Projeto Ouroboros

Sistema de integração automatizada entre C++ e linguagens de *script*

Fernando Omar Aluani
Wilson Kazuo Mizutani

Trabalho de conclusão de curso

Orientador: Prof. Dr. Marco Dimas Gubitoso

São Paulo, Dezembro de 2013

Agradecimentos

Muitos de nós programadores sofremos de uma curiosidade mórbida em fazer programas incrivelmente elaborados e inusitados idenpendentemente da utilidade deles só para saber se uma ideia que tivemos realmente funciona. Muitas vezes é esse impulso que leva ao desenvolvimento de sistemas importantes, outras ele simplesmente leva a boas risadas. Mas uma coisa é certa: essas obsessões são boa parte do que nos faz querer aprender mais e mais.

Gostaríamos de agradecer a todos que incentivaram em nós essa curiosidade mórbida, tanto professores quanto amigos. Seja por ter-nos mostrado um truque de código, ou ensinado alguma técnica secreta de programação, ou ainda simplesmente apoiado uma de nossas ideias malucas. Muito do que fizemos nesse trabalho vem da nossa vontade de experimentar com combinações divertidas do que aprendemos ao longo da nossa vida como estudantes de Ciência da Computação.

Resumo

Este trabalho é sobre uma biblioteca `C++` que estamos desenvolvendo desde 2011. O propósito dela é automatizar a integração entre aplicações programadas em `C++` e *script*s escritos em `Lua` ou `Python`. Normalmente, o desenvolvedor da aplicação teria que escrever pelo menos algumas dúzias de linhas de código para que ela pudesse simplesmente carregar um *script* para incorporar seus dados e rotinas. É justamente esse esforço adicional qual queremos poupar ao usuário. Essa monografia fala sobre a pesquisa que fizemos por ferramentas que nos ajudassem, sobre o processo de desenvolvimento e sobre o produto final do nosso trabalho. Começamos por um capítulo introdutório, seguida da parte objetiva, composta por três capítulos, e depois tratamos a parte subjetiva, também dividida em três capítulos.

No primeiro capítulo (1 – Introdução), introduzimos a noção de *script*ing e a utilidade que ela tem em aplicações de computador. Usamos um exemplo para ilustrar um caso de uso, e discutimos brevemente as possibilidades atualmente existentes para integrar *script*s. Depois, contamos sobre a motivação que nos levou a elaborar esse projeto, assim como o que diferencia ele das outras soluções presentes na comunidade Web. Terminamos listando os objetivos principais do trabalho com relação à biblioteca de programação que desenvolvemos.

Em seguida, partimos para uma discussão mais conceitual ao longo do próximo capítulo (2 – Conceitos e tecnologias estudadas). Procuramos entender o que caracteriza as linguagens de programação que estamos tentando integrar: `C++`, `Lua` e `Python`. Para isso, exploramos as noções de linguagens compiladas contra linguagens interpretadas, e como o conceito de máquinas virtuais ajuda a entender melhor a relação entre elas. Ao final, tratamos com mais profundidade sobre as ferramentas que `Lua` e `Python` nos oferecem, para que fique mais fácil de entender o que falamos nos demais capítulos do trabalho.

Depois, apresentamos a estrutura do nosso projeto (3 – Estrutura do Projeto). Buscamos modelar uma solução para lidar com a integração automatizada que desejamos que nosso sistema forneça. Esse capítulo revela como dividimos nosso trabalho em duas grandes frentes, que chamamos de **incorporação** e **exportação**. Tentamos também deixar claro como o usuário poderá interagir com tudo isso.

Discutimos, no quarto capítulo (4 – Atividades), sobre a evolução do nosso sistema, desde sua origem a dois anos atrás até as decisões que tomamos ao longo desse último ano tendo em vista o trabalho de formatura. Contamos sobre como nossa participação no USPGameDev levou à ideia inicial do projeto, assim como o que mudou quando o desvinculamos do grupo. Em seguida, entramos em alguns detalhes da implementação do nosso sistema, segundo as duas partes principais idealizadas no capítulo anterior. Também desabafamos sobre o SWIG, uma ferramenta que nos ajudou inicialmente mas tornou-se uma enorme dor de cabeça posteriormente. Terminamos falando sobre algumas utilidades que fizemos em CMake para facilitar o uso da nossa biblioteca.

No último capítulo da parte objetiva (5 – Resultados) apresentamos o estado em que o projeto se encontra atualmente. Mostramos e exemplificamos as funcionalidades, com alguns trechos de código para deixar mais explícido o que ele é capaz de fazer. Aproveitamos para deixar algumas instruções de uso, incluindo configuração e compilação das partes relevantes do sistema.

A parte subjetiva começa com um capítulo para cada autor (6 – Fernando Omar Aluani, e 7 – Wilson Kazuo Mizutani), contando sobre nossas dificuldades e frustrações no trabalho. Também apresentamos uma relação de disciplinas que cursamos ao longo da nossa graduação que acreditamos ter nos ajudado de uma maneira ou de outra. Encerramos a parte subjetiva da monografia com um breve capítulo (8 – Próximos passos) listando as nossas metas futuras para nosso projeto.

Sumário

Capítulo 1

Introdução

Programas de computador frequentemente lidam com diversos tipos de conteúdo: textos, imagens, músicas, vídeos, etc. Eles são armazenados e manipulados na forma de dados, segundo codificações específicas. Alguns tipos de conteúdo são mais complexos do que outros, como o banco de dados da sua rede social favorita, ou um registro de tomografias médicas. Algumas vezes dados são tão complicados que se tornam programas por sí sós. É um caso bastante comum em jogos eletrônicos, onde parte do conteúdo é a inteligência artifical dos personagens ou o roteiro de eventos que ocorre para cada interação do jogador. **Linguagens de *script*** são ferramentas bastante populares para tratar conteúdo desse gênero.

Para tanto, existem diversas ferramentas disponíveis no mercado e na comunidade Web. Mas cada uma delas tem suas especifidades e restrições, dificilmente trabalhando com a noção de *script*s de forma mais geral, e ainda mais raramente de maneira automatizada. Por isso, a triste realidade é que integrar *script*s em sua aplicação, por mais valor que isso agregue ao seu conteúdo, exige muito esforço.

Portanto, para nosso trabalho de formatura, apresentamos uma solução em *software* que permite ao usuário integrar de forma automatizada as linguagens de *script* `Lua` e `Python` a um programa escrito em `C++`. Apesar de a princípio nos restringirmos apenas a essas duas linguagens de *script*, elaboramos a arquitetura do nosso sistema de modo a facilitar a contemplação de novas linguagens. A compatibilidade inicial com duas linguagens diferentes de maneira simultânea visa justamente demonstrar essa versatilidade. Nesta monografia, são explicados todos os principais mecanismos desse sistema, assim como a teoria e a tecnologia por trás deles.

1.1 Motivação

Como sugerido por um exemplo acima, uma das nossas principais motivações para esse trabalho é o desenvolvimento de jogos digitais. Mais especificamente, foi nosso trabalho no USPGameDev[1] que nos levou a ele. Afinal, em uma equipe de desenvolvedores de jogos não há apenas programadores, mas também *designers* e artistas, que pouco ou nada sabem sobre programação. Nada melhor do que usar linguagens de *script*, conhecidas por sua expressividade e facilidade de uso, para simplificar o trabalho deles e aumentar a produtividade geral do grupo.

O problema é que não conseguimos decidir qual linguagem de *script* usar. Por um lado tínhamos `Python`, com suas vastas extensões e facilidades, e por outro `Lua`, com sua simplicidade e versatilidade. Depois de discutir bastante e considerar as possibilidades, pensamos em uma idéia "melhor de dois mundos": comportar ambas as linguagens, e ocultar os mecanismos internos de cada uma de maneira que os outros desenvolvedores do grupo não precisassem se preocupar com qual estivesse sendo usada de fato. Além disso, como programadores frescos que somos, queríamos que o sistema cuidasse de todo o trabalho sujo envolvido e tivesse uma interface limpa e elegante de usar. Como contaremos mais adiante, isso tudo ocorreu há dois anos quando nós dois, autores deste trabalho, tornamo-nos os responsáveis por esse projeto dentro do USPGameDev.

[1] O USPGameDev é um grupo de pesquisa e desenvolvimento de jogos da Universidade de São Paulo. Ver http://uspgamedev.org/ (último acesso: 15/09/2013).

Conseguimos obter resultados satisfatoriamente funcionais com alguns meses de desenvolvimento. No entanto, no começo desse ano ainda havia vários aspectos que queríamos melhorar, principalmente devido às limitações impostas por uma das ferramentas que usávamos. Por isso, decidimos transformar esse sistema em nosso trabalho de formatura, dando continuidade aos aprimoramentos e registrando nossos aprendizados e experiências na monografia.

1.2 Objetivos

O objetivo desse nosso projeto, contando com o que já desenvolvemos antes de ele se tornar oficalmente nosso trabalho de formatura, consiste no desenvolvimento de um sistema de *software* capaz de integrar aplicações desenvolvidas em `C++` com *script*s escritos em **Lua** e **Python** mantendo as seguintes características:

1. A aplicação deverá ser capaz de acessar *script*s de maneira simples o suficiente para que ela não precise se referir diretamente aos mecanismos internos das linguagens envolvidas ou até mesmo saber qual a linguagem com que os *script*s foram escritos.

2. O sistema cuidará de disponibilizar as funcionalidades relevantes da aplicação de maneira que os *script*s usados por ela possam acessá-las sem grandes complicações.

3. O sistema será um *Software Livre*, e portanto deverá ser de fácil distribuição e inclusão nas aplicações que desejarem seus serviços.

Parte Objetiva

Capítulo 2

Conceitos e tecnologias estudadas

Sempre que encontramos um problema, buscamos uma solução. Muitos dos produtos que usamos diariamente são a manifestação dessas soluções que alguém projetou para lidar com os empecilhos do dia-a-dia, desde uma mobília para conseguirmos sentar e comer até aparelhos elaborados para detectar doenças em nossos corpos. Entender o problema que se quer resolver, e saber desprender a solução dessa análise é um procedimento fundamental ao projetar esses produtos – e o mesmo vale quando desenvolvemos um *software*. Para tanto, existem várias maneiras de se modelar os problemas, e uma das mais simples é na forma de uma pergunta.

Como integrar C++ *com linguagens de* script*?* Um dos primeiros conceitos com o qual aspirantes a programadores têm que se habituar é que programas são o resultado de um código misterioso escrito usando uma dessas chamadas linguagens de programação. A partir desse momento, costuma-se passar por aquele período onde se acredita que a quantidade de linguagens dominadas define uma métrica das habilidades do programador ou da programadora, e ficamos maravilhados pela miríade de linguagens diferentes. Até que entendemos que por trás de todas elas a ideia é relativamente a mesma, naquele momento em que percebemos que "ah, então é assim que se faz um laço nessa linguagem".

Mas anedotas à parte, permanece o fato de que cada linguagem de programação funciona através do seu próprio mecanismo de implementação. Como seria possível qualquer tipo de integração entre elas, então? Esse capítulo tem como objetivo esclarecer essa questão, e com isso destacar o problema que o nosso sistema foi projetado para resolver, além do método necessário para tanto. Começamos tentando entender melhor o que são esses elementos que tentamos integrar.

2.1 Linguagens de Programação

Uma das coisas mais importantes que todo programador aprende cedo ou tarde é que computadores *não são máquinas inteligentes*. Eles fazem apenas e exatamente aquilo que mandamos eles fazerem[1]. Por outro lado, é isso que torna eles mais confiáveis que humanos no que diz respeito a muitas atividades. E o que explica essa característica intrínsica dos computadores é que eles operam através de instruções [S.13a].

Quando recebemos uma instrução, tipicamente nos está sendo pedido para realizar uma ação, possivelmente envolvendo objetos ou outras pessoas. Com o computador, as ações que lhe são instruídas envolvem dados [Sta10]. As transformações e manipulações que ele pode executar sobre dados, assim como o que os dados podem representar compõem o que chamamos de **modelo computacional**, pois indicam as possíveis **computações** que podem ser realizadas dentro de tal modelo. Por sua vez, dizemos que as intruções que especificam a computação que desejamos que o computador faça formam um **programa**. Finalmente, isso leva ao nosso objeto de interesse: para expressar esse programa ao computador usamos uma **linguagem de programação**. De maneira mais formal, podemos definir os termos acima conforme [Aab96]:

[1] A menos de problemas de *hardware*, é claro.

Definição 2.1:

1. Um **modelo computacional** é uma coleção de valores e operações.
2. Uma **computação** é a aplicação de uma sequência de operações sobre um valor para obter outro valor.
3. Um **programa** é a especificação de uma computação.
4. Uma **linguagem de programação** é uma notação para escrever programas.

Essencialmente, linguagens de programação são a maneira que se encontrou para deixar claro quais instruções o computador deve seguir, visto que ele as seguirá à risca e portanto queremos ter certeza do que lhe estamos pedindo. Agora, se queremos integrar linguagens, precisamos entender como são seus modelos computacionais e como interagir com eles na prática. No caso, estamos mais especificamente interessados em C++ e em linguagens de *script*. Então vamos estudar, nas próximas sessões, as características que elas possuem, assim como as possibilidades que elas nos oferecem.

2.2 Da linguagem para a máquina

O processo que leva do programa escrito à sua execução pelo computador pode ocorrer de diversas maneiras. A mais direta é quando a linguagem usada é exatamente aquela que o computador foi projetado para processar. Em geral, uma linguagem dessas usa apenas valores numéricos para representar instruções explícitas de *hardware* para manipulação de dados brutos, e são conhecidas como **linguagem de máquina**. Para elas, o modelo computacional corresponde às operações que o processador é capaz de executar e os dados que podem ser armazenados nos diversos tipos de memória da máquina.

Definição 2.2:

Linguagem de máquina é toda linguagem de programação que é processada diretamente por uma máquina como um computador.

Como é de se imaginar, desenvolver com essas linguagens não é exatamente uma tarefa simples. Cada arquitetura de computador possui uma versão própria delas, tornando trabalhoso de fazer um mesmo programa compatível com todas elas. Além disso, é preciso muitas instruções para realizar computações simples, pois as operações que um processador é capaz de fazer são normalmente bastante primitivas — como movimentar dados na memória, ou comparar o valor dos dados em lugares diferentes dela. Para lidar com esses problemas existem dois métodos que essencialmente buscam abstrair as computações desejadas em instruções mais abstratas, isso é, em uma linguagem de programação mais acessível — ou de alto nível, como se costuma dizer.

2.2.1 Compilação

O primeiro e mais tradicional deles é traduzir programas escritos nessa nova linguagem para a linguagem de máquina do computador alvo. E, obviamente, não queremos fazer essa tradução nós mesmos, mas sim desenvolver um programa que a faça por nós. Programas assim são ditos **compiladores**, pois eles *compilam* a linguagem fonte para o linguagem objeto. Nesse caso, também dizemos que a linguagem fonte é uma **linguagem compilada**.

Definição 2.3:

1. **Compiladores** são programas que traduzem uma linguagem fonte de alto nível em uma linguagem objeto mais próxima de uma linguagem de máquina.
2. Uma **linguagem de programação compilada** é a linguagem fonte de um compilador.

Em particular, C++ é uma linguagem compilada. Em suas primeiras versões programas escritos nela eram compilados para a linguagem C, que por sua vez compilam para linguagens bastante próximas de uma linguagem de máquina, conhecidas como linguagens de montagem; atualmente, C++ é compilado diretamente para essas linguagens também [S.13b]. Isso significa que uma das linguagens que estamos tentando integrar no fundo é basicamente uma linguagem de máquina disfarçada de maneira bastante elegante.

Para que mais de um programa escritos em uma linguagem assim possam unir suas funcionalidades, é preciso duas coisas. Primeiramente, pelo menos um deles tem que conhecer os símbolos[2] relevantes do outro; ou seja, o código que declara esses símbolos deve estar incluso no programa que os usará. A segunda parte envolve, uma vez que os programas estejam compilados para o código objeto, ligar esses símbolos importados às suas respectivas definições no programa de origem deles. Isso é feito por um programa à parte, chamado de *ligador*, ou pelo próprio compilador. A maneira que tanto C quanto C++ encontraram para lidar com esse procedimento foi fornecer uma divisão para seus arquivos de código fonte: uma parte seriam os *arquivos de cabeçalho* — onde os símbolos são declarados — e a outra seriam os *arquivos de implementação* — onde a definição do que os símbolos significam é feita.

Isso permite que as partes de um programa com rotinas interrelacionadas possam ser compiladas em programas menores para ser ligados juntos depois, e assim apenas parte do programa precisa ser recompilada se só ela for modificada. Também fica possível fazer um programa auxiliar que possa ser ligado com qualquer outro que precise de suas funcionalidades, que é o que chamamos de uma *biblioteca*. A desvantagem de tudo isso são as restrições impostas ao processo de desenvolvimento: tanto o compilador quanto o ligador (quando necessário) precisam estar à disposição e devidamente configurados, e as bibliotecas necessárias devem estar acessíveis a eles.

