos de dispositivos mais simples nem sempre é fácil, devido à grande restrição deles. Restrições como desempenho, memória e tamanho do jogo podem destruir um bom jogo. Atualmente, os modelos mais

simples não possuem, nativamente, nenhum suporte ao desenvolvimento de jogos.

A Sun, percebendo o enorme potencial do mercado e a falha que o MIDP 1.0 possuía (não possuir API para jogos), resolveu lançar a API de MIDP 2.0. Com esse lançamento, vários problemas foram sanados. Os celulares passaram a ter um mínimo suporte ao desenvolvimento de jogos, com a adição de um pacote projetado exclusivamente para isso.

Entretanto, os celulares continuaram a evoluir e novas funcionalidades eram necessárias. Para atender a essa constante demanda, a comunidade Java decidiu criar um processo para definir as API necessárias. Tais API são conhecidas como JSR. Existem várias JSR, que vão desde o acesso a *bluetooth* do dispositivo a JSR de serviços de localização.

Isso atendia às necessidades dos fabricantes para oferecer mais funcionalidades nos seus aparelhos. Entretanto, foi gerado uma enorme fragmentação no mercado. Muitas produtoras de jogos optaram por não utilizar muitas dessas JSR, uma vez que a maioria atingia uma parcela muito pequena do mercado.

Em virtude desse amplo mercado com uma grande variedade de aparelhos e funcionalidades, os desenvolvedores de jogos têm que se adaptar para poder desenvolver (e consequentemente, vender) o mesmo jogo para diversos aparelhos. É necessário fazer o *porting* de um determinado jogo (ou aplicação) para que ele possa rodar em vários dispositivos.

Tal processo é bastante caro. Além de haver diferenças de tamanho de tela e de funcionalidades oferecidas pelos aparelhos, existe o fato que as API de desenvolvimento de J2ME são implementadas pelos fabricantes (a Sun apenas especifica a interface). Isso gera um comportamento diferente entre aparelhos de fabricantes diferentes.

Existem formas para contornar tal problema, todavia, não há uma solução definitiva. Este projeto procura uma solução que não tenha um custo elevado para os desenvolvedores de jogos.

3. SharpLudus

Fábricas de software consistem em uma abordagem de desenvolvimento que visa aumentar a produtividade e reduzir o tempo de produção de software, focando em uma linha de produção (domínio). Tais fábricas são pouco comuns no universo dos desenvolvedores de jogos.

O SharpLudus é uma fabrica de software simples com o objetivo de diminuir os custos de produção de jogos do tipo *adventure 2D*. A utilização do SharpLudus tem como conseqüência o fato de que, de modo a produzir jogos para múltiplas plataformas, apenas alterações pontuais e modulares são necessárias, como em seu gerador de código.

O domínio focado pelo SharpLudus (*adventure 2D*), é um estilo de jogo bastante popular para *mobile*. A construção do jogo se dá através de múltiplos *screens* (salas e telas) interligados.

Tais *screens* são modelados através de uma linguagem de domínio-específico (DSL) visual[7], fornecida como parte da fábrica, chamada SharpLudus *Game Modeling Language (SLGML).* Esta DSL descreve boa parte do funcionamento do jogo. Tal DSL descreve um conjunto de *rooms* (salas) e telas de informação com diversos metadados úteis para a geração de código.

A SLGML tem como grande vantagem o fato de não ser usada apenas para descrição visual. Ela é usada também para a geração de código. Isto evita possíveis inconsistências entre o que foi documentado e o que foi implementado. Na verdade, um modelo em SLGML não é apenas documentação, sendo um artefato vivo que serve como input para outras etapas (automatizadas, inclusive) do processo.

Associados à SLGML, existem também possuem validadores semânticos para evitar que o criador do jogo (*game designer*) modele algo inconsistente com as regras do domínio (por exemplo, uma sala inatingível).

3.1 Game Modeling DSL (SLGML)

Como dito anteriormente, a *Game Modeling DSL* descreve várias informações úteis para a criação do jogo. Descreve configurações do jogo (resolução, tamanho de *tiles*, etc), estados do jogo (*rooms*) e o seu fluxo, incluindo condições de saída e outras propriedades.

A SLGML se apóia em seis conceitos-base para a descrição do jogo que podem ser vistos na figura 5. São eles:

- *AudioComponent*: um conceito abstrato que representa todos os sons existentes no jogo. É especializado em dois outros conceitos:
    - *BackgroundMusic*, que contem informação sobre a música de fundo a ser reproduzida;
    - *SoundEffect*, que define todos os outros sons do jogo.

- *Entity*: assim como o *AudioComponent*, é um conceito abstrato. Define todos os objetos do jogo que interagem entre si. Possui três especializações:

    - *MainCharacter* é o personagem principal, controlado pelo usuário;
    - *NPC* (*non-playable character*) representa personagens não controlados pelo usuário.
    - *Item*, que como o próprio nome já diz, se refere a itens do jogo. Assim como os *NPCs*, interagem com o *MainCharacter*.

- *EntityInstance*: representa a instância de um *Entity*, contendo informações como posição, velocidade, pontos de vida etc.

- *Event*: são regras que determinam o que deve acontecer no jogo. Um evento é formado por um conjunto de gatilhos (*triggers*) e reações (*reactions*).

- *Sprite*: Define imagens estáticas (frames), que juntas formam uma animação. São usadas tanto por conceitos mais simples (como *tile*) como por *Entities* (como para movimentar um NPC).

- *GameState*: Representa o fluxo de jogo. Cada *GameState* possui uma ou mais condições de saída, que ligam um *GameState* a outro. É especializado pelos conceitos *InfoDisplay* e *Room*. O *InfoDisplay* exibe informações básicas na tela. É útil para fazer telas de introdução, *game over* e de ajuda entre outras. O *Room* é o espaço pelo qual as entidades se movimentam.

