Gear2D: um motor extensível de jogos baseado em componentes

Leonardo Guilherme
Universidade de Brasília

Carla Castanho
Universidade de Brasília

Rodrigo Bonifácio
Universidade de Brasília

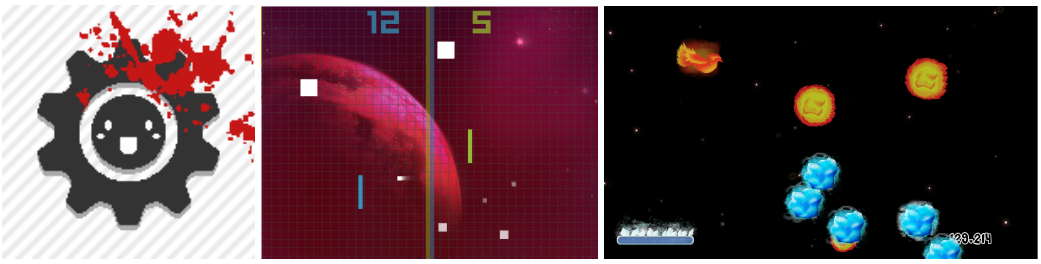

Figura 1: *Logo da Gear2D, screenshot do Pongroids, screenshot do ReIgnice*

Resumo

Motores de jogos elevam o nível de reuso na criação de jogos eletrônicos, centralizando em APIs coerentes com o domínio as funções mais comumente utilizadas. Entretanto, utilizando os motores atuais, a arquitetura de muitos jogos é construída modelando as entidades através de hierarquias de classes, o que pode ocasionar conflitos de nomes e/ou duplicação de código. Motores baseados em componentes minimizam esses problemas, contudo, devido ao forte acoplamento entre os componentes, ainda são pouco flexíveis no que tange à extensão e adaptação de entidades. Este artigo apresenta a *Gear2D*, um motor de jogos 2D que possibilita maior flexibilidade e extensão de entidades de maneira dinâmica através de componentes desacoplados. A implementação de dois jogos utilizando a *Gear2D* permitiu observar um bom grau de reuso e facilidades para a construção de jogos de forma declarativa— características desejáveis no processo de desenvolvimento de jogos eletrônicos.

Keywords:: Motores de Jogos, Reúso, Gear2D

Author's Contact:

leonardo.guilherme@gmail.com
{carlacastanho, rbonifacio}@cic.unb.br

1 Introdução

Motores de jogos introduziram um nível maior de reuso e flexibilidade nas arquiteturas dos jogos eletrônicos; o que antes era construído para cada jogo diferente, passou a ser centralizado e incorporado num conjunto de APIs e *framework*s consistentes com o domínio.

Motores de jogos, livres e proprietários, estão disponíveis tanto para desenvolvedores independentes quanto para grandes estúdios [dev 2011], proporcionando redução no ciclo de desenvolvimento e melhoria da qualidade no produto final. Além disso, os motores de jogos fazem com que os desenvolvedores se concentrem na criação do jogo propriamente dito, sem se preocuparem com outros aspectos que não fazem parte dos requisitos funcionais ou do *game design*.

Em alto nível, um motor de jogos pode ser considerado um software de prateleira (ou em Inglês *Commercial Off-the-Shelf* - COTS). Por outro lado, os motores de jogos, em geral, apresentam para o desenvolvedor uma arquitetura monolítica; entidades *in-game* são definidas através de relações de alto acoplamento e dependência, como herança[1] [Bilas 2002]. Atribuir características às entidades dos jogos com o uso de hierarquias de classes muitas vezes requer herança múltipla [West 2007], o que está associada a problemas como colisões de nomes, combinação de métodos, e herança repetida [Singh 1994].

[1]Por exemplo: `http://irrlicht.sourceforge.net/forum/viewtopic.php?f=19&t=26852`

Além disso, quando a taxonomia das entidades se torna longa, gerenciar a inserção de novos tipos de entidades que possuam características de duas famílias distintas se torna uma tarefa difícil, mesmo com a adoção de recursos linguísticos que simulam herança múltipla através de *Mixins* [I. Aracic and K.Ostermann 2006] ou *Traits* [Schärli et al. 2003], recursos presentes em linguagens pouco usadas na construção de motores de jogos, ou com a possibilidade de implementação de múltiplas interfaces (como suportado pela linguagem Java [Gosling 2000]).