2.2.2 Interpretação

O segundo método de usar uma linguagem de alto nível para abstrair linguagens próximas a linguagens de máquina seria providenciar um programa intermediário capaz de simular a execução de instruções. Chamamos esses programas de **interpretadores**. Também dizemos que sua linguagem fonte é uma **linguagem interpretada**. Essencialmente, ao invés de tentar levar a linguagem em si para mais perto da máquina, eles trazem a máquina para mais perto da linguagem. Isso é, produzem um modelo computacional virtual ao invés de serem computadores reais, porém continuam igualmente expressivos. Para tanto, eles simulam uma máquina condizente dentro de si, constituindo o que é conhecido como uma **máquina virtual**.

Definição 2.4:

1. Uma **máquina virtual** é uma simulação de um modelo computacional dentro de um programa.
2. **Interpretadores** são programas que processam uma linguagem fonte de alto nível usando uma máquina virtual.

[2] Esses símbolos são essenciamente nomes que estão associados a partes específicas de um programa, como suas variáveis e rotinas.

3. Uma **linguagem de programação interpretada** é a linguagem fonte de um interpretador.

Uma das grandes e principais vantagens de linguagens interpretadas é que suas abstrações podem produzir muito mais escrevendo muito menos. Outra vantagem é que como elas são processadas por um programa durante a execução deste, então o código fonte não precisa estar completo de uma vez só (como ocorre com a compilação). O interpretador pode ir recebendo as intruções uma de cada vez, conforme o usuário as digita nas famosas linhas de comando, por exemplo. Juntando essas duas características, é possível um usuário utilizar um terminal para acessar seu sistema operacional através de uma linguagem interpretada, mas também manter uma coleção de arquivos escritos na mesma linguagem implementando algumas rotinas comuns e repetitivas para que ele ou ela possa executá-los quando for mais conveniente. São esses arquivos auxiliares que chamamos de ***script*s** (do inglês, "roteiros").

Costumamos dizer que linguagens interpretadas que podem ser usadas para escrever *script*s são **linguagens de *script***. Infelizmente não é tão fácil definí-las formalmente, como indica [Wal13]. O que não chega a ser um problema para nossa discussão, pois as propriedades que nos interessam estão na definição das linguagens interpretadas.

E, com isso, chegamos às outras linguagens que queremos integrar: `Lua`[3] e `Python`[4], ambas consideradas linguagens de *script*. Com o objetivo de encontrar um meio de usá-las junto com `C++`, fizemos uma pesquisa sobre o funcionamento de suas respectivas máquinas virtuais. O que descobrimos, para nossa felicidade, é que ambas fazem parte de bibliotecas desenvolvidas em `C`, e que seus interpretadores nada mais são do que programas que usam essas bibliotecas. Eles cuidam de receber o código do usuário, seja através da linha de comando ou de *script*s, e passá-los para a máquina virtual, que os processa devidamente.

Então, obtivemos finalmente um meio de possivelmente resolver o problema proposto no começo desse capítulo. `C++` talvez possa ser integrado com `Lua` ou `Python` se um programa escrito nele for compilado e ligado junto com as bibliotecas dessas linguagens, obtendo assim acesso às suas respectivas máquinas virtuais. Tendo isso em mente, vamos buscar esclarecer o funcionamento dessas bibliotecas na seção a seguir, antes de passarmos para a solução que propusemos no projeto em si.

2.3 As bibliotecas de `Lua` e `Python`

Como vimos na seção 2.2.1, uma das coisas que um programa precisa para usar uma biblioteca são os cabeçalhos delas, pois eles fornecem os símbolos para as diferentes funcionalidades dela. Costumamos dizer que o conjunto desses símbolos declarados em cabeçalhos compõem uma interface, pois eles especificam como um programa pode interagir com aquela parte do código. No caso de uma biblioteca em `C` ou `C++`, chamamos isso de uma **Interface para Programação de Aplicações** ou simplesmente **API** (do inglês, *Application Programming Interface*)[5].

Essa seção é dedicada a um estudo breve das APIs que as máquina virtuais de `Lua` e `Python` oferecem. Ainda com o foco em encontrar meios para formar a integração que queremos, limitar-nos-emos a explicar apenas algumas funcionalidades mais relevantes dessas APIs.

[3] Versão 5.1, vide [IdFC]

[4] Versão 2.7, vide [pyt]

[5] Geralmente a ideia de API também vem atrelada a algum tipo de documentação que explica como usar a interface. Por exemplo, costuma-se disponibilizar uma referência do que cada rotina na biblioteca faz. Além disso, cada linguagem de programação tem sua própria maneira de expor uma API, o que vemos nessa seção é apenas a que `C` e `C++` usam.

2.3.1 Lua

Sempre que uma aplicação deseja usar uma máquina virtual de Lua, ela precisa usar uma estrutura declarada pela API chamada `lua_State`. Ela representa o estado da máquina e é usada por praticamente todas as rotinas da API. O comportamento geral da biblioteca abstrai uma pilha: as operações disponíveis alteram o estado da máquina inserindo, removendo e manipulando valores em uma estrutura de dados contendo uma sequência de valores, dos quais os últimos costumam ser os relevantes[6].

Esse método tem várias vantagens, dentre elas o fato que o tipo dos elementos na pilha não é explícito, permitindo que a linguagem tenha tipagem dinâmica. Além disso, o processo para executar rotinas fica bastante generalizado: os valores na pilha no começo da execução delas são os parâmetros, em ordem, e os valores que ela deixar ao final são os resultados devolvidos, também em ordem. Do lado de quem evoca a rotina, basta colocar ela mesma na pilha seguida dos parâmetros desejados e usar a função `lua_call()` da API. Por exemplo, suponha a seguinte função de potência em Lua:

```
1    -- Devolve x elevado a y
2    function pow (x,y)
3      return x^y
4    end
```

Usando a API para executar essa função, fazemos o seguinte:

```
1    // Indices passados nas chamadas abaixo sao a posicao da pilha na
2    // qual queremos aplicar a operacao. Numeros positivos indexam a
3    // partir da base da pilha, e negativos do topo para baixo.
4    lua_State *L;
5    double x, y;
6    // ... inicializa o estado e as variaveis ...
7    // Vamos supor que a funcao pow eh uma variavel global...
8    // A chamada abaixo coloca ela no topo da pilha
9    lua_getglobal(L, "pow");
10   // Verificamos se a funcao realmente existe e eh do tipo certo
11   if (lua_isnil(L, -1) || !lua_isfunction(L, -1)
12     throw runtime_error("Could not find function <pow>.");
13   // Empilhamos os parametros
14   lua_pushnumber(L, x);
15   lua_pushnumber(L, y);
16   // Nesse momento, o topo da pilha estah assim:
17   // [ ..., <pow>, x, y]
18   // Evocamos a funcao. Ela recebe dois parametros e devolve apenas um valor,
19   // entao indicamos nos argumentos passados abaixo
20   lua_call(L, 2, 1);
21   // Agora o resultado estah no topo da pilha e o resto foi removido
22   double result = lua_tonumber(L, -1);
```

Esse exemplo também serve para dar uma ideia de como funciona a conversão de valores entre a máquina virtual Lua e a linguagem C[7]. As funções com prefixo "`lua_push`" da API são usadas para converter para a máquina virtual colocando o valor no topo da pilha, enquanto que as com "`lua_to`" convertem da pilha para C. Note que essas últimas *não* removem os elementos da pilha, para isso existe a `lua_pop()`.

Um *script* Lua pode ser usado de duas formas pela máquina virtual. A maneira mais simples é tratá-lo como uma rotina e simplesmente executá-lo, processando as instruções escritas nele em sequência. A outra é tratá-lo como um "módulo", seguindo a nomenclatura da API. Nesse caso, as variáveis e funções definidas nele são agrupadas em um mesma estrutura. Por exemplo, um módulo seria assim:

[6] Uma implementação clássica de pilha teria operações que acessam apenas o último elemento inserido. Na API Lua, existem algumas funções que conseguem inserir e remover valores em posições arbitrárias da pilha.

[7] Como a API Lua está escrita em C, ela não oferece um meio direto de conversão para C++. No nosso sistema, usamos alguns artifícios para contornar isso.

```
1  module "funstuff"
2
3  function guesswhat ()
4    while true do
5      -- nothing
6    end
7  end
```

E ele seria usado da seguinte maneira:

```
1  require "funstuff"
2
3  print "Using a module"
4  funstuff.guesswhat()
```

A linguagem **Lua** em si oferece um tipo básico que representa uma tabela de símbolos[8], chamado de **table**. Elas são a base de todas as estruturas mais complexas da linguagem. A API fornece algumas funções para manipular essas tabelas, além de usar elas para alguns mecanismos internos como a armazenagem de variáveis globais e o controle de módulos. Também há um mecanismo para atrelar tabelas especiais aos valores dentro da máquina virtual para modificar o comportamento deles na linguagem (o que permite, por exemplo, construir tipos como números complexos compatíveis com as operações aritméticas usuais). Essas tabelas especiais são chamadas de *metatabelas*. Por outro lado, estruturas em **C** só podem ser inseridas e extraídas da pilha da máquina usando **void***, o que significa que é preciso tomar cuidado, pois o compilador acreditará que o programador sabe o que está fazendo. A linguagem chama essas estruturas fornecidas pela linguagem nativa de **userdata**.

Um tipo mais versátil é o tipo **function**, pois é possível passar funções em **C** para a pilha usando ele. Tudo que é preciso é que a assinatura delas seja **int <função> (lua_State *)**, isso é, que elas sejam funções que recebam o estado da máquina e devolvam um inteiro. Em sua implementação, elas devem manipular a pilha de maneira a satisfazer o protocolo especificado anteriormente: o que estiver na pilha inicialmente deve ser tratado como os parâmetros, e o que for deixado ao final serão os resultados, exceto que o valor inteiro devolvido indica quantos elementos do topo da pilha de fato devem ser devolvidos. Por exemplo, a mesma função **pow** mostrada acima podia ser escrita usando a API assim:

```
1  #include <lua.h>
2  #include <cmath>
3
4  int native_pow (lua_State* L) {
5    // A funcao luaL_error interrompe a funcao e joga um erro interno na
6    // maquina virtual. Costumamos chamar ela com return para deixar
7    // claro que nada vai acontecer depois dela.
8    if (lua_gettop(L) != 2) // verifica tamanho da pilha
9      return luaL_error(L, "Expected 2 parameters but got %d.", lua_gettop(L));
10   if (!lua_isnumber(L, 1) || !lua_isnumber(L, 2)) // verifica os tipos
11     return luaL_error(L, "Wrong parameter types");
12   double  x = lua_tonumber(L, 1),
13           y = lua_tonumber(L, 2);
14   lua_pushnumber(L, pow(x, y));
15   // Nao precisa limpar a pilha, basta avisar que soh o ultimo
16   // valor eh para ser devolvido
17   return 1;
18 }
```

Um procedimento similar é usado para fazer módulos **Lua** que venham de **C**, pois eles são construídos usando funções nativas como essa. A principal diferença é que além de usar a assinatura

[8] *Tabelas de símbolos* são estruturas de dados que costumam abstrair duas operações básicas em sua interface: busca e inserção de elementos através de *chaves*. Às vezes uma operação de remoção também está disponível, mas nem sempre ela é necessária. Em **Lua**, por exemplo, basta inserir o elemento nulo em uma chave para remover o que valor estava associado a ela.

correta, o nome da função que constrói o módulo precisa ter o prefixo "**luaopen_**" seguido do nome do próprio módulo. Para futuras referências, chamaremos essas funções especiais de *funções de inicialização*. Segue um exemplo de construção de um módulo chamado "**native**" que contém apenas a função **native_pow** que acamos de mostrar:

```
1   int luaopen_native (lua_State* L) {
2     // O argumento recebido eh o proprio nome do modulo, que
3     // no caso vamos ignorar
4     lua_settop(L, 0);
5     // O modulo serah formado por uma tabela com a funcao que queremos
6     lua_newtable(L);
7     lua_pushcfunction(L, native_pow);
8     // A pilha estah [tabela, funcao]. A chamada abaixo coloca a funcao
9     // na tabela usando a chave "pow"
10    lua_setfield(L, 1, "pow");
11    // O que sobra na pilha eh soh o proprio modulo, entao devolvemos ele
12    return 1;
13  }
```

Para *script*s poderem usar esse módulo, é preciso compilá-lo em uma biblioteca dinâmica[9] com o nome "**native.so**" em um Linux ou "**native.dll**" em um Windows. Depois é só fazer como antes:

```
1   require "native"
2
3   print ("2 ^ 10 = " .. native.pow(2,10))
```

Um último fato interessante é que a máquina virtual tem coleta automática de memória inutilizada. Diferentemente de implementações tradicionais, **Lua** não usa contagem de referências, mas sim verificação periódica de quais valores ainda são acessíveis. Basicamente, isso significa que não há problemas com referências cíclicas.

2.3.2 Python

Ao contrário de **Lua**, **Python** trabalha internamente com um *heap* (do inglês, "monte") para guardar seus objetos, porém raramente o usuário precisa mexer com ele diretamente. De fato, ao longo do desenvolvimento desse projeto nunca precisamos mexer nele.

A API **C** do **Python** é bem compreensiva e documentada, com muitas funções e macros para ajudar o usuário, nos diversos aspectos da interação com a máquina virtual, como conversão de valores, tratamento de objetos e de diversos protocolos que os objetos podem usar, como o protocolo de sequência (uma lista ou uma tupla em **Python**, por exemplo, usam esse protocolo) - facilitando o uso de sequências pelo código **C**.

Por exemplo, suponha o seguinte script **Python**:

```
1   # teste.py
2   def pow(x,y):
3       return x**y
```

Para executar a função **pow** em **C++** fazemos o seguinte:

```
1   #include <Python.h>
2
3   double executaPow(double x, double y) {
4       //primeiro importamos o modulo
5       PyObject* teste = PyImport_ImportModule("teste"); //new ref
6       if (teste == nullptr) {
7           // deu erro na importacao do modulo, e o python configura uma
```

[9] Bibliotecas dinâmicas são compiladas de maneira que o programa que as usa não precisa ser completamente ligado com elas na sua própria compilação. Em compensação, ele termina de ligar com elas quando é executado.

```
8             // excecao na API que pode ser analisada...
9             // entao aqui voce deve verificar essa excecao do python,
10            // e devolver um valor de erro ou soltar uma excecao de C++
11            return 0;
12        }
13
14        //gracas a facilidade da API, com uma unica chamada podemos
15        // 1. pegar uma funcao do modulo
16        // 2. construir uma tupla com os parametros a serem passados
17        // 3. executar a tal funcao, passando os parametros.
18        //e tambem eh possivel realizar essas operacoes separadamente,
19        //cada uma em uma chamada separada de funcao da API.
20        PyObject* result = PyObject_CallMethod(teste, "pow", "dd", x, y); //new ref
21        if (result == nullptr) {
22            // deu erro na chamada de funcao, o mesmo do caso teste==nullptr
23            // acima se aplica aqui.
24            return 0;
25        }
26
27        //convetermos o resultado para C
28        double resultado = PyFloat_AsDouble(result);
29
30        //corrigimos as contagens de referencia
31        Py_DECREF(teste);
32        Py_DECREF(result);
33
34        //e devlvemos o resultado
35        return resultado;
36    }
```

Como é possível notar no exemplo acima, outra facilidade da API é que ela usa um sistema primitivo de orientação a objetos em `C`, e quase tudo em `Python` (um módulo, uma classe, uma função, etc) é representado em `C` por um ponteiro para a estrutura `PyObject`, que além de informações sobre o tipo do objeto e outros valores, contém a contagem de referência dele. Quando a contagem de referências de um objeto se torna zero, a máquina virtual o remove da memória. E ai está um dos piores pontos da API `Python`: apesar de a máquina virtual gerenciar automaticamente as contagens de referência quando ela processa código em `Python`, o usuário que usar a API `C` tem que incrementar e decrementar manualmente as contagens dos objetos que ele está usando, o que é trabalhoso e propenso a erros.

Assim como em `Lua`, um *script* pode ser usado de duas formas pela máquina virtual: como uma sequência de operações a serem executadas ou como um módulo, contendo definições e operações que podem ser usadas por outros módulos. No entanto, a API `Python` não diferencia esses dois usos. Ao importar um *script* `Python`, ele será executado e quaisquer operações descritas nele serão realizadas, assim como quaisquer definições se tornarão acessíveis a quem importou o módulo. Por exemplo, o seguinte script:

```
1     from teste import pow
2
3     print "2 elevado a 3 eh igual a", pow(2,3)
```

Irá importar o *script* teste mostrado acima, que por si só não realiza nenhuma operação - ele simplesmente define a função `pow`. Após a importação, esse *script* não define nada, mas realiza uma operação: imprime texto no console, contendo a frase "2 elevado a 3 é igual a" seguido do valor devolvido pela execução de `pow(2,3)`.