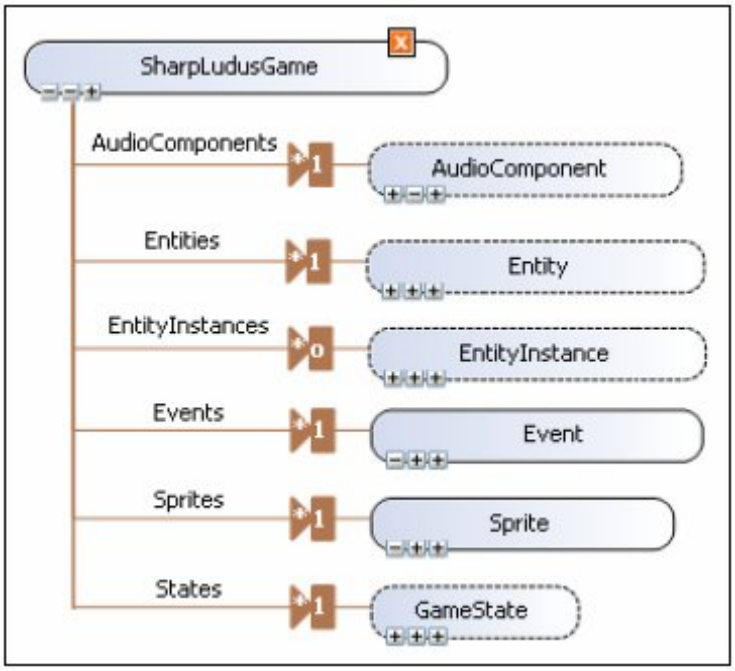
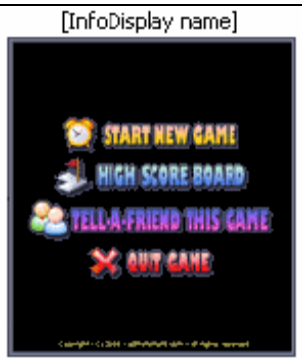


Figure 5 - Conceitos base para a modelagem de dados no SharpLudus.

3.2 Sintaxe da SLGML

A SLGML possui uma sintaxe (visual) simples. Na Tabela 3, são exibidos seus elementos:

Tabela 3 - Elementos gráficas do SharpLudus.

Elementos gráficos	Descrição
	***InfoDisplay***: Uma tela de informação é representada pela figura ao lado. Possui um texto em sua parte superior externa, que exibe seu nome.
	***Intro Purpose Decorator***: Esta imagem é usada para indicar que um *InfoDisplay* é a tela de introdução do jogo. É desenhada na parte superior interna, logo abaixo do nome do *InfoDisplay*.
	***Game Over Purpose Decorator***: Esta imagem é usada para indicar que um *InfoDisplay* é a tela de *game over*. É desenhada na parte superior interna, logo abaixo do nome do *InfoDisplay*.
	***Room***: Uma sala do jogo é representada pela figura ao lado. Possui um texto na parte superior externa, que exibe seu nome.
	***Transition***: As transições entre estados (telas e salas) são representadas visualmente como setas pretas.

3.3 Validadores Semânticos

Além de ajudar os game designers com recursos visuais para edição, a SLGML também possibilita verificação semânticas. Isto é feito através de validadores semânticos. A lista abaixo mostra alguns exemplos de regras semânticas associadas à SLGML e reforçadas por meio dos validadores:

- Um *GameState* tem que possuir pelos menos uma condição de saída;
- Um jogo tem que possuir um personagem principal (*MainCharacter*);
- Um jogo só pode ter uma única tela de introdução;
- Um jogo só pode ter uma única tela de game over;
- Um *Entity* tem q possuir pelo menos um *Sprite*;
- Todos os *GameStates* têm que ser atingíveis.

3.4 Game Engine

A *game engine* usada pelo fábrica de software SharpLudus é uma extensão da *game engine* disponibilizada pelo *DigiPen Institute of Technology* [8]. A engine original foi desenvolvida em *C#*[9] e utiliza a API multimídia do *DirectX*[10]. Algumas características da engine incluem:

-Criação e manipulação das entidades do jogo, incluindo atribuição de *sprites* e movimentação;

- Interação com o teclado;

- Suporte a efeitos sonoros;

- Manipulação de texto.

Entretanto, a engine original não continha suporte a todas as necessidades do SharpLudus[2], como *game states* e *game events*. A extensão cobriu todas as necessidades da fábrica. Tal expansão aumentou em 40% o código da engine, passando de 1700 linhas de código em 20 classes para 2340 linhas de código em 45 classes.

Nas figuras 6, 7 e 8 é possível visualizar a arquitetura deste game engine. As respectivas classes são responsáveis por:

- *Game*: define as principais informações do jogo, como frame hate, high score, resolução etc.
- *SoundEffect*: controla os efeitos sonoros do jogo.
- *Event*:, define a estrutura dos eventos de um jogo. Possui uma lista de *Trigger* e *Reaction*.
- *ITrigger*: interface para os *Triggers*.
- *IReaction*: interface para os *Reactions*.