Para contornar esse problema, alguns motores de jogos apresentam, aos desenvolvedores, uma arquitetura baseada em componentes (ex: Unity3D [uni 2011], Push Button Engine [pbe 2011], Gluon [glu 2011], entre outras). Nesse caso, entidades são definidas através da agregação de componentes ao invés de herança. Por exemplo, características associadas a dinâmica de corpo rígido, renderização, reprodução de áudio e outras comuns a todas as entidades ficam disponíveis através de componentes específicos e configuráveis. Esta disposição permite uma grande flexibilidade na criação de novas entidades. Por outro lado, o uso de motores que permitem referência direta entre componentes (como a Unity3D) não oferecem bons mecanismos para adaptar e estender os jogos dinamicamente, uma característica relevante para diferentes tipos de jogos, como os jogos educacionais que devem se adaptar às ações dos jogadores e os jogos online em massa (*Massive Multiplayer Online Games*, MMOG).

Esse artigo apresenta *Gear2D*, um motor de jogos 2D de propósito geral baseado em componentes, concebido para promover a flexibilidade e extensibilidade de jogos em tempo de execução. As principais contribuições envolvem (a) uma discussão sobre as decisões de projeto da *Gear2D*, bem como a apresentação de alguns detalhes de sua implementação (Seção 2); e (b) o detalhamento da construção de dois jogos desenvolvidos com a *Gear2D* (Seção 3), destacando os benefícios de reúso e facilidade na construção de jogos. Conclusões e trabalhos futuros são discutidos na Seção 4.

2 *Gear2D*

O desenho do motor *Gear2D* objetiva promover uma arquitetura baseada em componentes para a criação de jogos eletrônicos, com foco em flexibilidade e adaptação de entidades em tempo de execução. Tais objetivos foram alcançados com o uso extensivo de alguns padrões de projeto da *Gang of Four* [Gamma et al. 1995] (discutidos na Subseção 2.3), enquanto alguns padrões menos conhecidos na literatura (como o padrão *Adaptive Object Model* - AOM [Yoder and Johson 2002]) emergiram de forma menos planejada.

Por exemplo, para contornar os problemas ocasionados por longas hierarquias de entidades, acima mencionados, o padrão *Composite* [Gamma et al. 1995] foi adotado pelo motor de jogos. Nesse cenário, personagens (e outras entidades *in-game*) exercem o papel de *containers* para diversos componentes, permitindo que as características de uma entidade sejam atribuídas por meio de agregação.

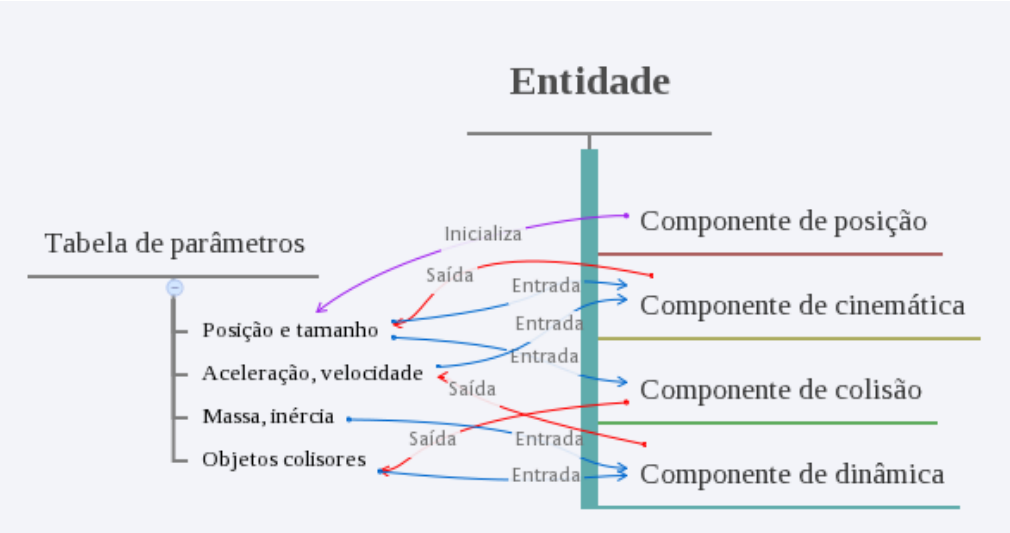

Figura 2: *Fluxo de execução dos componentes posição, cinemática, dinâmica e colisão. Componente de cinemática usa parâmetros de aceleração e velocidade para alterar parâmetros de posição, que são utilizados pelo componente de colisão para detectar colisões expondo-as num parâmetro que é usado pelo componente de dinâmica para produzir alterações nos parâmetros de aceleração e velocidade.*

Ou seja, um componente de renderização pode ser atribuído a um *sprite* para o mesmo possa ser exibido na tela.