Também é possível definir variáveis, funções e classes para serem usadas pelos *script*s através da API da máquina virtual. Usando a função `pow` como exemplo novamente, para implementá-la em `C` faríamos o seguinte:

```
1     #include <Python.h>
2     #include <cmath>
3
4     PyObject* native_pow (PyObject* self, PyObject* args) {
```

```
5         double x, y;
6         if (!PyArg_ParseTuple(args, "dd", &x, &y)) {
7             // - Ocorreu algum erro na conversao dos parametros.
8             // Devolver nullptr indica para a maquina virtual que ocorreu
9             // um erro nesta funcao, e nesse caso nao configuramos uma excecao pois
10            // a propria funcao PyArg_ParseTuple jah o faz quando ocorre um erro.
11            return nullptr;
12        }
13        // Objeto python devolvido deve ser sempre uma nova referencia
14        return Py_BuildValue("d", pow(x, y) );
15    }
```

A assinatura de **native_pow** é a assinatura de que funções em **C** precisam ter para serem aceitas pela API. Elas também são chamadas de **Python C Functions** ou **PyCFunction**. As formas mais comuns de permitir que essa função nativa seja chamada pelos *script*s são construir algum objeto que contenha ou execute essa função, ou definir um módulo pela API que o faça. De forma análoga a **Lua**, para criar um módulo desses também é necessário inicializá-lo com código **C**, o que também é feito com uma função de inicialização, que deve seguir a assinatura acima e ter o prefixo "**init**", seguido do nome do módulo. Então, para implementar o módulo **teste** que mostramos anteriormente dessa forma, estenderíamos o código do **native_pow** com o seguinte:

```
1     static PyMethodDef TesteMethods[] = {
2         {"pow", native_pow, METH_VARARGS, "pow(x,y) -> devolve x elevado a y."},
3         {NULL, NULL, 0, NULL}        /* sentinela */
4     };
5
6     PyMODINIT_FUNC initteste(void) {
7         Py_InitModule("teste", TesteMethods);
8     }
```

Para usar o módulo, caso você esteja incluindo a máquina virtual de **Python** em sua aplicação executável, basta compilar esse arquivo junto com seu programa, e chamar a função de inicialização **initteste** após inicializar a máquina virtual. Caso contrário, basta compilar o módulo em uma biblioteca dinâmica — mas não vamos entrar em detalhes sobre esse caso pois, para o escopo deste projeto, somente usamos módulos construídos usando a API em programas que estão incorporando as máquinas virtuais.

Capítulo 3

Estrutura do Projeto

No capítulo anterior revisamos e apresentamos conceitos visando responder a pergunta "Como integrar C++ com linguagens de *script*?". Agora que temos essas ferramentas a nosso dispor, vamos modelar uma solução. Para isso precisamos entender, em termos práticos, o que nosso sistema deve ser capaz de fazer, e então projetar uma estrutura que satisfaça esses requisitos.

Um usuário do nosso sistema estará tipicamente desenvolvendo uma aplicação em C++ que de alguma forma precisa ser capaz de interagir com **Lua** e **Python**, possivelmente ambas ao mesmo tempo. Como vimos na seção 2.2, essa interação pode ser estabelecida de maneira bastante direta entre elas se a máquina virtual da linguagem de *script* em questão tiver sido programada na linguagem compilada de interesse. Para evitar repetitividade e simplificar o texto, diremos, do ponto de vista das máquinas virtuais envolvidas, que:

Definição 3.1:

1. A **linguagem nativa** é a linguagem na qual a máquina virtual foi implementada.
2. A **linguagem virtual** é a linguagem que a máquina virtual processa para executar suas simulações.

Ou seja, C++ será a nossa linguagem nativa de interesse, enquanto que as linguagens virtuais serão **Lua** e **Python**, mas também poderiam ser quaisquer outras linguagens cujas máquinas virtuais estejam programadas em C ou C++.

3.1 Visão Geral

Para auxiliar a modelagem da estrutura, vamos dizer que tanto a aplicação quantos os *script*s estão divididos em **módulos**. Eles serão os objetos que representam o conjunto de elementos provenientes de uma ou de outra linguagem que podem ser acessados e manipulados. Por exemplo, se a aplicação em questão for um jogo de ação no qual o jogador enfrenta inimigos virtuais, o desenvolvedor poderia usar *script*s para implementar a inteligência artifical desse inimigos, para que fosse fácil ajustá-las sem ter que recompilar o jogo. Desse modo, cada um desses *script*s seria um módulo que a aplicação precisaria carregar e usar durante sua execução[1]. Eles, por sua vez, precisariam interagir com módulos da aplicação para que as inteligências artificiais conseguissem manipular seus personagens dentro do mundo virtual do jogo. Se elas quisessem saber o quão próximo o herói está, elas precisariam usar as funções do módulo de controle de posicionamento dos avatares. Se elas quisessem criar um projétil para atacar o jogador, elas precisariam acessar o módulo que contenha a classe[2] que representa o projétil desejado.

[1] Note como isso coincide com o conceito de "módulo" usado pelas APIs das linguagens de *script*.

[2] Classes são estruturas presentes em linguagens de programação *orientadas a objetos*. Seu papel é definir os dados e os comportamentos que um conjunto de objetos deve ter.

Figura 3.1

Esquematização bem simplificada do funcionamento sistema.

Dessa forma, uma primeira ilustração bem simplificada do funcionamento do nosso sistema é a representada na Figura 3.1. O sistema Ouroboros será responsável por fornecer acesso aos módulos entre a aplicação e as máquinas virtuais. Dessa forma, estabelecemos duas grandes funcionalidades no sistema: providenciar à aplicação a possibilidade de manipular os módulos das máquinas virtuais, e deixar os módulos da aplicação à disposição dos *script*s . Esses processos são conhecidos como **incorporação**[3] e **exportação**, respectivamente.

Definição 3.2:

1. **Incorporação** é quando a aplicação programada na linguagem nativa pode acessar e manipular os elementos dos módulos da máquina virtual.
2. **Exportação** é quando módulos da aplicação na linguagem nativa são registrados na máquina virtual de modo que os módulos desta possam usar funcionalidades daquela.

Nesse ponto vale à pena destacar um aspecto que ficou de lado até agora. Como o subtítulo desse trabalho indica, o propósito do sistema é não só integrar as ditas linguagens, como fazê-lo de maneira *automatizada*. Até porque as APIs das máquinas virtuais por sí sós já possuem seus próprios mecanismos de fornecer incorporação e exportação, indicando que nosso projeto seria redundante, não fosse o fato que eles são trabalhosos e específicos demais de usar. A intenção é que o usuário do nosso sistema não tenha que se preocupar com os detalhes de cada máquina virtual quando estiver desenvolvendo a aplicação dele e seja capaz de escrever normalmente os *script*s que trabalhem com ela — isso é, podendo acessar módulos externos através dos mecanismos usuais que a linguagem virtual fornece.

Logo, nosso sistema consistirá majoritariamente de uma biblioteca C++ que disponibiliza incorporação e exportação automatizada de e para *script*s. Faremos isso aproveitando as possibilidades de abstração que a orientação a objetos em C++ fornece para encapsular os comportamentos desejados das máquinas virtuais. As duas próximas sessões explicarão como projetamos as partes dessa biblioteca que atenderão cada um desses requisitos.

3.2 API unificada para incorporação de *script*s

Como vimos na seção 2.3, as máquinas virtuais com que estamos trabalhando possuem uma API própria através da qual podemos manipular seu conteúdo. Para evitar que o usuário acesse elas diretamente (e tenha que lidar com os mecanismos delas ele mesmo), podemos encapsulá-las

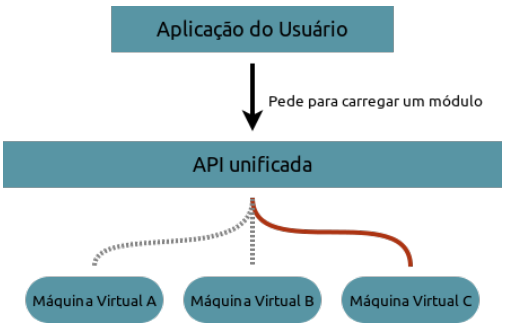
[3] Do inglês, "*embedding*".

por trás de uma camada que constitua uma nova API intermediária. Ela unificará o acesso às funcionalidades de todas as máquinas virtuais compatíveis. Mais especificamente, ela fornecerá os seguintes serviços:

1. Carregar um módulo a partir de um *script*;
2. Executar rotinas implementadas na linguagem virtual;
3. Acessar objetos definidos na linguagem virtual;
4. Converter valores entre tipos da linguagem nativa e tipos da linguagem virtual;

Idealmente, quando o usuário requisitar um desses serviços, ele não precisa saber qual máquina virtual realmente vai atendê-lo. Em termos práticos, isso significa que quando ele evocar a rotina que representa o serviço desejado, queremos que na verdade a rotina executada seja aquela que corresponda à máquina virtual apropriada "sem que ele perceba", como na figura 3.2. Ou seja, precisamos de funções com nomes e assinaturas[4] fixos simbolizando a operação almejada, porém sobrecarregadas com implementações circunstancialmente diferentes. Um jeito de obter esse efeito em C++ é através da herança de classes, que é o método que adotaremos. Basicamente, sempre que formos implementar uma funcionalidade que dependa do funcionamento das máquinas virtuais, definiremos uma classe abstrata[5] com as assinaturas desejadas e herdaremos ela em classes específicas para cada linguagem de *script*, que implementarão o que de fato acontece quando as funções da classe original são chamadas. Quando objetos dessas classes forem instanciados, os mecanismos de C++ garantirão que a implementação usada corresponderá à classe verdadeira deles (relativa a alguma máquina virtual), por mais que a aplicação do usuário só conheça a classe abstrata mãe.

Figura 3.2



Nesse exemplo, a aplicação do usuário pede para carregar um certo módulo script. *A API unificada deve discernir qual máquina virtual é a apropriada para tal (no caso, a máquina C) e delegar a tarefa, sem que o usuário precise saber.*

Vamos, então, basear-nos nos serviços listados acima para escolher a combinação de classes que a nossa API intermediária deverá oferecer. Uma maneira de fornecer acesso a objetos virtuais, como especifica o item 3, é encapsular seu comportamento em uma classe própria. Dessa forma, quando instâncias dessa classe fossem manipuladas pela aplicação do usuário, elas internamente executariam as operações necessárias na máquina virtual para que o efeito desejado ocorresse. E com isso ganhamos o item 4 de graça, pois basta colocar nessa classe métodos que convertam o valor

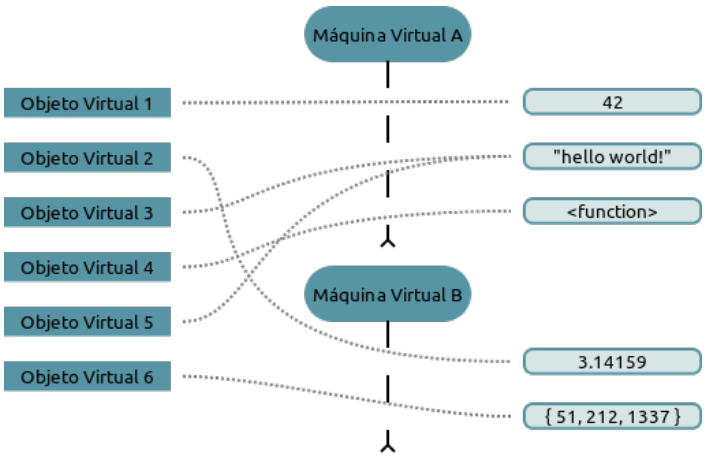
[4] A *assinatura* de uma rotina ou função especifica que tipos de parâmetros ela recebe e que tipos de valores ela devolve a quem a evocou.

[5] Em linguagens orientadas a objetos, *classes abstratas* são classes que não especificam a implementação de todas as operações dos objetos pertencentes ao conjunto que ela representa, determinando apenas a interface dessas operações. Outras classes deverão herdar dela para de fato implementar cada operação, e os objetos envolvidos pertencerão sempre a uma dessas classes filhas, mas poderão ser tratados como objetos da classe mãe também. Por outro lado, um objeto nunca pode pertencer *apenas* a uma classe abstrata, pois senão ele teria comportamentos indefinidos.

interno do objeto virtual para C++ e vice-versa. Por exemplo, o usuário poderia usar uma instância dessa classe para referir-se a uma cadeia de caracteres definida em um *script* para convertê-la para o tipo `string` em C++. O item 2 também é facilmente satisfeito pois, levando em consideração que nas linguagens de *script* com que estamos trabalhando funções são tipos de primeira classe[6], um objeto virtual também seria capaz de conter uma delas de forma que ao ser evocado desencadearia a rotina correspondente dentro da máquina virtual. A figura 3.3 exemplifica essas ideias.

Fica faltando apenas o item 1, com o qual podemos seguir o mesmo raciocínio que usamos com funções: podemos tratar módulos como objetos virtuais. Assim, quando o usuário pede para carregar um *script*, ele recebe uma instância de objeto virtual que abstrai o módulo em si. Isso exige que a classe tenha operações de acesso a membros dos objetos, para a aplicação poder alcançar os objetos virtuais que estão "dentro" do módulo. Para completar, a técnica de sobrecarga via herança nos permite criar implementações diferentes dessa classe para representar objetos virtuais de cada máquina e mesmo assim exigir que o usuário use apenas a classe base delas com as assinaturas de todas essas operações que vimos.

Figura 3.3



Cada instância da classe que representará objetos virtuais encapsula um valor dentro de alguma máquina virtual. Note que mais de uma instância pode ser responsável por um mesmo valor.

Por uma questão de projeto de código, são necessárias ainda mais duas classes. Afinal, quando o usuário quiser carregar um módulo para obter seu respectivo objeto virtual, que parte da nossa API ele deverá usar? A classe dos objetos virtuais não pode ser instanciada sozinha (até porque ela é abstrata). Normalmente, quem tem as informações necessárias para fazê-lo são as próprias máquinas virtuais. Então, uma solução seria definir uma classe para as máquinas virtuais também, novamente usando herança para abstrair o funcionamento delas ao mesmo tempo deixando que assinaturas relevantes fiquem expostas. A terceira classe seria justamente para gerenciar as instâncias de máquinas virtuais. As relações entre essas três classes e o usuário estão representadas na figura 3.4.

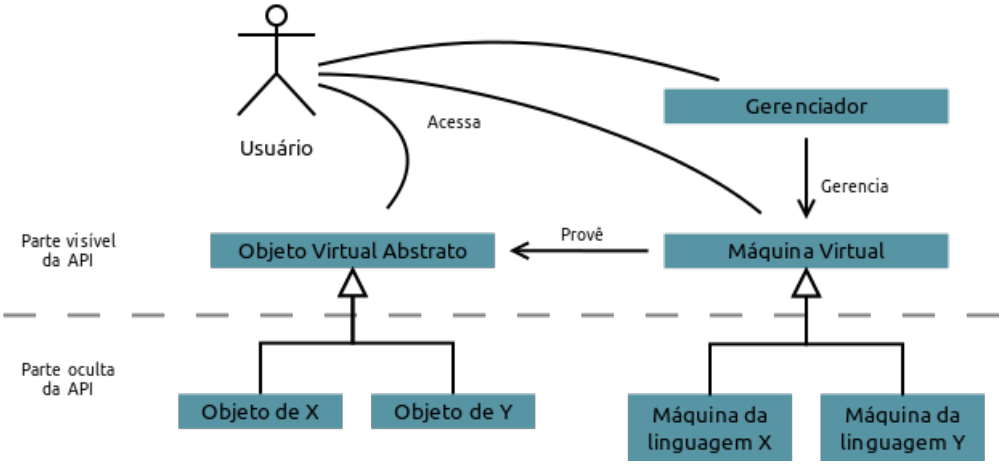
Chamamos a parte do nosso sistema que corresponde a essa API unificada de ***Ouroboros Project API*** ou simplesmente **OPA**.

3.3 Gerador de *wrappers* para exportação de interfaces nativas

Para entender melhor como fazer a exportação, precisamos saber o que exportar. No caso, como as aplicações dos usuários estarão escritas em C ou C++, os elementos que gostaríamos de exportar

[6] Quando funções são tipos de *primeira classe*, elas podem ser tratadas como se fossem um valor, o que permite que elas possam ser armazenadas em variáveis, passadas como argumento para outras funções e até mesmo devolvidas como resultado de outras funções também.