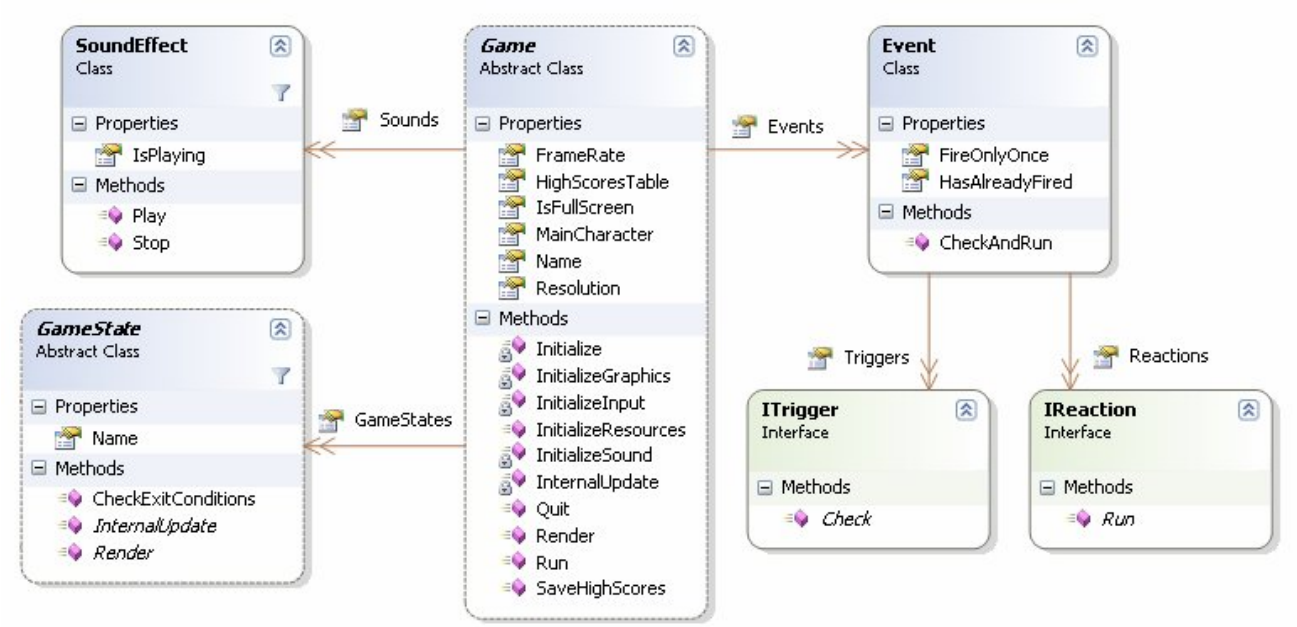


Figura 6 - Arquitetura da game engine do SharpLudus.

- *GameState*: estrutura geral para os estados do jogo.
- *Room*: estrutura das salas nas quais o jogo ocorre.
- *InfoDisplay*: estrutura dos displays de informação.
- *ExitCondition:* define a condição de saída de um *GameState*. Todo *GameState* deve especificar uma condição de saída.
- *BackgroundMusic*: música de background. Todo *GameState* pode ter um *BackgroundMusic*.

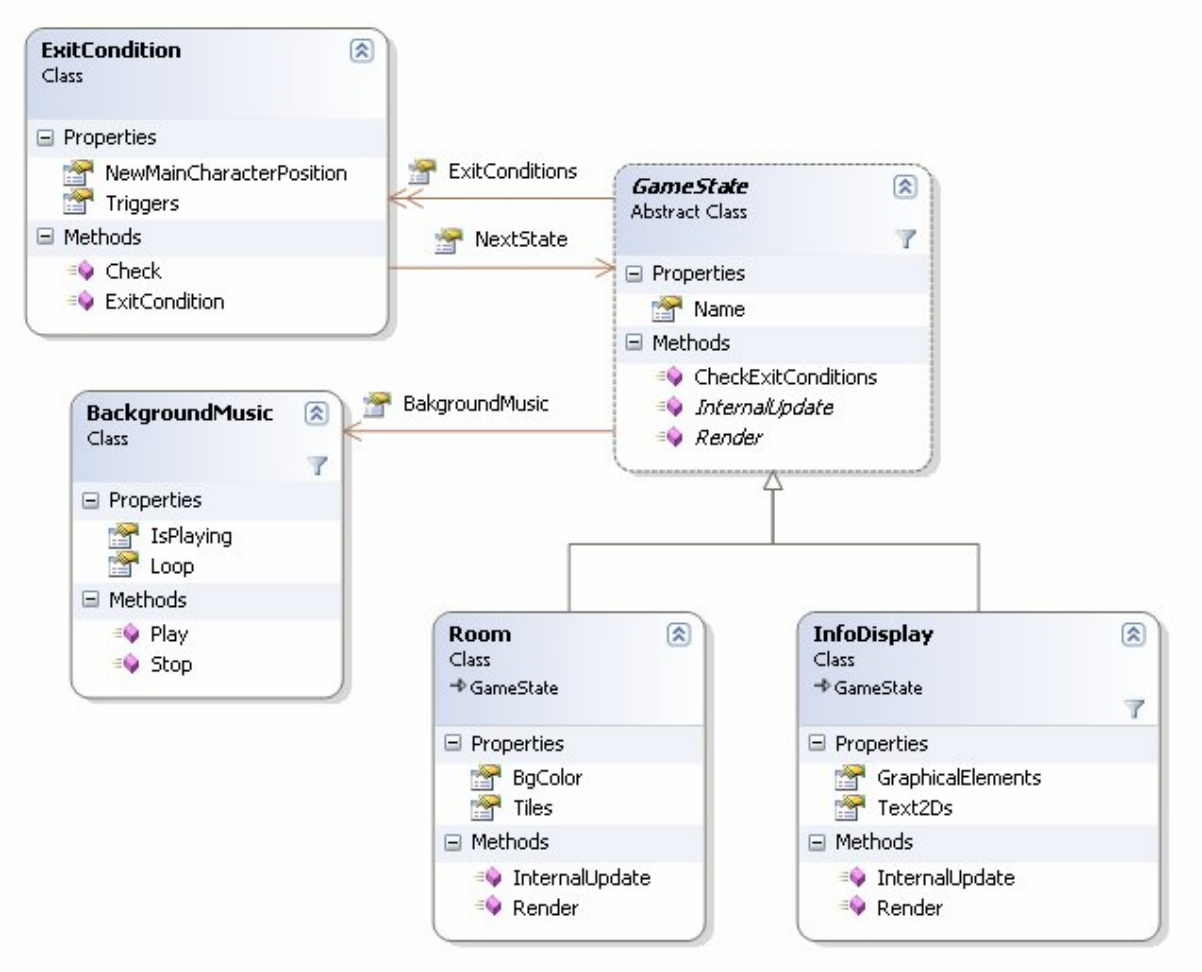


Figura 7 - Arquitetura da game engine do SharpLudus.

- *Picture*: encapsula cada imagem do jogo.
- *Frame*: encapsula uma *Picture*.
- *Sprite*: define as principais informações para uma animação no jogo, como frame atual, fim da animação etc.
- *Entity*: estrutura a qual define uma entidade do jogo.
- *Item*: entidade a qual representa os itens do jogo.
- *MainCharacter*: entidade a qual encapsula o personagem principal.
- *NPC*: entidade a qual representa todos os *non-player character* (*NPC*) do jogo.