2.1 Decisões de projeto

Embora a agregação seja um mecanismo suportado no paradigma orientado a objetos, a atribuição de novas propriedades, de forma dinâmica e independente de uma hierarquia única, não é um recurso comumente explorado no design OO; apesar de oferecer meios para adaptar as entidades dos jogos após uma definição inicial.

Para garantir a recombinação de componentes na *Gear2D* foi necessário a construção de um *framework* base para prover a habilidade de adicionar e remover componentes em tempos distintos de execução. Esse *framework* age como uma meta-arquitetura sobre as regras de negócio, ou seja, as definições de *gameplay* e das entidades requeridas.

Baseando-se na abordagem *Adaptive Object Model*, a informação sobre a composição de cada entidade foi elevada ao nível do metadado através de arquivos de configuração[2] e o suporte à adaptação em tempo de execução foi implementado através do padrão *Property* [Yoder and Johson 2002; Lasater 2006].

Para evitar a necessidade da referência direta a atributos e métodos entre componentes, no intuito de promover reuso e evitar acoplamento, a metáfora do quadro-negro [Craig 1993] foi implementada: componentes operam sobre uma tabela de parâmetros e atributos compartilhada entre eles para produzir um ou mais resultados desejados (veja Figura 2), sem a necessidade de fazer referência direta.

Gear2D utiliza a tabela de parâmetros não apenas para armazenar valores relevantes, mas também para a comunicação entre os componentes através do mecanismo *signal and slots*, semelhante ao provido pela plataforma Qt [Blanchette and Summerfield 2008]: cada parâmetro aceita *listeners*, que são notificados para cada nova alteração realizada num parâmetro que lhes interessa, provendo meios para a realização de chamadas de métodos.

Em resumo, as responsabilidades do *framework* implementado na *Gear2D* são:

- Carregar arquivos de configuração para a disposição inicial dos objetos e componentes da aplicação.
- Permitir a inserção e remoção de componentes em tempo de execução.
- Garantir que cada componente obtenha uma fatia de tempo para seu próprio processamento.
- Permitir a comunicação entre os componentes, disponibilizando para isso uma tabela extensível de parâmetros e atributos.
- Carregar novas entidades em tempo de execução através de uma API disponibilizada.

Utilizando tais características do *framework*, é possível criar componentes que, juntos, completam um motor de jogos e, através do uso destes, é possível construir um jogo eletrônico. Atualmente existem componentes para posicionamento espacial, execução de áudio, renderização 2D, entrada via teclado, detecção de colisões, cinemática e dinâmica de corpo rígido, entre outros, disponibilizados em um sítio web[3].

2.2 Classificação de componentes

Entre os focos do *framework* apresentado está o reuso e a colaboração. Utilizando a vantagem do desacoplamento entre componentes é possível criar diferentes componentes parametrizados pelo mesmo conjunto de atributos mas que produzam comportamentos distintos ou que possuam implementações independentes.

Para tal, uma classificação de componentes se torna necessária, de maneira que estejam agrupados por similaridade, permitindo que um componente possa ser substituído por outro quando necessário, característica importante em, por exemplo, sistemas multiplataforma (comportamentos iguais mas implementações diferentes).

Dessa maneira, componentes são classificados segundo uma família e um tipo, identificando-os de maneira única. Tipos são pertencentes à uma família e essas denotam uma funcionalidade que deve ser executada de forma compatível por todos os componentes desta família. Em outras palavras, uma família é usada para agrupar componentes por similaridade. Por exemplo, a família de renderização agrupa componentes que são destinados a produzir conteúdo visível.

2.3 Detalhes de implementação

Na *Gear2D*, os componentes são disponibilizados através de bibliotecas dinâmicas carregadas apenas em tempo de execução, sendo instanciados na medida em que são adicionados às entidades, permitindo que apenas o código necessário seja carregado e executado.