Figura 3.4



Isso não chega a ser uma diagrama UML, apenas uma representação simbólica das classes. A especificação mais detalhada delas será exposta no capítulo 4. Mas emprestamos a notação de herança.

são:

1. Variáveis globais.
2. Funções globais.
3. Tipos novos definidos na aplicação (usando `typedef`, classes ou equivalentes)
4. Outras informações, como *namespaces*[7].

No terceiro item, o ideal é que todas as operações inerentes ao tipo também sejam exportadas. Isso englobaria tanto operações aritméticas básicas quanto os métodos de classes, por exemplo. O usuário deverá ser capaz de criar objetos desses tipos em seus *script*s e usá-los da maneira mais fiel possível ao modo como eles seriam usados no código nativo da aplicação.

O que explicamos nas sessões 2.3.1 e 2.3.2 nos fornece algumas ferramentas úteis, apesar de trabalhosas, para exportar esses elementos que listamos. Basicamente, podemos usar os mecanismos de registro de módulos nas máquinas virtuais para expor os módulos reais da aplicação[8]. Aproveitando o exemplo que demos na seção 3.1, poderia haver um módulo na aplicação (no caso, um jogo) que cuidasse do posicionamento de avatares, e nós gostaríamos que *script*s incorporados pudessem acessar as funcionalidades desse módulo. Todas as variáveis, funções e tipos relacionados teriam que ser colocados no módulo virtual através das funções de inicialização, que precisam ser feitas usando o protocolo esperado pelas APIs.

Isso compõe um método de exportação que funciona, mas traz dificuldades. A maior delas é que C e C++ não oferecem um meio de definir funções novas em tempo de execução, impedindo nossa biblioteca de registrar módulos virtuais durante o processo de suas rotinas, pois as funções de inicialização já precisam estar criadas quando a aplicação do usuário for ser compilada. A outra grande dificuldade é que é igualmente impossível de analisar o código da aplicação durante a execução dela mesma dadas as linguagens compiladas que adotamos. Normalmente, se um desenvolvedor quisesse

[7] *Namespaces* (do inglês, "espaços de nomes") são estruturas em C++ que separam os elementos de uma aplicação em espaços distintos, facilitando a organização e expressividade do código, assim como evitando possíveis conflitos de nomes iguais usados em contextos diversos.

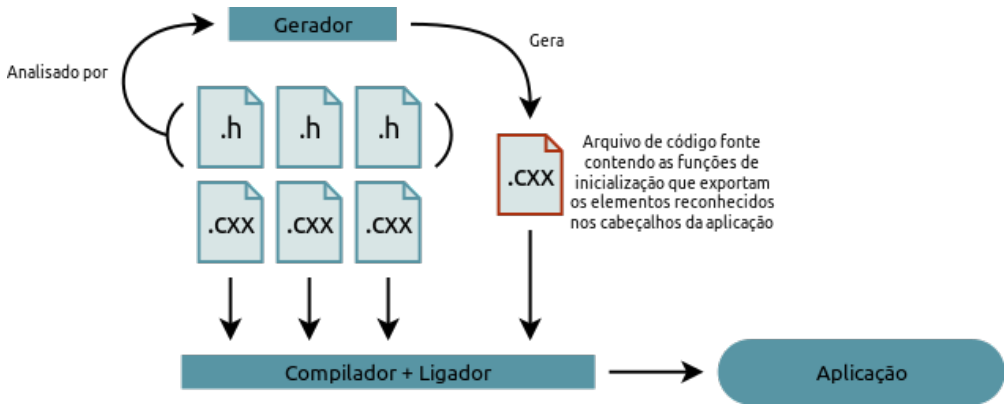
[8] Para não confundir a qual tipo de módulo estaremos nos referindo, chamaremos os módulos que registramos nas máquinas virtuais de *módulos virtuais*, enquanto que os da aplicação serão simplesmente *módulos*.

exportar seu código nativo para uma linguagem de *script*, ele simplesmente escreveria as funções de inicialização e manualmente colocaria cada elemento do código dele nos módulos exportados. Mas o que estamos tentando projetar aqui visa justamente *automatizar* a integração entre aplicação e *script*s, então simplesmente pedir para o usuário fazer esse trabalho sujo para nós está fora de questão. Assim sendo, não temos como saber quais variáveis, funções e tipos pertencem a um módulo e, mesmo que soubéssemos, não temos como fornecer as funções de inicialização necessárias quando a aplicação do usuário for executada e esse serviço for requerido da nossa biblioteca.

Um jeito de contornar esses problemas é *não* tentar fazer a exportação durante a execução da aplicação, mas sim antes de ela ser compilada, de maneira que tenhamos acesso ao seu código fonte. A partir dele somos capazes de reconhecer as declarações que queremos importar, e podemos escrever o código adicional das funções de inicialização a tempo de elas serem compiladas junto com o resto da aplicação. Isso seria feito por um outro programa — que obviamente precisa estar compilado antes da aplicação, uma vez que ele *interfere* com o código dela — que chamamos de *gerador de código*. Na prática, as únicas partes do código da aplicação que ele precisa analisar são os cabeçalhos, que é onde estarão as declarações. Uma visão geral desse procedimento está ilustrada na figura 3.5.

No entanto, ainda sobram algumas outras complicações porque, em geral, elementos nativos não podem ser exportados diretamente. Isso fica bastante claro quando olhamos para o modo como funções nativas são passadas para as máquinas virtuais: elas precisam ter uma assinatura específica, que ainda por cima varia de máquina para máquina. E fica ainda mais complicado quando pensamos em classes, pois em **Lua**, por exemplo, não há orientação a objetos implementada por padrão — embora seja possível construir mecanismos na linguagem que funcionem como classes, que é o que precisamos fazer.

Figura 3.5



Representação de como o gerador conribui na exportação. Em **C++**, *arquivos de cabeçalho usam a extensão* **.h** *enquanto que arquivos de implementação usam* **.cxx**, *embora haja outras conveções.*

Então, para cada tipo de elemento nativo que estamos interessados (variáveis, funções, tipos, etc), precisamos de uma maneira de adaptá-los para o formato que as máquinas virtuais esperam. Isso é possível através do uso de *wrappers* (do inglês, "embrulhos"), que nesse contexto são porções de código que encapsulam as atividades nativas da aplicação para expô-los adequadamente aos protocolos que as APIs das máquinas virtuais exigem para exportação. Assim, o que realmente estaremos inserindo nos módulos virtuais serão esses *wrappers*, e não os elementos originais. Entraremos em mais detalhes sobre como fazer esses *wrappers* no capítulo 4. Por enquanto, o importante é entender que essa parte do nosso sistema envolverá um gerador de código que produzirá os *wrappers* que ele julgar necessário dado o código fonte da aplicação do usuário, e os inserirá em módulos virtuais de maneira a efetivamente exportar as funcionalidades nativas para as máquinas virtuais.

Isso tudo significa que no final das contas nosso sistema não será apenas uma biblioteca, pois ele também terá executáveis fazendo o papel de geradores, um para cada linguagem de *script*. O que fizemos para evitar dividir nosso código em muitas partes foi limitar esses programas auxiliares a simplesmente evocarem algumas rotinas da nossa biblioteca que de fato farão a análise e a geração de código. Chamamos essa parte do nosso sistema de *Ouroboros Project Wrapper and Interface Generator* (OPWIG).

Capítulo 4

Atividades Realizadas

O objetivo desse capítulo é relatar as diversas atividades realizadas no planejamento e desenvolvimento do Projeto Ouroboros. Explicaremos como de fato implementamos a solução apresentada no capítulo anterior, as dificuldades técnicas encontradas e todos os outros recursos que utilizamos para concretizar nossa ideia.

Porém, a modelagem que fizemos até agora reflete o modo como nós enxergamos *atualmente* nosso projeto. Como ressaltamos na introdução, já havíamos desenvolvido boa parte do nosso sistema bem antes do trabalho de formatura, e ele passou por algumas modificações desde então. Logo, como este capítulo descreverá as atividades que fizemos, ele adotará um fluxo mais cronológico dos assuntos, descrevendo o que fizemos desde o começo do sistema. Por isso, haverá momentos em que o que fizemos ainda não condizirá com a estrutura apresentada no capítulo 3, mas ilustrará o processo pelo qual passamos antes de alcançar o nosso entendimento atual do projeto e do domínio onde ele exerce seu papel.

4.1 Origem do projeto

No início deste ano, elegemos para o nosso trabalho de formatura supervisionado o aprimoramento de um sistema que já estávamos desenvolvendo desde o começo de 2011, como parte das nossas atividades no USPGameDev. Originalmente, ele fazia parte de um outro projeto do grupo chamado UGDK[1], uma *engine*[2] de desenvolvimento de jogos digitais bidimensionais (ou simplesmente "jogos 2D"). A ideia surgiu devido a uma situação com outro projeto, também do USPGameDev: o jogo digital chamado *Horus Eye*[3]. Ele era programado em C++ e usava os recursos da UGDK para ter acesso às funcionalidade essenciais de mídia e interatividade que um jogo precisa. A equipe estava tendo muitas dificuldades em expandir o conteúdo dele desde o seu lançamento em outubro de 2010, pois não só apenas os membros mais antigos sabiam programar em C++, como também só eles sabiam *o que* precisava ser mudado para obter os resultados desejados. A solução que propusemos foi integrar o jogo com *script*s, pois assim até mesmo não programadores poderiam contribuir com o desenvolvimento de mais conteúdo para ele. Foi então que elaboramos o chamado "sistema de *script*s da UGDK", que depois viria a se tornar o objeto deste trabalho.

Ele fornecia ferramentas que buscavam simplificar o máximo possível a troca de dados entre o jogo e os *script*s. Para que a elaboração de conteúdo novo do jogo pudesse ser programada majoritariamente através destes, bastava isolar as partes mais críticas daquele. Por exemplo, para um desenvolvedor de conteúdo não interessa saber quais rotinas devem ser evocadas para criar uma nova fase, mas sim como descrever as fases; então, deixávamos os *script*s responsáveis por especificar a forma e os eventos de cada fase, e depois o código nativo do jogo cuidava de analisar esses dados e executar os procedimentos necessários para que a fase passasse a existir. E dessa maneira tornamos

[1] A UGDK é um dos principais projetos do USPGameDev, e seu nome é um acrônimo que pode significar tanto *USP Game Development Kit* quanto *USPGameDev Kit*. Ela é desenvolvida em C++, disponível em: http://uspgamedev.org/projetos/ugdk/ (último acesso: 26/11/2013)

[2] Mais informações: http://www.gamecareerguide.com/features/529/what_is_a_game_.php (último acesso: 26/11/2013)

[3] Disponível em: uspgamedev.org/horus-eye (último acesso: 25/07/2013)

possível aprimorar o jogo usando linguagens bem mais simples que C++ e sem a necessidade de conhecer todas as minúcias do seu código.

Figura 4.1

Uma fase feita para o Horus Eye *usando o sistema de* script*s da UGDK. É possível posicionar e especificar tanto elementos estáticos como vasos quanto elementos interativos como as portas e as múmias, graças às facilidades que* scripting *traz ao desenvolvimento.*

Comparando com a estrutura apresentada no capítulo anterior, o que tínhamos desenvolvido é o que hoje corresponde à OPA. A parte pela qual o OPWIG seria responsável era tratada por uma ferramenta à parte, sobre a qual falaremos ao longo das sessões que seguem. Ela é um dos principais motivos pelo qual, apesar de termos sido bem sucedidos em desenvolver esse sistema, ele ainda era razoavelmente delicado de manipular e continha várias limitações.

4.2 Decisões de projeto

Antes de adentrarmos as atividades de implementação, gostaríamos de apresentar as decisões que adotamos para o trabalho em nosso projeto. Principalmente porque elas também sofreram alterações ao longo do tempo, refletindo bastante no que tivemos que fazer ao longo do desenvolvimento. Trataremos tanto das ferramentas que optamos por usar, quanto das metodologias e convenções que escolhemos.

4.2.1 Decisões anteriores ao Trabalho de Formatura

Quando começamos a desenvolver o nosso "sistema de *script*s", como ele era integrado à UGDK e fazia parte do USPGameDev, muitas das nossas decisões do projeto refletiam os padrões de projeto que seguíamos no grupo. Segue uma relação das decisões tomadas sobre ferramentas e metodologias usadas. Quando necessário, daremos um resumo sobre as ferramentas apresentadas.

1. **Linguagens usadas**: C++ para a biblioteca, **Lua** e **Python** para integração.

Motivo - C++ era a linguagem que a UGDK e os nossos jogos usavam. Além disso é comum linguagens de *script* terem uma implementação de sua máquina virtual em C, ou até C++. Ainda mais, como C++ contém em si todas as funcionalidades de C, não temos perda técnica nenhuma em usar o primeiro ao invés do segundo. Também decidimos que o

sistema seria generalizado para poder funcionar com possivelmente qualquer linguagem de *script*, mas inicialmente iríamos incluir apenas compatibilidade com **Lua** (a qual o Wilson conhece bem) e **Python** (usada regularmente pelo Fernando).

2. **Compiladores**: g++[4] em Linux e Visual Studio[5] 2010 em Windows.

Sobre - O g++ é o compilador para C++ do projeto GNU, e o Visual Studio é um ambiente de desenvolvimento para C++ e algumas outras linguagens feito pela Microsoft para Windows, contendo uma IDE[6], compilador e outras ferramentas.

Motivo - No USPGameDev, cada membro é livre para usar o compilador que achar melhor, e os projetos evitam usar funcionalidades específicas que dificultem essa portabilidade.

3. **Compatibilidade**: Multi-plataforma com CMake[7].

Sobre - *Software* livre para gerenciar a compilação, os testes e a entrega de aplicações multi-plataforma.

Motivo - Para simplificar a compilação em diversas plataformas e compiladores. Além disso, apesar do nosso sistema fazer parte da UGDK, o usuário da *engine* podia configurar ela para ser compilada com essa parte dela desativada (caso ele não precise integrar com *script*s, por exemplo). Isso era feito através da manipulação de variáveis de configuração na descrição de projeto em CMake da UGDK.

4. **Distribuição e versionamento**: Usando um repositório Git[8] hospedado no GitHub[9].

Sobre - Git é um *Software* Livre para versionamento cooperativo de código fonte. É possível configurar seu próprio servidor ou recorrer a um pronto. No caso, o GitHub fornece esse serviço de hospedagem de repositórios Git.

Motivo - O grupo gosta de deixar as ferramentas que desenvolve à disposição da comunidade, para quem quiser aprender com elas. Como nosso sistema fazia parte do código da UGDK, ele compartilhava o repositório dela.

5. **Geração de *wrappers***: SWIG [swi].

Sobre - Por extenso, seu nome é *Simplified Wrapper and Interface Generator*, e ele essencialmente exporta código nativo para diversas linguagens virtuais, inclusive as que usamos. Iremos explicar bastante sobre como ele é usado algumas seções a frente.

Motivo - Foi a melhor opção para o que necessitávamos segundo nossa pesquisa, era uma ferramenta pronta para proporcionar a geração de *wrappers* que sabíamos que seria difícil de desenvolver nós mesmos a tempo de atender a demanda para o sistema no USPGameDev.

6. **Metodologia**: Uma variação própria de métodos ágeis do USPGameDev.

Sobre - O trabalho de formatura de Vinicius K. Daros [Dar11] trata com bastante detalhes sobre as metodologias de trabalho do USPGameDev naquela época.

Motivo - As metas que seguíamos nas iterações de trabalho estavam mescladas com as metas do jogo *Horus Eye* e da *engine* UGDK, de forma que a maior parte do que desenvolvíamos era voltado para ser usado nesses outros programas. Logo, trabalhávamos em conjunto com o resto do grupo usando a mesma metodologia.

[4] Disponível em: http://gcc.gnu.org (último acesso: 28/11/2013)
[5] Mais informações: http://www.visualstudio.com (último acesso: 22/11/2013)
[6] *Ambiente de Desenvolvimento Integrado*, do inglês, "*Integrated Development Environment*"
[7] Disponível em: http://www.cmake.org (último acesso: 22/11/2013)
[8] Disponível em: http://git-scm.com/ (último acesso em: 22/11/2013)
[9] Acessível em: https://github.com/ (último acesso: 22/11/2013)

4.2.2 Mudanças no Projeto para o Trabalho de Formatura

Quando começamos o trabalho de formatura, fizemos algumas mudanças nas nossas decisões de projeto originais. Primeiramente, separamos o sistema para ser um projeto separado, totalmente independente de qualquer outro projeto do USPGameDev, e o nomeamos Projeto Ouroboros. Partindo do código fonte original, criamos outro repositório Git específico para o Ouroboros[10], e também o hospedamos no GitHub. Mantivemos o uso de CMake, pois, dentre outras razões, ele permitiu-nos simplificar algumas partes do nosso sistema tanto para os desenvolvedores quanto para os usuários (mais informações na seção 4.7). Abaixo listamos as principais mudanças feitas com relação às outras decisões anteriores, assim como algumas novas que surgiram com a emancipação do nosso sistema.