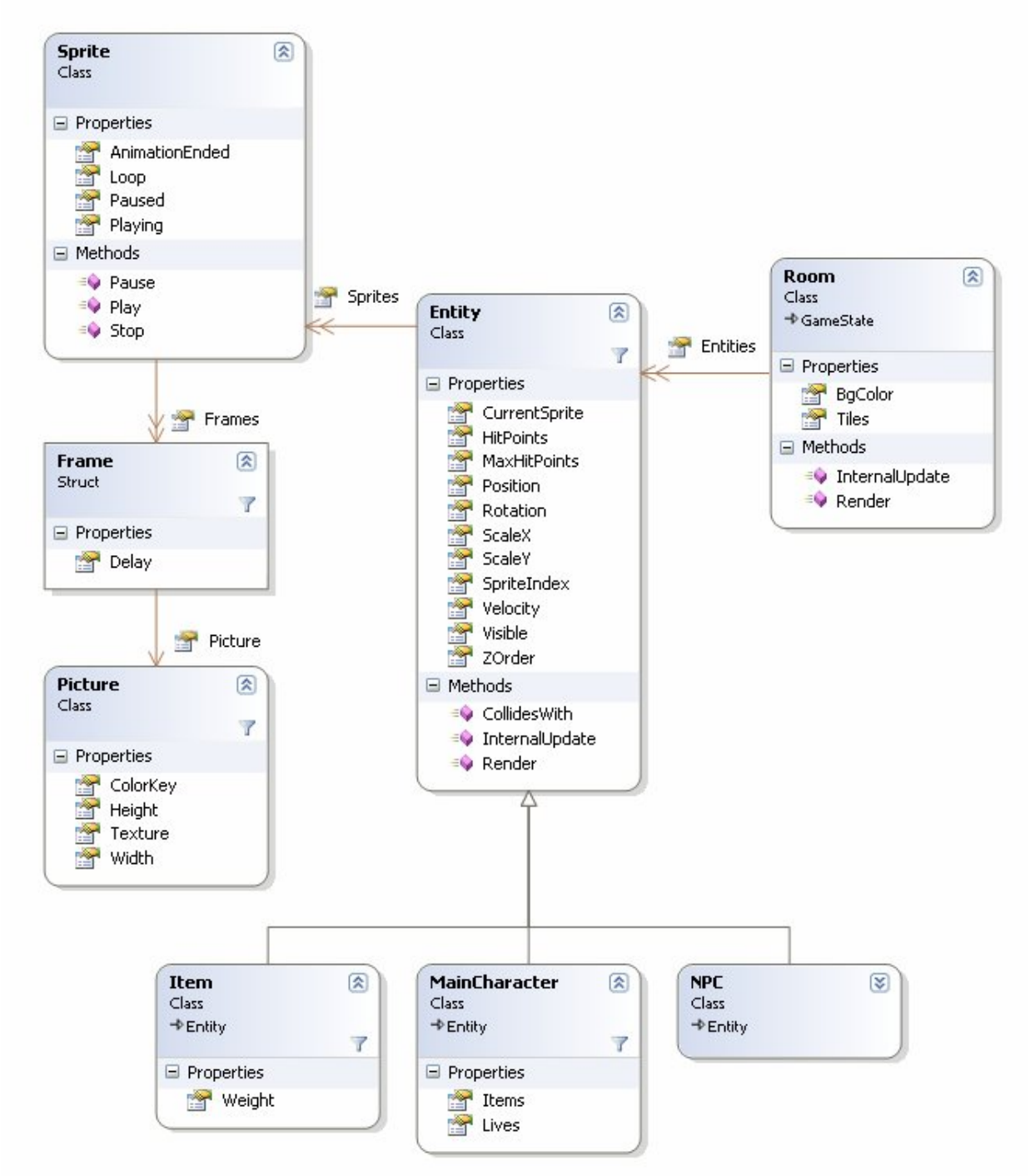
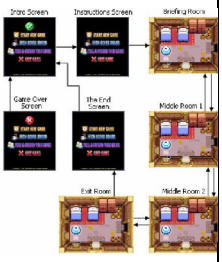


Figura 8 - Arquitetura da game engine do SharpLudus.

A construção de um jogo para esta game engine ocorre por partes. Básicamente as partes mais simples são incializadas primeiro e englobadas por partes mais complexas da engine. Na tabela abaixo é possível visualizar esta construção.

Tabela 4 - construção do elementos do jogo

Imagem	Frame	Sprite	Entidade	Sala	Jogo
		John Parado	John MainCharacter		
		John Andando			
					
					

3.5 Code Generator

O SharpLudus[2] possui um gerador de código para a *game engine* descrita anteriormente. Tal gerador de código é a única grande alteração no SharpLudus desse trabalho e será descrita mais à frente.

O gerador de código o qual o SharpLudus[2] recebe como entrada um arquivo SLGML. A geração de código se da através de uma linguagem de script disponibilizada pelo *DSL Tools*[7]. Tal linguagem é semelhante a *C#*[9] e é usada para manipular as informações do arquivo .

Neste script, todo o código entre as *tags* <# e #> é considerado a parte lógica do gerador. Nela processamos as informações necessárias para a geração do código. Já código escrito entre as *tags* <#= e #> deve retornar alguma expressão, a qual é escrita no arquivo gerado. Todo o texto fora desta *tags* é considerado "caracteres de escape", ou seja, códigos os quais serão escritos pelo gerador. Na figura 9 pode se ver um exemplo de um gerador de código responsável pela geração da classe sprite.

```
public static class Sprites {
<#
foreach(Sprite sprite in this.SharpLudusGame.GameSprites) {
    string spriteName = sprite.Name.Trim().Replace(" ","");
#>
    public static Sprite <#=spriteName#> {
        get {
            Sprite result = new Sprite();
            result.Name = "<#=spriteName#>";
            result.Loop = <#=sprite.Loop.ToString().ToLower()#>;
<#
    foreach(Frame frame in sprite.Frames) {
#>          Picture <#=frame.Picture.Name#> = new Picture(
                @"<#=frame.Picture.FilePath#>",
                Color.<#=frame.Picture.TransparentKey#>);
            Game.Add(<#=frame.Picture.Name#>);
            Frame <#=frame.Name#> = new Frame(
                <#=frame.Picture.Name#>,
                <#=frame.Delay#>);
            result.Add(<#=frame.Name#>);
<#  }
#>          result.Play();
            return result;
        }
    }
<#  }
#>
}
```

Figura 9 - Trecho do script do gerador de código: gerando a classe sprite

No SharpLudus[2], o gerador de código gera várias classes em um único arquivo. Tais classes, geradas para a linguagem *C#*[9], são:

- *AudioComponents*: responsável por toda a inicialização dos elementos de áudio do jogo (*sound effect* e *background music*). Tal classe implementa o design pattern *Singleton*[18].
- *Sprites*: define todos os sprites do jogo.
- Uma classe para cada *Entity* especificada pelo game designer. Tais classes são extensões das classes *Item*, *MainCharacter* ou *NPC*.
- *EntityInstances*: responsável pela geração de todos as instâncias das entidades. Tal classe implementa o design pattern *Singleton*[18].