[2] Isso também classifica *Gear2D* como um motor orientado a dados (*data-driven*) [Bilas 2002]

[3] `http://gear2d.sourceforge.net/`

Isso é possível através de APIs para carga de bibliotecas dinâmicas como a libdl[4], existente na maioria dos ambientes POSIX (*Portable Operating System Interface*).

O padrão de programação *Abstract Factory* [Gamma et al. 1995] foi aplicado para a instanciação de entidades e seus componentes. Na fábrica de componentes, a família e tipo do componente são usados como parâmetros para a busca da biblioteca dinâmica, e essa exporta um método que é responsável por instanciar corretamente o componente, retornando para a *Gear2D* a respectiva referência. Assim, o tipo exato do componente só é conhecido por sua própria implementação (Figura 3).

Através dessa referência, o *framework* utiliza a tabela de parâmetros iniciais carregados dos arquivos de configuração. Tais parâmetros, escritos utilizando a linguagem YAML[5]), descrevem as características das entidades e permitem que os componentes sejam configurados antes de serem atribuídos às entidades. Isso compreende inicializar corretamente os parâmetros que necessitam na tabela de parâmetros e inicializar suas estruturas internas. Tais estruturas internas são de responsabilidade do componente e não do *framework*.

Durante a execução do laço de eventos, os componentes são agrupados por família e executados em sequência. Isso é, todos os componentes da família de cinemática são executados, em seguida todos da família de detecção de colisão são executados, e assim por diante. Durante o tempo de execução de cada instância dos componentes, a tabela de parâmetros pode ser acessada e seus valores lidos ou alterados conforme a necessidade, gerando o *gameplay* desejado (Figura 2).

Para o desenvolvimento do *framework* (e também dos outros componentes multimídia), foi utilizada a biblioteca *Simple DirectMedia Layer* (SDL[6]) para prover a API multiplataforma para a carga dinâmica de bibliotecas compartilhadas. SDL é uma biblioteca multimídia multiplataforma e de código aberto.

3 Criando o *Gameplay*

Diferentes motores de jogos e outras ferramentas de criação apresentam maneiras distintas de representar o *gameplay*. Em motores baseados em componentes, é natural que a dinâmica de jogo também seja representada utilizando componentes, se beneficiando dos outros componentes agregados numa mesma entidade. Essa abordagem também é utilizada na *Gear2D*. Nesta seção apresentamos alguns aspectos associados a implementação de dois jogos (*Pongroids* e *ReIgnice*) utilizando a *Gear2D*.

3.1 *Pongroids*

Pongroids é um jogo para até dois jogadores inspirado no clássico Pong [Wikipedia 2011] com tema espacial (uma partida de tênis jogada no espaço sideral). As raquetes representam os jogadores e são controladas através do teclado; o objetivo é fazer a bola atravessar o campo adversário sem ser rebatida por asteroides ou pelo outro jogador. Quando ocorre uma colisão entre um asteroide e a bola, o asteroide se divide em outros de menor tamanho é a bola rebatida para o lado oposto.

A construção de *Pongroids* além de servir como espaço de teste para componentes, foi essencial para identificar componentes desejáveis num motor de jogos e componentes específicos de *gameplay*. Conforme mencionado, as definições iniciais de cada entidade são criadas através de arquivos de configuração; cada arquivo corresponde à definição de uma entidade. Nessa definição está uma lista de componentes para serem adicionados e parâmetros iniciais a serem passados para esses componentes (como ilustrado na Figura 4).

Com o reúso do motor *Gear2D*, a construção do *Pongroids* exigiu um total de 502 linhas de código distribuídas em cinco componentes:

```
# lista de componentes a adicionar
attach: collider dynamics renderer

# detecção de colisão
collider.tag: bar
collider.ignore: asteroid leftfield

# parâmetros referentes à renderização
renderer.surfaces: bar=barleft.bmp
bar.alpha: 1.0
```

Figura 4: *barleft.yaml: Arquivo representando o jogador no campo esquerdo. O parâmetro* attach: *especifica a lista de componentes à adicionar. Os outros parâmetros indicam valores iniciais a serem usados pelos componentes no momento de sua adição. Por exemplo, o componente* renderer *utiliza* renderer.surfaces *para a listagem de imagens a carregar e renderizar.*

lobby: Componente que implementa o menu da tela inicial do jogo. Utiliza o componente de teclado para controlar propriedades de renderização de texto e para carregar o tipo de partida escolhida (um ou dois jogadores).