1. **Linguagens usadas**: As mesmas, só que com um padrão mais recente de C++ conhecido como C++11.

Sobre - Nesse novo padrão, inúmeras facilidades são introduzidas à linguagem. As mais relevantes para nosso projeto foram as funcionalidades de coleta de lixo automatizada e os *variadic templates*.

Motivo - Além da diversas facilidades que tornaram o desenvolvimento do sistema mais produtivo, algumas das ferramentas que passamos a usar já adotavam esse novo padrão, e portanto achamos melhor acompanhá-las nessa iniciativa para evitar incompatibilidades.

2. **Compilador**: Apenas g++.

Motivo - O uso de C++11 introduziu um problema no projeto: compatibilidade de compiladores com essa versão nova. Como o C++11 ainda é razoavelmente recente, nem todos compiladores reconhecem todas suas novas funcionalidades. Mais notavelmente, não conseguimos mais compilar no Windows com o Visual Studio 2010, e nem com sua versão mais recente, Visual Studio 2013. No Linux, tivemos que usar uma versão mais nova do g++, a 4.7. Apesar de existirem versões ainda mais recentes, essa era a mínima necessária.

3. **Gerador de *wrappers***: Abandonamos o uso de SWIG para substituí-lo pelo nosso próprio gerador, o OPWIG.

Sobre - Já explicamos sobre ele na seção 3.3 e falaremos sobre sua implementação na seção 4.6.

Motivo - Também trataremos dos motivos por trás desse mudança em outra seção, a 4.5.

4. **Metodologia**: Organização das metas em *milestones*.

Sobre - Após separar o projeto de USPGameDev, inicialmente mantivemos listas de tarefas pendentes e fixávamos objetivos periodicamente para serem completados o quanto antes (algo vagamente parecido com iterações de programação ágil, porém não tão bem organizado). A partir do segundo semestre desse ano, adotamos o uso de *milestones*[11] (ou "marcos", em português): elaborávamos uma pequena aplicação de um usuário fictício do nosso sistema, e desenvolvíamos apenas o necessário para que a aplicação dele funcionasse devidamente. Após completar uma *milestone*, montávamos uma outra mais complicada e que exigisse mais do nosso projeto, e repetíamos o processo. Além disso, recentemente passamos a aproveitar algumas funcionalidades do GitHub justamente voltadas para o gerenciamento de tarefas que caíram muito bem com essa metodologia.

Motivo - Ao estabelecer uma meta tão clara quanto um caso de uso, o desenvolvimento fica muito mais produtivo e objetivo. Durante o primeiro semestre gastamos bastante tempo cobrindo partes do nosso sistema que até agora não foram completamente utilizadas. Se

[10] Repositório do Projeto Ouroboros: https://github.com/Rewasvat/ouroboros (último acesso: 1/12/2013)

[11] O uso que damos à palavra *milestone* nesse trabalho provavelmente não reflete exatamente o sentido usual que ela tem no desenvolvimento de *software*.

esse esforço tivesse sido melhor direcionado, provavelmente o projeto estaria mais adiantado. O método das *milesones* ainda por cima é bastante compatível com os princípios de programação ágil, pois provê um processo iterativo que prioriza o desenvolvimento das funcionalidades mais críticas e relevantes ao produto.

5. **Testes automatiazdos**: Usando *googletest*[12] e fazendo integração contínua com Travis[13].

Sobre - O *googletest* é um arcabouço para escrever e executar testes automatizados em C++, desenvolvido pela Google. Já o Travis é um serviço de integração contínua: ele automaticamente clona respósitórios no GitHub quando há alguma modificação neles, e executa uma sequência de comandos (determinadas pelo dono de cada repositório) para verificar se o código do projeto compila com sucesso e passa nos testes especificados para ele. Para usar o Travis basta registrar o repositório desejado usando sua própria conta do GitHub, e ele irá inclusive enviar e-mails para os desenvolvedores sempre que algo falhar nos testes.

Motivo - Como passamos a desenvolver nosso próprio gerador de *wrappers*, precisávamos de uma segurança melhor sobre o funcionamento dele. Usar testes automatizados permitiu-nos desenvolver com muito mais tranquilidade, certeza e produtividade. Ao mesmo tempo, o uso do Travis permite que testemos nosso *software* em um ambiente que não seja o nosso (que já está todo preparado e portanto seria uma referência viesada), assim como saber imediatamente quando algo parou de funcionar graças às notificações que ele nos envia.

4.3 Uma interface comum para Lua e Python.

Desenvolver uma interface comum entre as duas linguagens de *script* escolhidas foi uma das coisas que fizemos para a UGDK, e hoje o trabalho realizado nesse quesito foi o que formou a OPA vista na seção 3.2. Desde o começo vimos desenvolvendo uma interface generalizada capaz de usar uma ou outra máquina virtual sem diferenças na interface para o usuário. A seguir explicamos o que implementamos de fato nas principais classes da OPA.

Como vimos no capítulo anterior, a princípio precisaríamos de três classes na nossa API: uma para representar objetos virtuais, outra para máquinas virtuais e a última para gerenciar essas máquinas. Mas na realidade, por questões de usabilidade, implementamos a classe de objetos virtuais em duas partes. Uma delas é a que o usuário utilizará, e portanto terá vários mecanismos facilitadores. A outra será a que de fato fará o trabalho sujo, escondido do usuário graças à primeira. Elas são as classes `VirtualObj` e `VirtualData`, respectivamente.

As das máquinas virtuais e do gerenciador serão mantidas como projetamos. No entanto, o usuário não usará a de máquinas virtuais com tanta frequência, já a do gerenciador será muito mais presente e importante para ele. Principalmente porque o gerenciador encapsula os serviços das máquinas virtuais, então o acesso direto a elas torna-se desnecessário.

A seguir entramos em detalhes sobre cada uma dessas classes.

4.3.1 `VirtualObj`

Essa classe representa um objeto virtual qualquer, como explicado na seção 3.2. É quase a única classe que o usuário precisa usar. Ela implementa alguns métodos que possibilitam ao usuário realizar sobre o tal objeto operações comuns em elementos de diversas linguagens de *script*. Para tornar o uso dessa classe mais agradável, também sobreescrevemos alguns operadores de C++ de maneira a facilitar uso de algumas dessas operações:

- **Executar o objeto como uma função**: simplificado pelo operador `()` do C++, permitindo ao usuário usar as próprias instâncias de `VirtualObj` como funções. Exemplo:

[12] Disponpivel em: https://code.google.com/p/googletest/ (último acesso: 29/11/2013)
[13] Acessível em: https://travis-ci.org/ (último acesso: 29/11/2013)

```
1         VirtualObj vObj, vReturnObj, vObjArg1, ..., vObjArgN;
2         VirtualObj::List vObjList; // o mesmo que std::list<VirtualObj> vObjList;
3                                    // uma lista de instancias de VirtualObj.
4
5         // Podemos evocar o objeto sem parametro nenhum
6         vReturnObj = vObj();
7         // Com um numero arbitrario deles
8         vReturnObj = vObj( vObjArg1, vObjArg2, ... , vObjArgN );
9         // Ou usando uma lista para junta-los
10        vReturnObj = vObj( vObjList );
```

- **Acessar e alterar atributos do objeto**: simplificado pelo operador `[]` do C++, permitindo ao usuário usar as instâncias de `VirtualObj` como tabelas de símbolos. Exemplo:

```
1         VirtualObj vObj, vAttrObj, vObj_AttrName;
2         // Podemos acessar membros usando cadeias de caracteres como chave
3         vAttrObj = vObj["nome_do_atributo"];
4         // Ou outros objetos virtuais
5         vAttrObj = vObj[ vObj_AttrName ];
```

- **Executar um método do objeto (instância de uma classe)**: simplificado pelo operador `|` do C++. Chamadas de métodos em linguagens de *script* normalmente passam a instância da classe (o próprio objeto) como o primeiro argumento da função, e o uso desse operador com o objeto simplifica essa operação. Exemplo:

```
1         // Recebe argumentos como nas chamadas de funcao
2         (vObj | "nome_do_metodo")( ... );
3         // E eh equivalente a
4         vObj["nome_do_metodo"](vObj, ...);
```

- **Conversão de valores**: como a maioria dos métodos (e operadores) da classe `VirtualObj` recebem e devolvem instâncias dela mesma, a conversão de valores é a forma que fizemos para poder converter entre objetos de C++ e esses objetos virtuais. Para tanto, a classe possui dois métodos *template*[14] que possibilitam a conversão de valores entre os dois ambientes. Exemplo:

```
1         // Com ponteiros para tipos definidos pelo usuario:
2         T* valor = vObj.value<T*>();
3         vObj.set_value<T*>(valor);
4
5         VirtualObj vNumObj; //suponha um VirtualObj contendo um numero
6         double num = vNumObj.value<double>(); // pegar o valor
7         vNumObj.set_value<double>(num);       // atribuir o valor
```

Na prática, `VirtualObj` é uma classe que encapsula instâncias da classe `VirtualData`, que veremos adiante, na seção 4.3.4. Todas essas operações são repassadas para a instância dessa última usando um padrão de código chamado **ponteiro para implementação**[15]. A classe `VirtualData` não deve ser usada diretamente pelo usuário.

[14] *Templates* são uma funcionalidade de C++ que permite definir parametricamente funções e classes, fazendo-as exigir um ou mais tipos em uma notação especial quando são usadas. Com isso, essas funções ou classes têm seus corpos escritos de forma generalizada para vários parâmetros possíveis. *Templates* também podem receber valores aritméticos e cadeias de caracteres, mas na interface de `VirtualObj`, e na maioria do nosso trabalho, somente usamos com tipos. Mais informações em: http://www.cplusplus.com/doc/tutorial/templates/ (último acesso: 26/11/2013)

[15] Vide http://en.wikibooks.org/wiki/C++_Programming/Idioms#Pointer_To_Implementation_.28pImpl.29 (último acesso: 15/09/2013)

4.3.2 ScriptManager

Essa é a segunda principal classe da interface comum entre linguagens, e praticamente a única outra classe que o usuário precisa usar além da `VirtualObj`. Ela segue o padrão de projeto ***singleton***[16] e corresponde ao gerenciador visto na seção 3.2. Sua responsabilidade é, portanto, gerenciar as máquinas virtuais que estão disponíveis no sistema, permitindo ao usuário executar algumas operações simples porém importantes nelas como um conjunto:

- **Inicialização e Finalização**: o gerenciador tem métodos para inicializar e finalizar o conjunto de máquinas virtuais. A inicialização é necessária para poder usá-las e também serve para especificar o diretório na qual *script*s serão carregados por ele. A finalização encerra a atividade das máquinas, liberando quaisquer recursos do computador que elas estejam usando. Exemplo:

```
1        // SCRIPT_MANAGER() eh uma macro que fornece um ponteiro para a
2        // instancia de ScriptManager.
3        bool ok = SCRIPT_MANAGER()->Initialize("./caminho/para/os/scripts/");
4
5        SCRIPT_MANAGER()->Finalize();
```

  Se o valor booleano que a chamada a `Initialize()` devolve for verdadeiro, significa que o gerenciador foi inicializado sem problemas. Caso contrário, deve ter ocorrido algum problema na inicialização dele.

- **Registro e busca de máquinas virtuais**: o gerenciador inicialmente não sabe quais são as máquinas virtuais disponíveis. O usuário pode tanto registrá-las manualmente com um simples método, quanto delegar essa responsabilidade para o código gerado pelo OPWIG, como veremos mais adiante. Caso ele queira, o usuário também pode buscar uma máquina virtual registrada. Elas são representadas no nosso sistema pela classe `VirtualMachine`, como veremos na próxima seção. Exemplo:

```
1        //Registrando uma maquina virtual.
2        VirtualMachine* vm = new python::PythonMachine();
3        SCRIPT_MANAGER()->Register(vm);
4
5        //Buscando uma maquina virtual.
6        VirtualMachine* pyvm = SCRIPT_MANAGER()->GetMachine("Python");
7        //pyvm corresponde ao mesmo vm declarado antes.
```

- **Executar código de *script***: o gerenciador permite ao usuário tentar executar um código qualquer, dado como uma cadeia de caracteres, na máquina virtual de sua escolha. Exemplo:

```
1        SCRIPT_MANAGER()->ExecuteCode("Lua", "print(42)");
```

- **Carregar módulos**: possivelmente o método mais importante da classe `ScriptManager`, este método recebe um caminho para um arquivo de *script* (sem a extensão dele). Ele então determina se o *script* existe, e de acordo com a extensão do arquivo, qual máquina virtual deve processá-lo. A tal máquina virtual então carrega o *script* e a função devolve uma instância de `VirtualObj` representando o módulo carregado de acordo com a linguagem de *script* usada. Exemplo:

```
1        VirtualObj modulo = SCRIPT_MANAGER()->LoadModule("modulo");
2        VirtualObj modulo2 = SCRIPT_MANAGER()->LoadModule("pacote.subpacote.modulo");
```

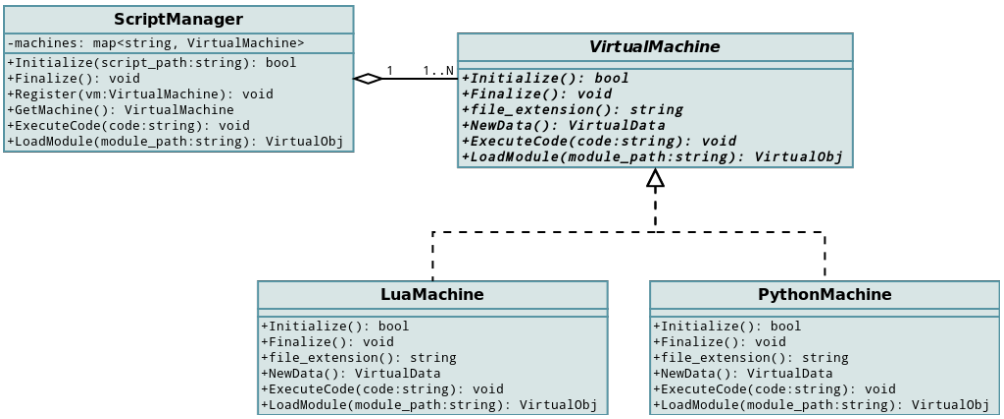
[16] *Singleton* é um padrão de projeto que permite ter apenas uma instância de uma classe num dado momento. Mais informações em [GHJV95].

4.3.3 `VirtualMachine`

A `VirtualMachine` é uma das duas classes abstratas do sistema que devem ser implementadas para cada linguagem que se deseja ser compatível com a OPA, como explicado na seção 3.2 e ilustrado pela figura 3.4. A `VirtualMachine` representa a máquina virtual da linguagem em si, sendo usada internamente pelo gerenciador, e portanto ela raramente ela será usada diretamente pelo usuário. A figura 4.2 contém o diagrama UML que expressa a relação entre essas duas classes e também as duas implementações de máquina virtual padrão do nosso sistema. Alguns métodos importantes de sua interface são:

- Inicialização e Finalização da máquina virtual, evocadas pelo gerenciador no momento adequado.
- Método para retornar a cadeia de caracteres correspondente à extensão de um arquivo de *script* dessa linguagem.
- Carregar módulos, que é a operação usada internamente pelo gerenciador.
- Executar código *script*, também usado pelo gerenciador.
- Fornecer uma nova instância vazia de `VirtualData` (ver próxima seção) correspondente à linguagem da máquina.

Figura 4.2

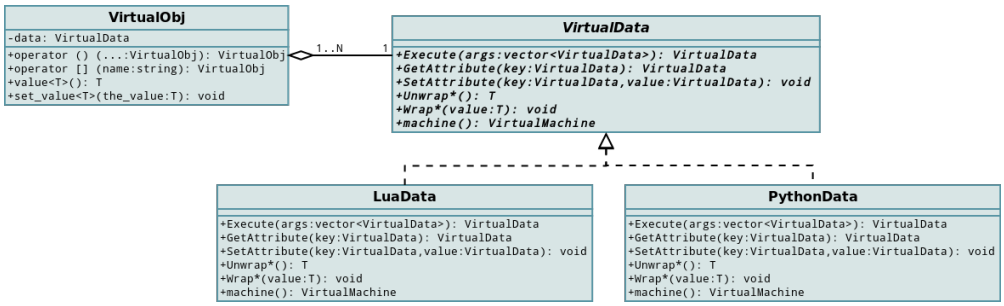


Os métodos apresentados correspondem às operações que destacamos para cada classe. A instância única de `ScriptManager` *armazena uma tabela que relaciona os nomes das linguagens de* script *com a instância de máquina virtual correspondente.*

4.3.4 `VirtualData`

`VirtualData` é a segunda classe abstrata do sistema que deve ser implementada para uma linguagem de *script* poder ser usada pela OPA. Ela provê os mecanismos essenciais dos objetos virtuais de uma linguagem específica, fornecendo todas aquelas operações da classe `VirtualObj` (descritas anteriormente). Assim, uma instância de `VirtualObj` pode ter uma instância de `VirtualData` implementada para uma linguagem ou outra sem precisar saber qual, graças ao polimorfismo através de herança. Essa relação está ilustrada no diagrama UML da figura 4.3. Fora os métodos para as operações descritas na subseção 4.3.1, um ponto importante da interface de `VirtualData` é um método para fornecer a máquina virual (contida no gerenciador) responsável pela instância de `VirtualData` em questão.