- *States*: responsável por prover informação sobre os objetos de screen, *room* e *information display*. Tal classe implementa o design pattern *Singleton*[18].
- A classe principal do jogo, a qual possui o mesmo nome da propriedade *Name* descrita em *SharpLudusGame*, herda da classe *Game* da game engine. Nesta classe gerada, as configurações do jogo são inicializadas e os eventos são registrados.
- *Program*: contem o método *Main* e é responsável inicialização e execução do jogo.

4. Mudanças para geração de código em J2ME

Este capitulo descreve as mudanças necessárias para a geração de código para a plataforma J2ME[4].

Com um arquivo único e próprio para a geração de código, o SharpLudus possibilita a geração de código para múltiplas game engines, de linguagem de programação diferentes ou não.

Todavia o suporte a arquivos Portable Network Graphics (PNG)[11] foi adicionado para facilitar o uso de transparência nas imagens e para diminuir o tamanho final do arquivo Jar[13].

Algumas restrições devem ser seguidas quando se modela um jogo para a Plataforma J2ME no SharpLudus, utilizando o gerador de código deste trabalho:

- Não há suporte a transparência para imagens no formato Bitmap (BMP)[12], em virtude da game engine desenvolvida não suporta tal característica.
- Os frames do devem ter o mesmo tamanho, isto é necessário uma vez que a classe *Sprite* de MIDP 2.0 usada na game engine exige que os frames tenha o mesmo tamanho.
- Os arquivos de resources (imagens e sons) devem ter nomes diferentes, isto ocorre porque todos os arquivos de resources são salvos no mesmo diretório, o que não acontece com o SharpLudus original.
- As teclas devem ser mapeadas da seguinte for:
    - *LeftArrow* é equivalente a tecla 4 ou a tecla esquerda do direcional;
    - *RightArrow* é equivalente a tecla 6 ou a tecla direita do direcional;
    - *UpArrow* é equivalente a tecla 2 ou a tecla cima do direcional;
    - *DownArrow* é equivalente a tecla 8 ou a tecla baixo do direcional;
    - Return é equivalente a tecla 5 ou a tecla central do direcional.

4.1 Code Generator

O novo gerador desenvolvido continua a receber como entrada um arquivo SGML e é escrito na mesma linguagem de script que o gerador original. Apesar da linguagem de script ser bastante intuitiva, foi decidido que o gerador de código deveria ser o mais simples possível, deixando toda a complexidade para a game engine.

Em algumas das classes geradas, são inicializados muitos objetos estáticos, um recurso bastante usado no desenvolvimento de jogos para J2ME.

Mesmo com as mudanças no gerador, a estrutura final do arquivo gerado é bastante semelhante com o arquivo gerado inicialmente para C#. Entretanto isso não significa que a codificação do gerador seja semelhante, como podemos constatar na figura 10, a qual também define um trecho do código responsável pela geração dos sprites.

```
    static {
        CreateImage createImage = CreateImage.getInstance();
        SpriteVector spriteVector = SpriteVector.getInstance();
        Sprite result;
<#
        String[] atributos = new String[this.SharpLudusGame.GameSprites.Count];
        int indexAtributos = 0;

        foreach(Sprite sprite in this.SharpLudusGame.GameSprites) {
                string spriteName = sprite.Name.Replace(" ","_").ToUpper();

                foreach(Frame frame in sprite.Frames) {
                        string frameName = frame.Name.Trim().Replace(" ","");
#>
                        createImage.addImage("<#=getPNGString(frame.Picture.FilePath)#>");
<#
                }
#>
                result = new Sprite(createImage.getFinalImage(),
                                createImage.getFrameWidth(), createImage.getFrameHeight());
                //retorna a posição no Vector de Sprites
                spriteVector.createSprite(result);
<#
                atributos[indexAtributos++] = spriteName;
        }

#>
        }
<#
        for(int i = 0; i < atributos.Length; i++) {
#>
        public static final int <#=atributos[i]#> = <#=i#>;
<#
        }
#>
```

Figura 10 - Gerando os sprite para J2ME

Assim como no gerador original, o novo gerador também gera várias classes em um único arquivo. As classes geradas são:

- *Resources*: responsável por toda a inicialização dos elementos de áudio do jogo (sound effect e background music) e todos os sprites. No caso dos elementos de áudio, eles são inicializados em objetos estático. Já os sprites são inicializados em um bloco estático.
- Uma classe para cada Entity especificada pelo game designer. Tais classes são extensões das classes Item, MainCharacter ou NPC.
- *EntityInstances*: responsável pela geração de todos as instâncias das entidades. Tal classe inicializa seus objetos estáticos através de blocos estáticos.
- *States*: responsável por prover informação sobre os objetos de screen, room e information display. Assim como EntityInstances, inicializa seus objetos estáticos através de blocos estáticos.
- *GameEvents*: nela as configurações do jogo, como tamanho do mundo, são inicializadas e os eventos são registrados.

4.2 Game Engine

O desenvolvimento de uma *game engine* foi necessária. A *game engine* da *mobili games*[14] foi utilizada como referência, entretanto esta engine tratava apenas de aspectos mais simples como:

- Captura de teclas;
- Som;
- Fluxo de tela;
- Suporte a menu.

Como na game engine original do SharpLudus, suportes adicionais foram necessários para atender as necessidades do projeto. Porém, a game engine para J2ME[4] ficou mais compacta que a engine original, como podemos ver na tabela 5.