partida: Componente que implementa as regras da partida, isso é, a contagem de pontos e a atualização das imagens de fundo baseado no tempo do jogo.

asteroide: Componente para regular quantos e quais asteroides (pequenos, médios ou grandes) estão em jogo e quantos novos asteroides seriam instanciados quando ocorre uma colisão.

raquete: Componente que toca um som quando ocorre a colisão entre a raquete e a bola. Note que o controle da raquete é feito através do componente de controle de personagens, que alteram parâmetros de aceleração baseado em teclas acionadas pelo teclado, configuráveis também através de parâmetros. Tal componente não é específico do jogo *Pongroids*, mas sim disponibilizado juntamente com o motor.

raquete-ia: Componente que simula a inteligência artificial utilizada em uma das raquetes quando a partida é para apenas um usuário.

Todos os componentes precisam ser atribuídos a entidades para que possam funcionar e a comunicação entre duas entidades ocorre de forma flexível, através de parâmetros. Em outras palavras, um componente de uma entidade pode ler ou escrever parâmetros em outras entidades. Para tal, a API da *Gear2D* oferece mecanismos para recuperar instâncias de entidades bem como acessar parâmetros de entidade.

Por exemplo, o componente *lobby* cria instâncias dos menus e submenus dependendo do *input* do usuário, controlando suas propriedades diretamente para refletir as escolhas do jogador. Já o componente *match* controla a pontuação, registrando-se como *listener* nos parâmetros de posicionamento da bola do jogo. Sempre que avançar uma das linhas de fundo ele ajusta a pontuação de acordo.

3.2 *ReIgnice*

Ignice é um jogo do gênero *side-scroller*; o personagem é representado por uma ave que possui dois estados: fogo e gelo. O desafio do jogo consiste na habilidade do jogador em evitar meteoros que são do elemento contrário. Isso é, enquanto a ave é de fogo, deve evitar meteoros de gelo e acertar meteoros de fogo. É possível inverter o estado da ave no decorrer da fase.

ReIgnice é um clone do jogo *Ignice*, com o intuito de fazer uma comparação entre duas abordagens distintas. *Ignice* foi originalmente construído apenas com a biblioteca SDL e uma arquitetura moldada para a sua construção. *ReIgnice* foi construído utilizando a *Gear2D*, reusando a maioria dos componentes já criados mas em uma outra configuração.

[4] http://linux.die.net/man/3/dlopen
[5] http://www.yaml.org, *YAML ain't markup language*
[6] http://www.libsdl.org

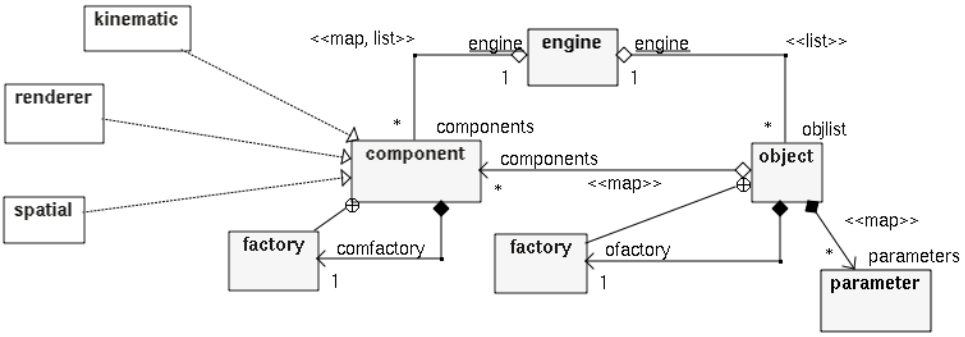

Figura 3: *Classes do* framework

Na implementação do *ReIgnice*, três novos componentes foram criados:

fenix: Componente adicionado à entidade da fênix, o personagem principal do jogo. Suas tarefas principais compreendem alterar a posição do personagem baseando-se em parâmetros de posição do mouse (que são reportados através do componente mouse), trocar a imagem atual do personagem (fogo, gelo e transições entre os elementos), gerenciar pontos de vida, entre outros.

meteoro: Componente utilizado nos meteoros de fogo e gelo. Note que o mesmo componente é utilizado para todos os tipos e tamanhos de meteoros.

splash: Componente utilizado na tela. Carrega o sai do jogo de acordo com a interação com o mouse.