Figura 4.3



Os métodos `Wrap` *e* `Unwrap` *mostrados na verdade correspondem a uma série de métodos distintos, cada um responsável pela conversão entre valores nativos e virtuais de um tipo específico. Optamos por não indicar todos eles para simplificar o diagrama.*

4.4 Integração com SWIG

O SWIG é uma ferramenta para exportar uma interface C/C++ para diversas linguagens de *script*. Um dos usos padrão dessa ferramenta (e a forma como usamos) é gerar o código de *wrapper* necessário para criar módulos virtuais para uma máquina virtual. E é isso que possibilita a linguagem virtual correspondente ter acesso a uma interface desenvolvida em C++.

A princípio, o SWIG servia para complementar nosso sistema, visto que tínhamos as versões antigas da OPA para permitir que a aplicação do usuário incorpore *script*s mas não havíamos feito nada a respeito da exportação de funcionalidades da aplicação para os *script*s. Após pesquisar possíveis soluções o que encontramos foi o SWIG, uma ferramenta já um tanto quanto antiga mas ainda usada na indústria e que fazia basicamente o que queríamos (o processo ilustrado na figura 3.5).

Como o SWIG é uma ferramenta já pronta, e que é executada para gerar código que será compilado com o código do usuário, a interação entre ele e o nosso sistema é bem simples. A principal parte está no fato que a OPA usa umas rotinas geradas pelo SWIG (para cada linguagem de *script*) para conseguir converter instâncias de tipos nativos definidos usuário entre C++ e as máquinas virtuais.

O SWIG usa arquivos separados de extensão "`.i`", chamados de "arquivos de interface do SWIG", para determinar quais arquivos de C++ ele irá processar, e como irá gerar o *wrapper* para tal código, como o seguinte:

```
1  %module NOME_DO_MODULO
2
3  %include "std_string.i"
4  %include "std_vector.i"
5
6  %{
7  #include <src/coisas/coisa.h>
8  %}
9
10 %include <src/coisas/coisa.h>
```

Nesses arquivos de interface do SWIG é também possível escrever código que será incluído explicitamente no código gerado, ou usado internamente em algumas etapas, como por exemplo nas conversões de valores que passam pelos *wrappers*. O segundo tipo de interação com o SWIG que fizemos está nesses arquivos. Usando o mecanismo de extensões do SWIG, desenvolvemos algumas funcionalidades com diversos objetivos nesses arquivos de interface, mas podemos classificá-las de dois jeitos:

1. Operações para fornecer à OPA informações sobre os tipos nativos definidos pelo usuário que serão exportados para as máquinas virtuais, através das estruturas de dados que o SWIG usa para representá-los. A OPA necessita desses dados para poder converter tais tipos entre a máquina virtual e o C++.

2. Operações para mudar ou adicionar alguma funcionalidade ao *wrapper* gerado. Elas essencialmente tentam resolver algum problema do SWIG, como a incapacidade de herdar tipos de C++ nas linguagens de *script*. Veremos mais sobre esses problemas na próxima seção.

Aqui está um exemplo mais detalhado de um arquivo de interface do SWIG, com algumas de nossas adições em uso:

```
1  %module NOME_DO_MODULO
2
3  // Inclui algumas definicoes nossas e do SWIG para simplificar este arquivo.
4  %include <module/export.swig>
5  %include <module/proxy.swig>
6  %include <module/widetypes.swig>
7  %include "std_string.i"
8  %include "std_vector.i"
9
10 // O codigo abaixo eh transcrito diretamente para o wrapper gerado.
11 // Normalmente aproveitamos ele para incluir os cabecalhos apropriados.
12 %{
13 #include <src/coisas/coisa.h>
14 #include <src/coisas/mais_coisa.h>
15 %}
16
17 // Se necessario, com essa(s) linha(s) voce pode fazer esse modulo referenciar
18 // outro modulo caso ele use alguma definicao nesse outro.
19 %import(module="outro_modulo") <src/outras_coisas/outro.h>
20
21 // Se necessario, com essa(s) linha(s) voce pode ignorar declaracoes em C++ para
22 // que elas nao sejam exportadas.
23 %ignore coisas::CoisaEstranha;
24
25 // Se necessario, com essa linha voce pode definir quais metodos e funcoes
26 // que devolvem novas instancias de objetos - isso eh necessario para o controle
27 // correto da memoria, pois indica que a maquina virtual pode tomar posse do objeto
28 %newobject coisas::CoisaBase::Foo;
29
30 // Nao eh possivel exportar templates de C++, mas voce pode exportar uma
31 // especializacao, se necessario:
32 %template(IntCoisa) coisas::CoisaVariavel<int>;
33
34 // Se necessario, com essa linha voce pode habilitar proxies para tal
35 // classe de C++, para que ela possa ser herdada pelos scripts. Note que
36 // soh isso nao eh suficiente para habilitar heranca.
37 proxy_class(coisas::CoisaBase)
38
39 // Finalmente isso informa ao SWIG o que exportar. No caso, diz para tudo no
40 // arquivo indicado ser exportado, exceto se algo foi ignorado usando %ignore.
41 %include <src/coisas/coisa.h>
42
43 // Esse par enable/disable_disown eh essencialmente o contrario do %newobject
44 // acima: eles definem que parametros de funcoes com o tipo/nome especificados
45 // aqui terao sua posse roubada pelo C++.
46 // Isso serah explicado em detalhes mais adiante.
47 enable_disown(coisas::CoisaBase* c)
48 %include <src/coisas/mais_coisa.h>
49 disable_disown(coisas::CoisaBase* c)
50
51 // Aqui definimos quais classes que estamos exportando
52 namespace coisas {
53     export_class(CoisaBase)
54 }
55
56 // E confirmamos as classes exportadas, para o OPA saber converte-las
57 // quando usar VirtualObj::value() e VirtualObj::set_value()
58 confirm_exports(NOME_DO_MODULO)
```

4.5 Problemas com SWIG

Ao longo do tempo em que fomos usando o SWIG notamos alguns problemas com ele, que eram refletidos no nosso sistema, impondo certas restrições. Alguns desses problemas conseguimos apaziguar com código inserido nos arquivos de interface do SWIG, como mencionado na seção anterior. Os problemas eram:

- **A determinação de posse de objetos depende da API do usuário e o design do SWIG não é prático para reconhecer essas especificações**.

  Ceder a posse (*disown*) é o ato de fazer que algum método da interface C++ retire a posse de uma instância de uma classe C++ que ele recebe como argumento vindo da máquina virtual. Normalmente, quando esta cria algum objeto C++, ou seja, um objeto na linguagem de *script* que encapsula uma instância de alguma classe C++ que foi exportada, ela retém a posse de tal objeto. Quando ela for remover tal objeto virtual, seja lá por qual razão, se ela detiver a posse dele, ela também irá remover da memória a instância C++ encapsulada. Isso pode causar problemas se alguma parte do código nativo tem alguma referência para tal instância, e tenta remover ou simplesmente usar ela em algum momento após a máquina virtual já tê-la removido. Portanto fizemos que métodos da interface C++ que recebem instâncias de classes e cuidam da remoção delas internamente tiram da máquina virtual a posse de tal instância.

  A questão da posse de um objeto entre C++ e uma máquina virtual é essencial ao problema que nosso sistema se propõe a tratar e ao nosso ver não tem como ser evitada. Por isso, essa complicação no SWIG é um grande obstáculo ao nosso trabalho.

- **O SWIG não exporta o aninhamento[17] de classes, *structs* e *unions*, e aparentemente não pretende resolver isso tão cedo**.

- **O SWIG não tem trata a herança de classes C++ nos *script*s**.

  A nossa "solução" para isso foi criar um sistema de *proxies*[18]: classes intermediárias em C++ que derivam de uma classe base `BaseProxy` e da classe C++ que foi exportada e da qual queremos herdar. Essas classes *proxy* contêm uma instância de `VirtualObj` encapsulando um objeto virtual que deve ser uma instância da classe em *script* que herda a classe nativa em questão. Elas também implementam os métodos abstratos da classe base, usando o objeto virtual para repassar as chamadas dos métodos para a classe virtual. Fora a definição da classe `BaseProxy`, e das classes *proxy* para cada classe C++ passível de ser herdada, para esse sistema funcionar foi também necessário uma grande parte de código adicionado nos arquivos de interface do SWIG.

- **O SWIG não exporta em seus *wrappers* variáveis e métodos de classes com modificador de acesso *protected* (acessíveis somente pela classe em si e por classes derivadas).** Isso decorre do fato que ele não tem suporte a herança, como mencionado anteriormente, mas ainda assim é um problema ao nosso ver.

- **Problemas com *wrappers* de Lua**:

  – Uma das informações necessárias (metatabelas das classes) só pode ser obtida fazendo suposições sobre a implementação interna dos *wrappers* de Lua do SWIG. Ou seja, é preciso usar más práticas de programação para conseguir o acesso a esses dados.

  – Duas funções internas dos *wrappers* de Lua do SWIG precisam ter uma linha de código alterada para que a solução de *proxies* acima funcione. Tivemos que manipular o funcionamento interno do SWIG para que ele passasse a usar versões nossas dessas funções ao invés das originais.

[17] Aninhar tais estruturas de C++ é o ato de definir algumas delas dentro de outra delas.

[18] Uma alternativa para simplificar esse processo está no nosso blog: http://projeto-ouroboros.blogspot.com.br/2013/07/gerador-de-proxies.html (último acesso: 15/09/2013)

4.6 Nosso próprio gerador de *wrappers*

Boa parte do que explicamos nas seções anteriores neste capítulo foi o trabalho que realizamos antes do trabalho de formatura. E o principal ponto do trabalho nesse nosso último ano é exatamente o de substituir o SWIG por uma ferramenta própria que criamos, a fim de simplificar o uso e corrigir os erros listados na seção anterior. Mantivemos a OPA, fazendo algumas alterações para resolver erros, melhorar o sistema ou substituir as partes relativas ao SWIG para a nossa nova ferramenta: o nosso gerador de *wrappers*, OPWIG, cujo papel foi descrito na seção 3.3.

Atualmente, ele ainda não é capaz de substituir completamente o SWIG, mas ele já realiza algumas tarefas básicas importantes, como exportar funções, variáveis e classes. E continuamos o seu desenvolvimento em um passo constante razoável. Infelizmente não podemos ainda falar que já conseguimos resolver os problemas do SWIG pois como nossa ferramenta ainda está no meio do desenvolvimento, ainda não chegamos no ponto de ter que lidar com eles. Mais especificamente, ainda não chegamos em *milestones* que exijam essas funcionalidades mais avançadas. No entanto, já conseguimos notar que pelo menos será relativamente fácil corrigir o problema de classes, `structs` e `unions` aninhadas. De fato, nosso gerador, ao analisar código C++, já reconhece essas declarações, só falta adaptarmos os *wrappers* para tratá-las e testar em uma *milestone* se de fato funciona.

Uma vantagem presente no OPWIG que não existia no SWIG é o registro automático de máquinas virtuais no gerenciador, simplificando o código do usuário para usar o sistema. Esse registro automático funciona da seguinte maneira: cada módulo gerado pelo OPWIG contém um bloco de código de *bootstrap* (inicialização), que é executado automaticamente quando o programa é carregado, dado que ele tenha sido compilado junto com o código de *wrapper* gerado. O *bootstrap* então registra a máquina virtual da linguagem de *script* para a qual ele está exportando, caso ela ainda não tenha sido registrada, e registra o módulo de *script* e seus sub-módulos nela. Colocamos um exemplo completo do que o gerador faz na seção 5.2.

Internamente, o OPWIG contém:

- Um analisador de código C++.
- Classes representando metadados de C++.
- Um gerador de código, que segue uma **especificação** para cada máquina virtual.
- O molde que uma especificação dessas deve seguir, recebendo os metadados com base nos quais será produxido o código que deverá ser escrito no arquivo gerado. Essa definição abstrata deve ser implementada para uma dada linguagem de *script* para que o OPWIG seja capaz de gerar código de *wrappers* para a máquina virtual dela. Novamente, nós criamos a implementação para `Lua` e `Python` como padrão.

4.6.1 O analisador de C++

Um dos pontos mais importantes desse projeto, e que ocupou boa parte do nosso primeiro semestre em 2013, foi o desenvolvimento desse analisador. Ele é uma ferramenta que processa uma sequência de símbolos em uma dada linguagem, de acordo com a gramática dela, e gera uma estrutura de dados que representa o código analisado, e ao mesmo tempo verifica se ele está sintaticamente correto.

No nosso caso, queremos analisar código C++ e gerar metadados que representem as declarações feitas nele, como váriaveis, funções e classes. Esse processo passa por três etapas:

- A **análise léxica**, que consiste em transformar os caracteres do texto lido em símbolos significativos de acordo com um conjunto de expressões regulares que eles devem satisfazer. Esses símbolos são chamados de *tokens* ou símbolos terminais, formando a base da análise seguinte. Para facilitar a implementação dessa etapa no OPWIG, usamos o gerador de analisadores léxicos **Flexc++** [Bro13b].

- A **análise sintática**, que seguindo a gramática da linguagem considerada, descobre se os símbolos terminais fornecidos pela análise léxica formam uma expressão válida, como uma declaração, uma lista de parâmetros, uma definição de classe, etc. Para implementar essa parte, usamos o gerador de analisadores sintáticos **Bisonc++** [Bro13a].

  O termo *parser* significa analisador sintático, mas é comum encontrar essas duas primeiras etapas sendo realizadas juntas por uma mesma ferramenta. Como a análise sintática é o passo mais importante, usualmente tal ferramenta é simplesmente chamada de *parser*.

- A **análise semântica**, que deve descobrir as implicações das expressões validadas pela análise sintática e tomar uma ação apropriada.

Tanto o Flexc++ quanto o Bisonc++ exigem um arquivo separado para descrever as especificações que eles devem seguir. No caso, nós fizemos o arquivo do primeiro seguindo as palavras reservadas e outro símbolos terminais da linguagem C++, e o do segundo seguindo a gramática dela. No arquivo do Bisonc++ podemos especificar ações (blocos de código C++) a serem executadas quando um tipo expressão da gramática é reconhecido. Nós programamos tais ações para criar metadados que representam as declarações do código C++ analisado, dando ínicio à análise semântica.

4.6.2 Metadados de C++

Esses metadados são representados por um conjunto de classes no OPWIG, indicando a estrutura e as declarações de código C++ encontrados na análise anterior. No momento, temos dois principais tipos de metadados:

- Metadados de objetos normais: váriaveis, funções e enumerações.
- Metadados de escopos: classes e *namespaces*, que além de seus atributos próprios também contêm outros metadados aninhados, inclusive outros escopos.

Mas esses metadados por si só não realizam nada, eles simplesmente contêm dados. O OPWIG constrói eles usando o analisador, e consome eles para gerar o código correto dos *wrappers*.

4.6.3 Especificação de *wrappers*

O gerador de código, junto com a especificação de *wrappers* para uma linguagem de *script*, transforma os metadados em um arquivo de código C++ contendo os *wrappers* que possibilitarão que o código analisado seja exportado. O que o gerador faz é razoavelmente simples: ele cria o arquivo de *wrappers* seguindo um conjunto de blocos ordenados que são gerados combinando os critérios internos do OPWIG com a especificação para linguagem de *script* envolvida:

1. **Inicial**: fornecido pela especificação usada, esse bloco deve conter definições comuns para os *wrappers* no resto do arquivo.
2. **Dependências**: fornecido pelo próprio gerador, esse bloco contém o código que inclui os arquivos de cabeçalho que foram analisados para criar cada módulo.
3. **Intermediário**: fornecido pela especificação usada, esse bloco é similar ao inicial, mas é posicionado após o código das dependências. Assim, se necessário, o desenvolvedor pode introduzir código tanto antes quanto depois das inclusões de cabeçalhos.
4. **Metadados**: construído pela especificação usada, esse bloco na verdade é composto por uma sequência de sub-blocos, relativos aos *wrappers* de cada metadado. Quando estes são normais, gera-se apenas um sub-bloco na sequência, enquanto que quando eles são de escopo, outra subsequência de blocos é formada: primeiro um bloco para marcar o começo do escopo, depois vários para os metadados contidos nele (com possivelmente mais escopos), e um último para marcar o final do escopo. No final, temos todos os *wrappers* necessários para exportar as definições que os metadados representam.