Tabela 5 - Comparação entre game engines

	Engine C#	Engine J2ME
Número de classes	45	31
Linhas de código	2340	Cerca de 2000

Dessas 31 classes, oito são destinadas as *Reactions* e sete destinadas aos Triggers, sobrando dezesseis classes as quais podem ser observadas nas figuras 11 e 12 e possuem as seguintes funcionalidades:

- *AudioComponents*: possui o controle total dos componentes de áudio. Nesta classe os sons são armazenados, executados e interrompidos. As referências para os componentes de áudio em *Resources* é a forma para acessar tais componentes. A classe *ÁudioComponents* implementa o design pattern *Singleton*[18].
- *CreateImage*: classe responsável pela manipulação das imagens do jogo. Foi necessário a criação desta classe em virtude do SharpLudus não construir *Sprites* com uma única imagem.
- *Entity*: classe abstrata a qual tem como especialização as classes *Item*, *MainCharacter* e *NPC*.
- *Item*: encapsula os itens do jogo.
- *MainCharacter*: definição do *MainCharacter*.
- *NPC*: definição dos *non-player character* (*NPC*) do jogo.
- *Event*: o encapsulamento dos eventos do jogo ocorrem nesta classe.
- *Trigger*: classe abstrata a qual define os *Triggers* do jogo.
- *Reaction*: classe abstrata a qual define as *Reactions* do jogo.
- *GameStates*: classe abstrata a qual tem como especialização *InfoDisplay* e *Room*.
- *InfoDisplay*: define as telas de informação do jogo.
- *Room*: define a estrutura das salas nas quais o jogo ocorre.
- *Game*: classe principal do jogo. É responsável por atualizar e desenhar todo os componentes do jogo.

- *MIDletController*: classe necessária a todas aplicações desenvolvidas para a plataforma J2ME[4]. Nesta classe é definido os comportamentos para inicialização, interrupção e destruição da aplicação. A esta classe também foi atribuído o controle do loop principal do jogo.
- *SpriteVector*: é responsável por armazenar todos os sprites do jogo. Assim como *AudioComponents*, a forma para acessar os sprites deve ser feita pelas variáveis estáticas em *Resources*. Também implementa o design pattern *Singleton*[18].
- *Text2D*: indica onde e que texto devem ser pintados na tela.