Comparativamente com *Pongroids* (Subseção 3.1), foram observados muitos componentes em comum utilizados: componentes de renderização, cinética e colisão foram extensivamente usados nas duas implementações, sem alterações. O componente de corpo rígido não foi usado no *ReIgnice* pois não havia simulações físicas a serem feitas.

O código original de *Ignice*, que foi criado usando apenas a biblioteca SDL, é constituído de 2404 linhas de código C++, enquanto *ReIgnice*, em seus três componentes, totaliza 383 linhas de programação e 286 em arquivos de configuração (semelhantes aos descritos na Subseção 3.1), mantendo os aspectos originais de *gameplay*.

4 Conclusões e trabalhos futuros

Através do uso de padrões e abordagens como *AOM* e *Blackboard*, foi possível construir um *framework* que permite elaborar jogos de maneira declarativa, possibilitando que as entidades sejam estendidas em tempo de execução, adicionando novos componentes e parâmetros arbitrários de acordo com a necessidade do *gameplay*.

Usando a API disponibilizada aos criadores de componentes, é possível através da *Gear2D* fazer uso de uma tabela compartilhada de atributos para a criação de comportamentos originais, como demonstrado na Seção 3 com a implementação de *Pongroids* e *ReIgnice*, possibilitando redução considerável no esforço necessário para a construção de jogos diferentes.

No que tange à tabela de parâmetros, essa é indexada por *strings*[7], podendo se tornar um gargalo em termos de performance em situações que hajam muitos parâmetros em uma entidade. Esse ainda não é o ideal e maneiras de realizar a indexação desses parâmetros com maior performance estão sido estudadas.

Ainda, a API para desenvolvedores de componentes é disponibilizada em C++; alterações em componentes provocam recompilações do mesmo, o que também não é o ideal. No futuro, planeja-se a utilização da linguagem de scripting Lua, mapeando os parâmetros como variáveis e as funcionalidades da API como funções dentro do script, possibilitando menores ciclos de desenvolvimento.

Referências

BILAS, S., 2002. A Data Driven Game Object System. http://scottbilas.com/files/2002/gdc_san_jose/game_objects_paper.pdf. Acessado em agosto/2011.

BLANCHETTE, J., AND SUMMERFIELD, M. 2008. *C++ GUI Programming with Qt 4*. Prentice Hall Open Source Software Development Series.

CRAIG, I. D. 1993. Formal techniques in the development of blackboard systems. *IJPRAI 7*, 2, 197–219.

2011. Devmaster: 3d game and graphical engines. http://www.devmaster.net/engines/. Acessado em agosto/2011.

GAMMA, E., HELM, R., JOHNSON, R. E., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

2011. Unity3d. http://gluon.gamingfreedom.org/. Acessado em agosto/2011.

GOSLING, J. 2000. *The Java language specification*. Prentice Hall.

I. ARACIC, V. GASIUNAS, M. M., AND K.OSTERMANN. 2006. Overview of caesarj. *Transactions on Aspect-Oriented Software Development 3880* (February), 135–173.

LASATER, C. G. 2006. *Design Patterns*. Wordware Publishing.

2011. Push button engine. http://www.pushbuttonengine.com. Acessado em agosto/2011.

SCHÄRLI, N., DUCASSE, S., NIERSTRASZ, O., AND BLACK, A. 2003. Traits: Composable Units of Behaviour. In *ECOOP 2003*, 327–339.

SINGH, G. B. 1994. Single versus multiple inheritance in object oriented programming. *SIGPLAN OOPS Mess. 5* (January), 34–43.

2011. Unity3d. http://www.unity3d.com. Acessado em agosto/2011.

WEST, M., 2007. Evolve your Hierarchy. http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy. Acessado em agosto/2011.

WIKIPEDIA, 2011. Atari. "http://en.wikipedia.org/wiki/Atari". Acessado em agosto/2011.

YODER, J. W., AND JOHSON, R. 2002. The adaptive object-model architectural style. In *WICSA*.

[7] Sequência de caracteres literais representando texto