5. **Final**: construído pela especificação usada, esse bloco deve conter todo o código necessário para exportar os módulos para a máquina virtual em questão. Normalmente isso inclui tabelas das rotinas e das classes exportadas e a função de inicialização (aquelas vistas na seção 2.3) de cada módulo virtual envolvido.

6. ***Bootstrap***: fornecido pelo gerador de código, esse bloco é responsável por registrar a máquina virtual da linguagem d *script* no gerenciador da OPA, caso ela ainda não tenha sido, e exportar para ela os módulos virtuais formados a partir dos *wrapers*.

4.7 Funções auxiliares de CMake

Como dissemos na seção 3.3, a geração dos *wrappers* precisa ser feita antes da compilação da aplicação do usuário pelos geradores responsáveis por tratar cada máquina virtual usada. Isso gera uma complicação para o usuário e outra para os desenvolvedores. A do lado do usuário é que ele precisa lembrar de re-gerar os *wrappers* sempre que ocorrer alguma alteração relevante no código fonte de sua aplicação. E no lado dos desenvolvedores, sempre que for implementada a compatibilidade com uma nova máquina virtual de linguagem de *script*, eles precisam compilar o executável do gerador correspondente também, e o código fonte deste é sempre praticamente o mesmo independentemente da máquina virtual usada, pois tudo que ele faz é chamar algumas rotinas da nossa biblioteca principal. Para facilitar esses dois procedimentos, fizemos algumas funções em CMake que automatizam eles. A única restrição dessa nossa solução é que o usuário teria que usar CMake para gerenciar a compilação da aplicação dele também. Sinceramente, é uma ferramenta muito útil e aceita por desenvolvedores, além de compatível com diversos ambientes de desenvolvimento, então lhe seria vantajoso usá-la.

Enfim, para a geração de *wrappers* do usuário, fizemos a função `ouroboros_wrap_module()`. Basicamente, ele deve usá-la para cada módulo da aplicação que ele quiser exportar, especificando o nome do módulo, a máquina virtual para qual exportar, o diretório onde os arquivos gerados devem ficar e a lista de cabeçalhos que o gerador precisa analisar para formar o módulo. A função por sua vez fornece ao usuário o nome do arquivo que será gerado para que ele o inclua entre os arquivos que farão parte da compilação da sua aplicação. Um exemplo de uso é:

```
1    ouroboros_wrap_module (
2      my_module                      # nome do modulo
3      lua                            # linguagem para qual exportar
4      src/                           # diretorio de saida
5      MY_PROJECT_LUA_GENERATED_SRC   # variavel que receberah o resultado
6    )
```

Para automatizar o processo de fazer o executável do gerador de *wrappers* para as máquinas virtuais, fizemos uma função chamada `ouroboros_generate_opwig()`. Dada a linguagem de *script* da máquina virtual em questão e a classe de especificação de *wrappers* apropriada, essa função gera o código do gerador apropriado (sim, nós geramos o gerador). Com isso, o desenvolvedor precisa apenas fazer seu trabalho no código fonte da `libouroboros` e deixar que o CMake providencie o gerador. O código que ele precisa colocar nas suas listas de CMake é algo equivalente a:

```
1    ouroboros_generate_opwig (
2      lua                                 # nome da linguagem
3      languages/lua/wrapperspecification.h  # cabecalho com a especificacao
4      opa::lua::WrapperSpecification        # classe que implementa especificao
5    )
```

Capítulo 5

Resultados

Neste capítulo iremos apresentar os resultados obtidos com o desenvolvimento do nosso projeto até então. Parte desses resultados surgiram antes do trabalho de formatura, mas ainda assim foram aprimorados durante a evolução dele. Eles formam o que chamamos de sistema do Projeto Ouroboros, na forma de uma biblioteca `C++` de nome **`libouroboros`**. Como vimos no capítulo 3, ela é composta por duas grandes partes, uma responsável por incorporar *script*s e outra por exportar funcionalidades nativas para *script*s. São a OPA e o OPWIG, resepectivamente.

Para deixar clara a separação entre eles, os cabeçalhos da biblioteca dividem a interface dela em dois *namespaces*: **`opa`** e **`opwig`**. Normalmente, quando o usuário usar nosso sistema, ele limitar-se-á ao uso das classes e rotinas do primeiro. Mas o segundo ainda se faz necessário para que os geradores de *wrappers* funcionem apropriadamente.

Nossa intenção é que a **`libouroboros`** seja facilmente compilada em várias plataformas usando o CMake para gerar arquivos relativos à compilação, como *Makefile*s no Linux. Não chegamos a testar em Mac OS mas por enquando em Windows não é possível compilar usando Visual Studio porque ele não reconhece `C++11` completamente ainda. Em Linux, contanto que os pacotes necessários estejam instalados, a biblioteca compila e funciona sem problemas usando um `g++` de versão 4.7 ou mais recente.

As extensões que fizemos da OPA e do OPWIG que comportam as linguagens de *script* `Lua` e `Python` também estão funcionais (embora ainda não com todas os recursos que gostaríamos), sendo distríbuidas como extensões da **`libouroboros`** na forma de bibliotecas separadas: **`libouroboros-lua`** e **`libouroboros-python`**, para `Lua` e `Python`, respectivamente. Assim é fácil incluí-las em uma mesma aplicação: basta ligá-la tanto com a **`libouroboros`** quanto com elas. Caso o usuário não queira usar uma das linguagens, é só não ligar com a biblioteca correspondente. Caso ele queira alguma outra linguagem, é só ele obter a implementação da **`libouroboros`** específica dela e ligá-la junto. Essas extensões exigem, como é de se esperar, as bibliotecas de `Lua` (versão 5.1) e de `Python` (versão 2.7). **Os pacotes dessas bibliotecas precisam estar instaladas no sistema para que o CMake as encontre e as ligue devidamente com as extensões de nossa biblioteca**.

5.1 OPA

Dentre as duas partes, apenas a OPA está com praticamente todas as funcionalidades desejadas implementadas. Como descrito nos capítulos anteriores, ela inclui um sistema para gerenciar as máquinas virtuais e incorporar *script*s independente de sua linguagem, tudo com uma interface simples, genérica e robusta para que o usuário de `C++` possa trabalhar sem se preocupar com os pormenores dos serviços fornecidos. É provável que ainda alteremos algumas coisas no sistema, arrumando os eventuais erros que surgirem ou implementando novas funcionalidades.

5.1.1 Instruções de Uso

Aqui iremos explicar brevemente como usar a OPA da **`libouroboros`**, para demonstrar o que obtivemos de resultados com ela. Como essa é a única parte que o usuário irá usar diretamente da biblioteca, também explicamos aqui como compilá-la usando CMake e Makefile. Depois mostraremos

um simples exemplo de um programa que carrega um *script*, pega o valor de uma variável dele e do resultado da execução de uma função.

Compilação

Como ainda não disponibilizamos nenhum pacote com os binários da `libouroboros`, você terá que obter o código fonte dela e compilá-la manualmente. Para compilar, basta usar o CMake na pasta raiz do projeto que ele gerará os arquivos de compilação necessários. E então, basta executar os comandos de `make` para compilação:

```
$ cmake .
$ make libouroboros
$ make libouroboros-python
$ make libouroboros-lua
```

Os dois ultimos comandos são para compilar as implementações para cada linguagem, e você precisa usá-los somente se quiser de fato elas. Note que apesar de a `libouroboros` não ter nenhuma dependência externa, as implementações das extensões para cada linguagens dependem dos pacotes de desenvolvimento de suas respectivas bibliotecas, para ter acesso à implementação das APIs das máquinas virtuais. É também possivel simplesmente executar

```
$ cmake .
$ make
```

Isso irá compilar a `libouroboros`, `libouroboros-python`, `libouroboros-lua` e os geradores correspondentes (mais sobre como usar eles na próxima seção). Caso queira, as seguintes opções podem ser passadas para o CMake para alterar seu comportamento:

- `OUROBOROS_CREATE_BINDINGS`: quando ativada, o CMake irá tentar criar as implementações específicas de cada linguagens.
- `OUROBOROS_LUA_BINDINGS`: quando ativada, a implementação de `Lua` será habilitada.
- `OUROBOROS_PYTHON_BINDINGS`: quando ativada, a implementação de `Python` será habilitada.

O padrão é todas elas estarem ativadas. É possível também compilar os testes unitários da OPA, que criamos para testar suas funcionalidades, usando o seguinte comando:

```
$ make opa_test
```

E para executá-los:

```
$ ./test/opa_test
```

Uma vez compilada a `libouroboros`, lembre-se de incluir os cabeçalhos necessários no código de sua aplicação e de ligar as bibliotecas com ela, além de ligar também com a `libouroboros-lua` e/ou a `libouroboros-python`.

Exemplo de Código

Nesse exemplo, o programa começa inicializando o gerenciador, especificando a pasta na qual ele deverá procurar por *script*s (linha 9). Depois ele carrega um *script* (linha 12), pega o valor de uma variável de dentro dele (linha 15), muda o valor dessa variável (linha 18), e executa uma função que fornece um valor de resultado (linhas 21-22). Por fim, ele finaliza o gerenciador (linhas 28-29):

```
#include <opa/scriptmanager.h>
#include <opa/virtualobj.h>

using opa::VirtualObj;
using opa::ScriptManager;

int main() {
    //inicializando
    SCRIPT_MANAGER()->Initialize("./scripts/");

    //carregando script
    VirtualObj modulo = SCRIPT_MANAGER()->LoadModule("exemplo");

    //pegando variavel
    double valor_antigo = modulo["number"].value<double>();

    //mudando o valor da variavel
    modulo["number"].set_value<double>(42.0);

    //executando funcao
    VirtualObj funcao = modulo["DoStuff"];
    VirtualObj resultado = funcao();

    //finalizando
    SCRIPT_MANAGER()->Finalize();
    delete SCRIPT_MANAGER();

    return 0;
}
```

Esse exemplo omite a inicialização das máquinas virtuais. Elas precisam ser registradas antes de o gerenciador ser inicializado. Os *wrappers* gerados pelo OPWIG fazem isso automaticamente, mas como nesse pequeno exemplo não usamos nenhum, temos que fazer isso manualmente. Basta fazer essas modificações no código acima:

```
... [includes anteriores] ...
#include <opa/config.h>
#ifdef OUROBOROS_LUA_BINDINGS
#include <languages/lua/luamachine.h>
#endif
#ifdef OUROBOROS_PYTHON_BINDINGS
#include <languages/python/pythonmachine.h>
#endif

using ...

int main () {
#ifdef OUROBOROS_LUA_BINDINGS
    if (SCRIPT_MANAGER()->GetMachine("Lua") == nullptr)
        SCRIPT_MANAGER()->Register(new opa::lua::LuaMachine());
#endif
#ifdef OUROBOROS_PYTHON_BINDINGS
    if (SCRIPT_MANAGER()->GetMachine("Python") == nullptr)
        SCRIPT_MANAGER()->Register(new opa::python::PythonMachine());
#endif

    //inicializando
    SCRIPT_MANAGER()->Initialize("./scripts/");

    ...
}
```

Os `ifdef`s usarão as configurações que estavam ativadas no CMake para determinar quais máquinas virtuais devem ser registradas, e quais cabeçalhos precisam ser inclusos para tanto. Finalmente, para esse exemplo funcionar é necessário um *script* como o seguinte, em **Python**:

```python
#exemplo.py

number = 1138.0

def DoStuff():
    return "coisas foram realizadas"
```

ou esse, em Lua:

```lua
--exemplo.lua

number = 1138.0

function DoStuff()
    return "coisas foram realizadas"
end
```

E é ai que você pode ver a robustez da OPA e de sua generalização das linguagens. Nesse exemplo você pode usar tanto o *script* em Lua como o *script* em Python e o código C++ não será alterado e irá funcionar do mesmo jeito.

5.2 OPWIG

Essa é a parte com o propósito de substituir o SWIG. Ela ainda não implementa todas funcionalidades que ele tem, e portanto ainda não consegue substituí-lo, porém já é funcional. As estruturas C++ que ela é atualmente capaz de exportar nos *wrappers* gerados são:

- ***Namespaces***, que são traduzidos em módulos e sub-módulos na máquina virtual, de acordo com a hierarquia que eles apresentam.
- **Variáveis globais**, possivelmente constantes.
- **Funções globais**
- **Classes simples**, com:
  - Destrutor.
  - Construtor trivial (não recebe argumentos).
  - Atributos, possivelmente constantes.
  - Métodos não estáticos.

Os tipos de atributos, váriaveis, paramêtros e valores devolvidos que são reconhecidos são tanto os primitivos (naturais da linguagem nativa, como `int` ou `double`) quanto os complexos (classes do usuário que foram exportadas junto). Além disso o código gerado pelo OPWIG também tem, como dissemos no capítulo anterior, um bloco de inicialização (*bootstrap*) que registra a máquina virtual correspondente no gerenciador, caso ela não tenha sido ainda, e registra nela os módulos e sub-módulos que esse arquivo gerado visa exportar.

É importante notar também que o código gerado pelo OPWIG depende de algumas funcionalidade presentes na OPA, portanto qualquer programa compilado junto com o código gerado por ele deve ser ligado com a `libouroboros`. Esse comportamento foi uma decisão de projeto que fizemos para simplificar o código gerado pelo OPWIG e garantir uma melhor integração com a parte de incorporação. Supomos que se um usuário está usando o OPWIG então ele provavelmente também usará a OPA, já que o maior diferencial do nosso sistema com relação ao SWIG é este apenas fornece a geração de *wrappers* para exportação enquanto que o Projeto Ouroboros provê essas duas vias de comunicação entre linguagem nativa e linguagens de *script*.

5.2.1 Instruções de Uso

Aqui iremos explicar como usar o OPWIG, desde a compilação dos geradores até a execução deles e a inclusão do código gerado em sua aplicação, novamente com a intenção de demonstrar as capacidades dele.

É importante lembrar que o OPWIG na verdade é uma parte da nossa biblioteca. Por simplicidade de código e organização do sistema, todo o código do OPWIG é compilado junto com a OPA na `libouroboros`. Cada extensão individual para linguagens de *script* (no caso, a `libouroboros-lua` e a `libouroboros-python`) deve fornecer a sua especificação de *wrappers* para que o OPWIG saiba como gerá-los. Depois, para de fato usá-lo é necessário um outro arquivo de código, compilado como um executável separado, que use a especificação desejada quando evocar as rotinas do OPWIG. Como mencionamos na seção 4.7, esses arquivos de código são gerados automaticamente pelo CMake, pois eles seguem o seguinte padrão:

```
1  #include <opwig/opwig.h>
2
3  // Inclui o cabecalho para a especificacao de wrappers da maquina virtual em questao
4  #include <...header da especificacao da linguagem...>
5
6  int main (int argc, char** argv) {
7      // O parametro do template LanguageSpecification nesta chamada de funcao
8      // devera ser a classe que implementa a especificacao wrappers.
9      return opwig::gen::Execute< LanguagenSpecification >(argc, argv);
10 }
```

Tais executáveis são os geradores de fato, e cada um deverá ser ligado junto com a `libouroboros` e as extensões dela que a linguagem de *script* em questão exige. Esses geradores são gerados com um nome de prefixo "`opwig-`" seguido pelo nome da linguagem de *script* para a qual eles exportam. A interface do OPWIG disponibilizada pela `libouroboros` e os módulos de CMake que criamos fazem ser trivial a terefa de criar executáveis do OPWIG para cada linguagem, e as listas de CMake do repositório do projeto já usam essas facilidades para construir o `opwig-lua` e o `opwig-python`.

Compilação

Como ainda não disponibilizamos um pacote com as ferramentas prontas, será necessário compilar os geradores. Para tal, basta usar o CMake como acabamos de explicar:

```
$ cmake .
$ make opwig-lua
$ make opwig-python
```

O `opwig-lua` depende da `libouroboros-lua`, e analogamente o `opwig-python` depende da `libouroboros-python`, então eles precisam estar disponíveis no sistema. Vale lembrar que também é possível compilar os geradores juntos com a `libouroboros` e suas extensões padrões usando:

```
$ cmake .
$ make
```

É possível também compilar o extenso conjunto de testes unitários do OPWIG, que criamos para testar suas funcionalidades, usando o seguinte comando:

```
$ make opwig_test
```

E para executá-los:

```
$ ./test/opwig_test
```

Execução

Após compilado, executar um dos geradores é simples:

```
$ opwig-* [--module-name=NOME] ARQUIVO-1 ARQUIVO-2 ... ARQUIVO-N
```

Onde:

- **opwig-*** é o OPWIG em questão. Você precisa especificar a linguagem para a qual você quer exportar (por exemplo, `opwig-python` para o gerador de *wrappers* para `Python`).
- **NOME**: é o nome do módulo que será gerado.
- **ARQUIVO-i**: são os arquivos de cabeçalho em `C++` que contêm a interface que você deseja que seja exportada no módulo.