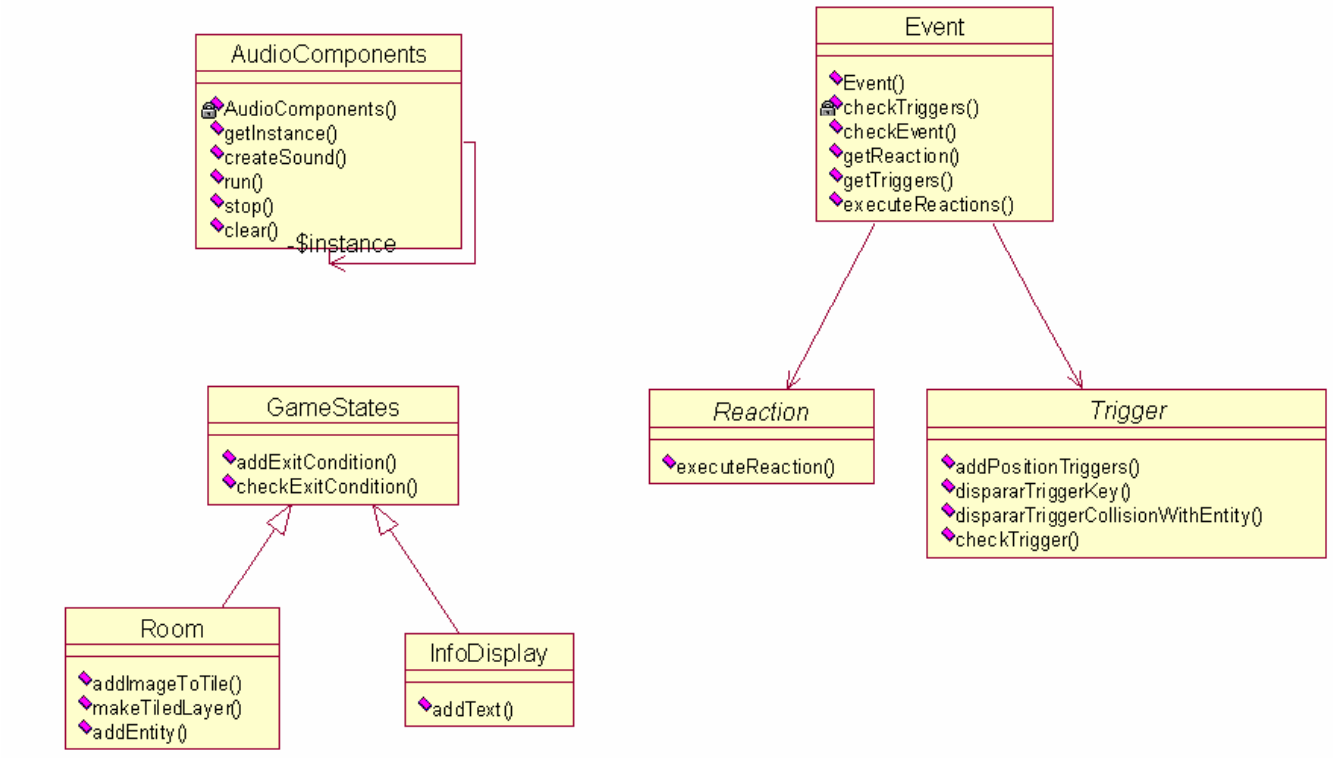


Figura 11 - Arquitetura da game engine para plataforma J2ME

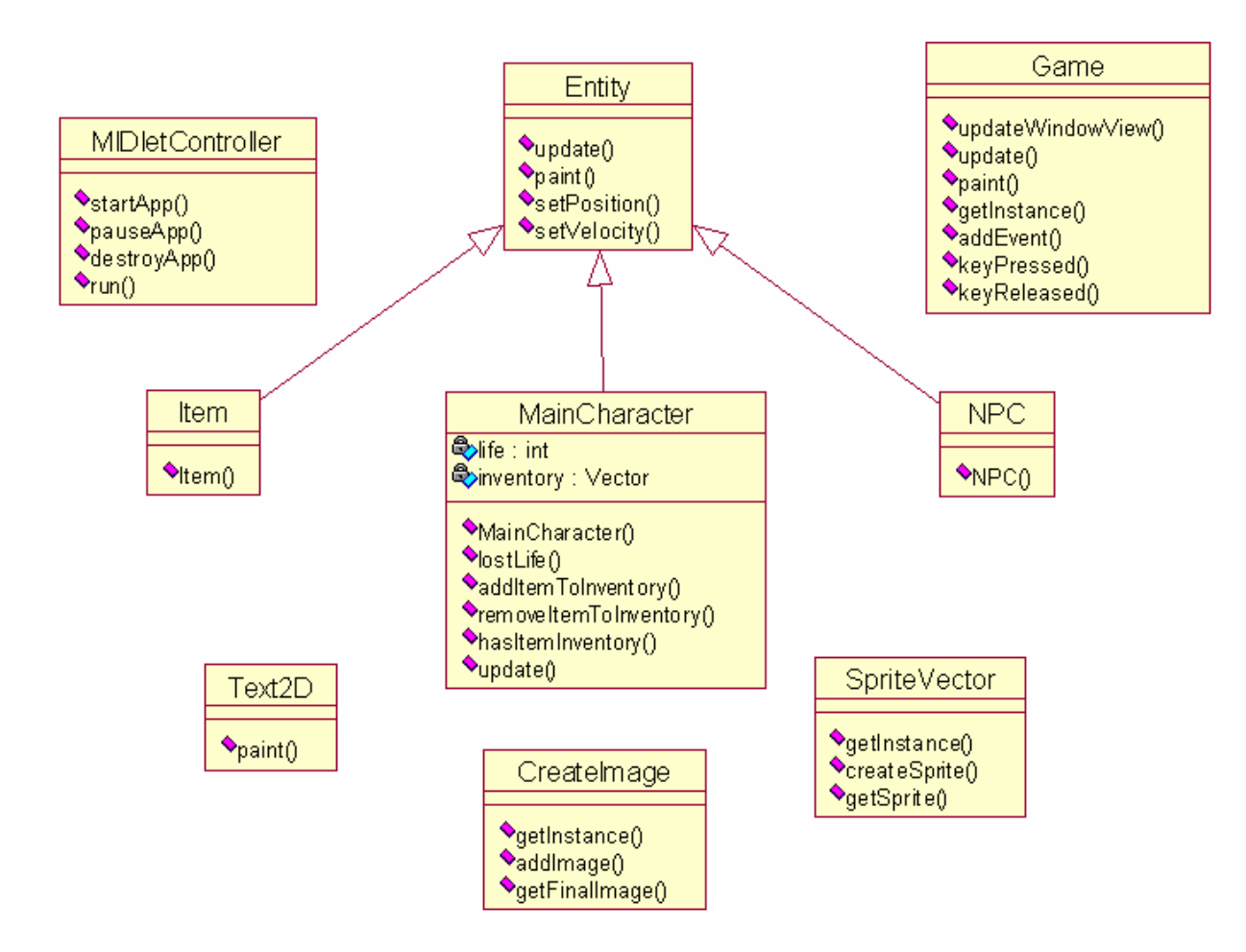


Figura 12 - Arquitetura da game engine para plataforma J2ME

5. Estudo de caso

Este capitulo mostra um exemplo do que pode ser feito com as modificações no SharpLudus[2].

Embora as modificações para que fosse possível gerar código para J2ME[4] foram bastantes, a modelagem do jogo continua de forma semelhante, como podemos ver na figura 12.

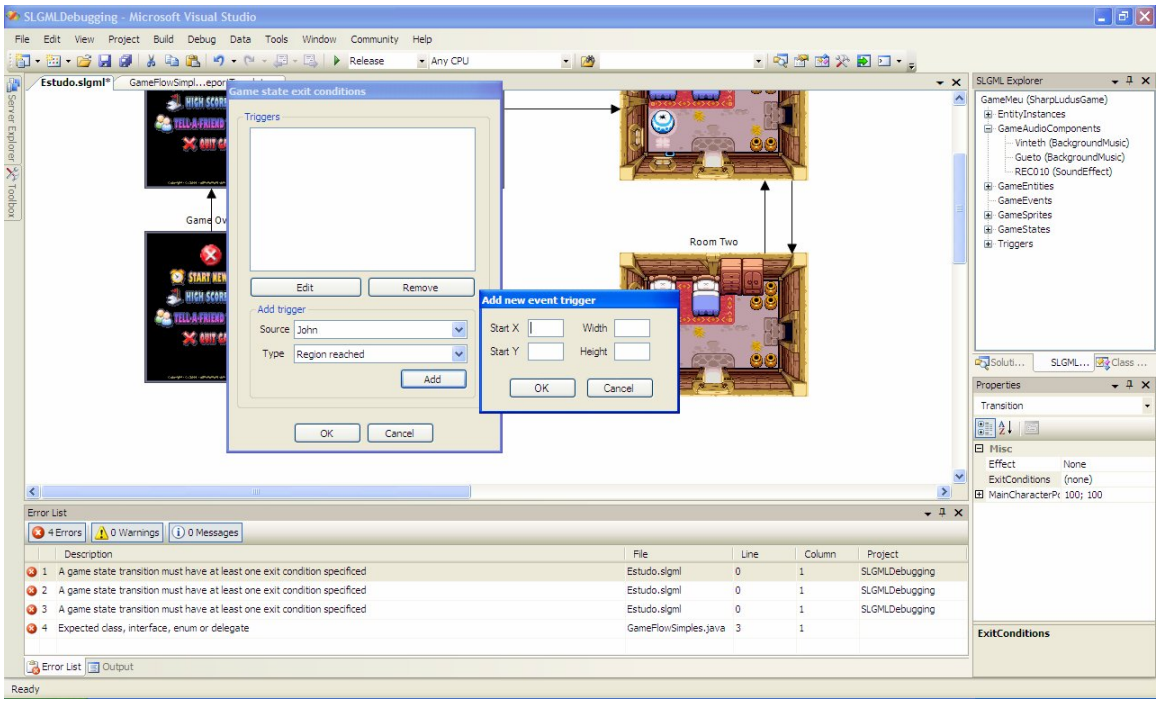


Figura 13 - Modelagem do jogo

Entretanto existe a necessidade de seguir as restrições descritas no capítulo anterior. Como no SharpLudus o tamanho da tela é igual ao tamanho do mundo jogo, uma janela de visão foi implementada para os *Rooms*, Todavia, para as telas de *Informa Display* é necessário que os elementos gráficos fiquem no canto superior esquerdo, como podemos constatar na figura 13. A área de pintura depende do tamanho da tela do celular.

Depois que a modelagem do jogo é feita, basta copiar a classe gerada (*Resources.java*) para a raiz dos *sources* da *game engine* em J2ME[4]. A parti desse passo, é possível rodar o jogo como um projeto normal em J2ME.

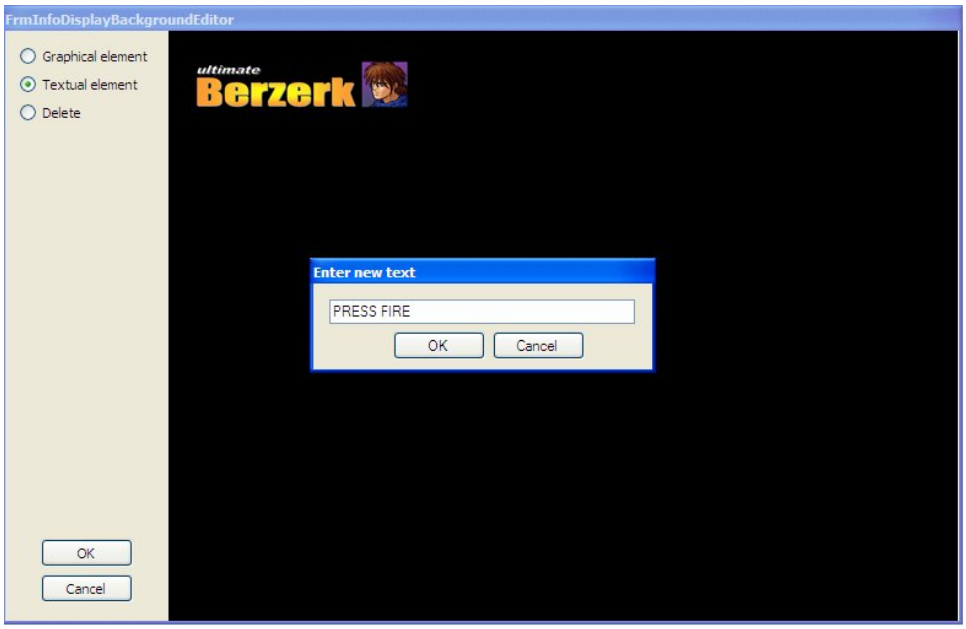


Figura 14 - Edição de Info Display

Mesmo conseguindo executar um jogo, como mostra a figura 14, a ferramenta ainda deve passar por melhorias, para deixar de ser uma ferramenta acadêmica para se tornar uma ferramenta comercial.

Figura 15 - Imagem do jogo

6. Conclusão e trabalhos futuros

A utilização de fábrica de software no desenvolvimento de jogos é possível, como mostra o SharpLudus. Seu uso para a geração de código para múltiplas plataformas foi demonstrado neste trabalho, onde a principal mudança foi o gerador de código e a *game engine*, como descrito no capítulo quatro.

Entretanto mudanças de paradigma são sempre lentas e neste caso de uma fábrica de software para a plataforma J2ME[4], mostrou-se pouco madura. Problemas quanto à restrição de desempenho, memória e espaço não tiveram uma solução adequada.

Este trabalho mostra um inicio de como industrializar o processo de produção de jogos para J2ME. Mas ainda a muito o que ser feito, como:

- Primeiramente, é necessário buscar a estabilidade do SharpLudus. O projeto foi desenvolvido em cima de versões beta do *DSL Tools*[7] e ainda possui vários *bugs*.
- Otimização da *game engine*.
- Tornar o gerador de código mais complexo para que a *game engine* possa se tornar mais simples.
- Adequação da SLGML para geração de código para J2ME.
- Geração de classes em arquivos separados.
- Diminuição da quantidade de classes geradas.
- Separação das definições de imagem e sprite.
- Integração com ferramentas de compactação de resources.
- Suporte ao processo de *porting*, tal importante no universo *mobile*. Neste caso seria necessário uma grande mudança na arquitetura do SharpLudus, uma vez que seria necessário ter uma mesmo lógica de jogo (eventos), mas com tamanhos de mundo, tile, imagens e sprites diferentes.
- Gerador de código respeitar os *know issues* dos celulares.
- Suporte a edição do código gerado, seria bastante importante essa *feature* para ajustes do jogo.

- Integração com ambientes de desenvolvimento para J2ME, como Eclipse[15] e Netbeans[16].
- Desvincular o SharpLudus do Visual Studio[17].
- Desvincular o SharpLudus do DirectX SDK[10].
- Melhoramento da interface gráfica.

7. Referências

[1] eMarketer, Mobile Gaming, http://www.emarketer.com/Report.aspx?mobile_gaming_feb06 - Último acesso em 04 de outubro de 2006.

[2] SharpLudus, http://www.cin.ufpe.br/~sharpludus/ - Último acesso em 04 de outubro de 2006.

[3] Informa Telecoms and Media, http:// www.informatm.com - Último acesso em 04 de outubro de 2006.

[4] Java™ Platform, Micro Edition, http://java.sun.com/javame/index.jsp - Último acesso em 04 de outubro de 2006.

[5] Java Community Process, http:// jcp.org - Último acesso em 04 de outubro de 2006.

[6] Java, http://java.sun.com/ - Último acesso em 04 de outubro de 2006.

[7] MSDN.com, Visual Studio 2005 Team System: Domain-Specific Language (DSL) Tools, http://lab.msdn.microsoft.com/teamsystem/Workshop/DSLTools/ - Último acesso em 28 de setembro de 2006.

[8] DigiPen.edu, MSDN Webcast Archive - Video Game Development: Learn to Write C# the Fun Way, http://www.digipen.edu/webcast.

[9] Microsoft.com, C# Developer Center, http://msdn.microsoft.com/vcsharp/.

[10] Microsoft DirectX, http://www.microsoft.com/directx - Último acesso em 28 de setembro de 2006.

[11] PNG, http://www.w3.org/Graphics/PNG/ - Último acesso em 03 de outubro de 2006.

[12] Bitmap, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/gdi/bitmaps_99ir.asp?frame=true - Último acesso em 03 de outubro de 2006.

[13] The Java™ Archive (JAR) file format, http://java.sun.com/docs/books/tutorial/jar/basics/ - Último acesso em 04 de outubro de 2006.

[14] Mobili Games, http://www.mobiligames.cjb.net - Último acesso em janeiro de 2005.

[15] Eclipse, http://www.eclipse.org/ .

[16] Netbeans, www.netbeans.org/ .

[17] Visual Studio, http://msdn.microsoft.com/vstudio/ .

[18] Elisabeth Freeman, Eric Freeman, Bert Bates , Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, 2004.