Cada gerador é especificado para uma só linguagem e portanto ao ser executado ele obrigatoriamente irá gerar um arquivo de código com o módulo exportado para essa linguagem apenas. O arquivo gerado é criado na mesma pasta onde gerador foi executado, seguindo o seguinte padrão de nomenclatura:

```
<nome da linguagem de script>_<nome do módulo>_wrap.cxx
```

Por exemplo, se meu módulo exportado se chamasse "`mymodule`". o nome do *wrapper* gerado para `Lua` seria "`Lua_mymodule_wrap.cxx`". Alternativamente, você pode optar por usar CMake para compilar sua aplicação, o que lhe permite usar as funcionalidades descritas na seção 4.7 para automatizar todo esse procedimento.

Vamos mostrar agora o que nosso gerador produz em um caso bem simples. Não será demonstrado todas as capacidades dele, pois não só teria que ser um caso muito artifical, como também o código gerado seria muito extenso (atualmente, a nossa mais recente *milestone* gera um total de 1203 linhas de código, somando os *wrappers* de `Lua` e de `Python`). O cabeçalho que será analisado será o seguinte:

```
1 // Arquivo "test.h"
2
3 // variavel global
4 int x;
5
6 // funcao global
7 double foo (double a, double b);
```

Como acabamos de ver, podemos gerar os *wrappers* com os comandos:

```
$ opwig-lua --module-name=test test.h
$ opwig-python --module-name=test test.h
```

E serão gerados arquivos com nome "Lua_test_wrap.cxx" e "Python_test_wrap.cxx", cujo conteúdo seguirá a sequência de blocos descrita na seção 4.6.3. Os código gerados ficam, respectivamente:

```
1
2 // This is a generated file.
3
4
5 #include <test.h>
6
7 #include <languages/lua/aux/exportmodule.h>
8 #include <languages/lua/luamachine.h>
9 #include <languages/lua/converter.h>
10 #include <languages/lua/header.h>
```

```cpp
#include <opa/scriptmanager.h>
#include <opa/module.h>
#include <opa/converter.h>
#include <iostream>
#include <string>
#include <list>
#include <stdexcept>

using std::list;
using std::string;
using std::cout;
using std::endl;
using std::runtime_error;
using opa::Module;
using opa::lua::LuaMachine;
using opa::lua::State;
using opa::lua::Constant;
using opa::lua::aux::ModuleInfo;
using opa::lua::aux::ExportModule;

// Begin wrappers

namespace generated {


int wrap_function_foo (lua_State* L) {
    int args = 0;
    if ((args = lua_gettop(L)) < 2)
        return luaL_error(
            L,
            "Error: %s expected %d arguments but received only %d.\n",
            "foo",
            2,
            args
        );
    opa::lua::Converter convert(L);
    double arg_0;
    double arg_1;
    try {
        arg_0 = convert.ScriptToType<double>(1);
        arg_1 = convert.ScriptToType<double>(2);
    } catch (runtime_error e) {
        return luaL_error(
            L,
            "Error: could not convert %s's arguments (%s).\n",
            "foo",
            e.what()
        );
    }
    double result = foo(arg_0, arg_1);
    lua_settop(L, 0);
    convert.TypeToScript<double>(result);
    int stack = 1;
    return stack;
}


int wrap_getter_x (lua_State* L) {
    opa::lua::Converter convert(L);
    convert.TypeToScript<int>(x);
    return 1;
}

int wrap_setter_x (lua_State* L) {
    opa::lua::Converter convert(L);
    int value;
    try {
        value = convert.ScriptToType<int>(1);
    } catch (runtime_error e) {
        return luaL_error(
            L,
            "Error: could not convert value to %s's type (%s).\n",
            "x",
            e.what()
        );
```

```
86      }
87      x = value;
88      lua_settop(L, 0);
89      return 0;
90  }
91
92
93  } // namespace generated
94
95  namespace {
96
97  /// Forward declaration.
98  int init (lua_State* L_);
99
100 luaL_Reg functions[] = {
101     { "foo", generated::wrap_function_foo },
102     { nullptr, nullptr }
103 };
104
105 luaL_Reg getters[] = {
106     { "x", generated::wrap_getter_x },
107     { nullptr, nullptr }
108 };
109
110 luaL_Reg setters[] = {
111     { "x", generated::wrap_setter_x },
112     { nullptr, nullptr }
113 };
114
115 luaL_Reg member_getters[] = {
116     { nullptr, nullptr }
117 };
118
119 luaL_Reg member_setters[] = {
120     { nullptr, nullptr }
121 };
122
123 luaL_Reg member_functions[] = {
124     { nullptr, nullptr }
125 };
126
127 ModuleInfo info(
128     "test", init,
129     {
130         {"getters",getters}, {"setters",setters}, {"functions",functions},
131         {"member_getters",member_getters},
132         {"member_setters",member_setters},
133         {"member_functions",member_functions}
134     },
135     { }
136 );
137
138 /// [-(1|2),+1,e]
139 int init (lua_State* L) {
140     return ExportModule(L, &info);
141 }
142
143 } // unnamed namespace
144
145 extern "C" {
146
147 /// [-(1|2),+1,e]
148 int luaopen_test (lua_State* L) {
149     return init(L);
150 }
151
152 } // extern "C"
153
154
155 namespace {
156
157 class Bootstrap final {
158   public:
159     Bootstrap ();
160 };
```

```
161
162 Bootstrap entry_point;
163
164 Bootstrap::Bootstrap () {
165     cout << "Bootstrapping Lua module test" << endl;
166     LuaMachine *vm = dynamic_cast<LuaMachine*>(
167         SCRIPT_MANAGER()->GetMachine("Lua")
168     );
169     if (vm == NULL) {
170         vm = new LuaMachine;
171         SCRIPT_MANAGER()->Register(vm);
172     }
173     vm->RegisterModule(Module<int(*)(lua_State*)>("test", luaopen_test));
174 }
175
176 } // unnamed namespace
```

```
 1 // FILE GENERATED BY OPWIG - OUROBOROS PROJECT.
 2
 3 #include <Python.h>
 4 #include <languages/python/pythonmachine.h>
 5 #include <languages/python/pythonconverter.h>
 6 #include <languages/python/wrapperbase.h>
 7 #include <opa/scriptmanager.h>
 8 #include <opa/module.h>
 9 #include <iostream>
10 #include <string>
11
12
13 #include <test.h>
14
15 using std::string;
16 using std::cout;
17 using std::endl;
18 using opa::Module;
19 using opa::python::PythonMachine;
20 using opa::python::PythonConverter;
21 using opa::python::wrapper::NumArgsOk;
22 using opa::python::wrapper::AddToParentModule;
23 using opa::python::wrapper::FuncErrorHandling;
24 using opa::python::wrapper::AddTypeToModule;
25
26 namespace opwig_py_generated {
27
28
29 PyObject* OPWIG_wrap_foo(PyObject* py_self, PyObject* args)
30 try {
31     if (!NumArgsOk(args, 2)) return nullptr;
32     PythonConverter converter (true);
33     double fArg0 = converter.PyArgToType<double>(args, 0);
34     double fArg1 = converter.PyArgToType<double>(args, 1);
35     double fValue = ::foo(fArg0, fArg1);
36     return converter.TypeToScript<double>(fValue);
37 }
38 catch (std::exception& e) { return FuncErrorHandling(e); }
39
40 PyObject* OPWIG_wrap_x(PyObject* self, PyObject* args)
41 try {
42     PythonConverter converter (true);
43     int oldValue = ::x;
44     if (static_cast<int>(PyTuple_Size(args)) == 1) {
45         ::x = converter.PyArgToType<int>(args, 0);
46     }
47     else if (!NumArgsOk(args, 0)) return nullptr;
48     return converter.TypeToScript<int>(oldValue);
49 }
50 catch (std::exception& e) { return FuncErrorHandling(e); }
51
52
53 } //namespace opwig_py_generated
54
55 //module test method table
56 static PyMethodDef testMethods[] = {
```

```
57      {"foo", opwig_py_generated::OPWIG_wrap_foo, METH_VARARGS, "calls C++ wrapped function" },
58      {"x", opwig_py_generated::OPWIG_wrap_x, METH_VARARGS, "wraps C++ variable - call() for
            get, call(newValue) for set [if possible]" },
59      {NULL, NULL, 0, NULL} //sentinel
60  };
61
62  PyMODINIT_FUNC
63  inittest(void) {
64      Py_InitModule("test", testMethods);
65  }
66
67  namespace {
68
69  class Bootstrap final {
70    public:
71      Bootstrap ();
72  };
73
74  Bootstrap entry_point;
75
76  Bootstrap::Bootstrap () {
77      cout << "Bootstrapping Python module test" << endl;
78      PythonMachine *vm = dynamic_cast<PythonMachine*>(
79          SCRIPT_MANAGER()->GetMachine("Python")
80      );
81      if (vm == NULL) {
82          vm = new PythonMachine;
83          SCRIPT_MANAGER()->Register(vm);
84      }
85      vm->RegisterModule(Module<void (*)(void)>("test", inittest));
86  }
87
88  } // unnamed namespace
```

Um dos nossos objetivos para um futuro próximo é conseguir gerar *wrappers* com menos código mas que ainda assim sejam razoavelmente legíveis.

Usando os módulos gerados

Para usar os módulos gerados, basta compilar eles junto com seu programa, ligado com a `libouroboros` e com as extensões necessárias. Qualquer *script* em uma linguagem compatível que for incorporado ao seu programa (usando a OPA) será capaz de incluir o módulo gerado usando o mecanismo padrão da linguagem para inclusão de módulos externos. E o módulo em si deverá ser usado como qualquer outro módulo normal dessa linguagem.

Vamos mostrar um exemplo mais elaborado que o anterior, para mostrar o quão fácil fica usar as definições exportadas em *script*, mas dessa vez omitiremos o código que seria gerado. Suponha a seguinte interface em C++:

```
1  // coisas.h
2
3  const char* prefixo = "Supimpa";
4
5  namespace funcoes {
6      void FazAlgumaCoisa(double num);
7  }
8
9  namespace objetos {
10
11     class Coisa {
12       public:
13         Coisa();
14         ~Coisa();
15
16         Coisa* PegaCoisa( const char* nome );
17         void ColocaCoisa(Coisa* coisa);
18
19         double fator;
20     };
```

```
21
22 }
```

Sendo exportada como um módulo de nome "`coisas`", essa interface pode ser usada em `Python` da seguinte forma:

```
1 import coisas
2
3 c = coisas.objetos.Coisa()
4 c2 = c.PegaCoisa( coisas.prefixo + "Batuta" )
5 c.ColocaCoisa( coisas.objetos.Coisa() )
6
7 coisas.funcoes.FazAlgumaCoisa( 42 * c.fator )
```

Ou em `Lua` assim:

```
1 require "coisas"
2
3 c = coisas.objetos.Coisa()
4 c2 = c:PegaCoisa( coisas.prefixo .. "Batuta" )
5 c:ColocaCoisa( coisas.objetos.Coisa() )
6
7 coisas.funcoes.FazAlgumaCoisa( 42 * c.fator )
```

5.3 *Milestones*

Como mencionamos na seção 4.2, adotamos uma metodologia de *milestones* a partir do segundo semestre desse ano. Inicialmente as *milestones* estavam localizadas no mesmo repositório do Ouroboros, mas depois optamos por separá-las em outro repositório para ficar mais similar à interação real entre um usuário e o nosso sistema[1].

Até o momento fizemos seis *milestones*, todas funcionais, e usamos o CMake para simplificar sua compilação. Todas seguem o padrão de um simples programa de linha de comando, onde o usuário digita comandos em texto e o computador devolve as respostas em texto também. Os mecanismos internos de tal programa são determinados pelo código `C++`, que é exportado via OPWIG para *script*s em `Lua` e `Python` que especificam os comandos e respostas da aplicação. Por sua vez, o programa usa a OPA para carregar esses *script*s e executar as funções adequadas.

Segue um breve relato do que projetamos em cada *milestone* e o que elas exigiam que o nosso sistema fornecesse ao usuário.

***Milestone* 0:** A primeira *milestone* que criamos (seguindo indexação por zero, como todo bom programador) é bem simples. Ela simplesmente necessitava que o OPWIG:

- Exportasse funções globais recebendo e devolvendo tipos primitivos.
- Exportasse um módulo a partir de um único arquivo de cabeçalho.

***Milestone* 1:** Já na segunda *milestone*, adicionamos:

- *Namespaces* como sub-módulos do módulo principal.
- Funções devolvendo `void` — ou seja, que não devolvem nenhum valor.
- Exportação de um módulo a partir de um ou mais arquivos de cabeçalhos.

***Milestone* 2:** Na terceira *milestone* nós testamos se os *script*s eram capazes de acessar os sub-módulos exportados diretamente, além de adicionar o seguinte:

[1] Repositório das *milestones*: https://github.com/Kazuo256/ouroboros-milestones (último acesso: 1/12/2013)

- *Namespaces* aninhados (*namespaces* dentro de *namespaces*) como uma árvore apropriada de sub-módulos.
- Váriaveis globais de tipos primitivos, possivelmente constantes (isso é, com o modificador `const` do `C`).

***Milestone* 3:** Nessa *milestone* nós adicionamos a exportação de classes simples, como mencionado em 5.2, com a diferença que os tipos envolvidos só podiam ser primitivos.

***Milestone* 4:** Essa *milestone* foi a única que saiu do padrão. Em vez de criar um programa para testar funcionalidades que iríamos adicionar ao OPWIG, resolvemos arrumar algumas outras partes do sistema:

- Movemos o código das especificações de *wrappers*, que até então faziam parte um executável separado da biblioteca, para dentro da `libouroboros`.
- Criamos os módulos de CMake para facilitar o uso do Ouroboros.
- Alteramos diversas funções do OPA para jogar exceções de `C++` quando algum erro ocorresse, ao invés de ignorá-los ou simplesmente imprimir alguma mensagem no console.

Consequentemente essa *milestone* não tem nenhum programa associado para testar o código.

***Milestone* 5:** Finalmente na sexta *milestone* resolvemos implementar algo importante que estava faltando: a conversão de valores que fossem instâncias de classes `C++`, para possibilitar que atributos, variáveis, paramêtros de funções e seus resultados devolvidos pudessem ter tipos complexos (isso é, definidos pelo usuário) ao invés de apenas tipos primitivos.

5.3.1 Executando a *Milestone*

Agora vamos explicar melhor como compilar e executar uma *milestone*. Como cada uma delas é um incremento da anterior, basta exemplificarmos a última delas. Para compilar a partir da pasta raiz do repositório das *milestones* basta usar:

```
$ cmake .
$ make milestone-05
```

E para executar:

```
$ cd milestone-05
$ ./bin/milestone-05
```

Para ilustrar melhor as capacidades do nosso sistema, incluímos as partes relevantes do código fonte dessa última *milestone* a seguir. No final da seção, a figura 5.1 mostra um exemplo da execução dela. Começamos pelos cabeçalhos cujas definições são exportadas para as máquinas virtuais que processarão os *script*s:

```
1  // info.h
2
3  #ifndef OUROBOROS_MILESTONE_PROMPT_INFO_H_
4  #define OUROBOROS_MILESTONE_PROMPT_INFO_H_
5
6  class Info {
7    public:
8      Info ();
9
10     const char* subject ();
11     const char* predicate ();
12     const char* object ();
13
14     void set_subject (const char* the_subject);
15     void set_predicate (const char* the_predicate);
```

```
16     void set_object (const char* the_object);
17
18   private:
19     char subject_[256], predicate_[256], object_[256];
20 };
21 #endif // OUROBOROS_MILESTONE_PROMPT_INFO_H_
```

```
 1 // input.h
 2
 3 #ifndef OUROBOROS_MILESTONE_PROMPT_IN_H_
 4 #define OUROBOROS_MILESTONE_PROMPT_IN_H_
 5
 6 #include "info.h"
 7
 8 namespace input {
 9
10 class Receiver {
11   public:
12
13     /// Receives a message from the input.
14     const char* ReceiveMessage ();
15
16     /// Receives a number from the input.
17     double ReceiveNumber ();
18
19     /// Receives a confirmation from the input.
20     bool ReceiveConfirmation ();
21
22     /// Receives an information from the input.
23     Info* ReceiveInfo ();
24
25   private:
26     char buffer[256];
27 };
28 } // namespace in
29 #endif
```

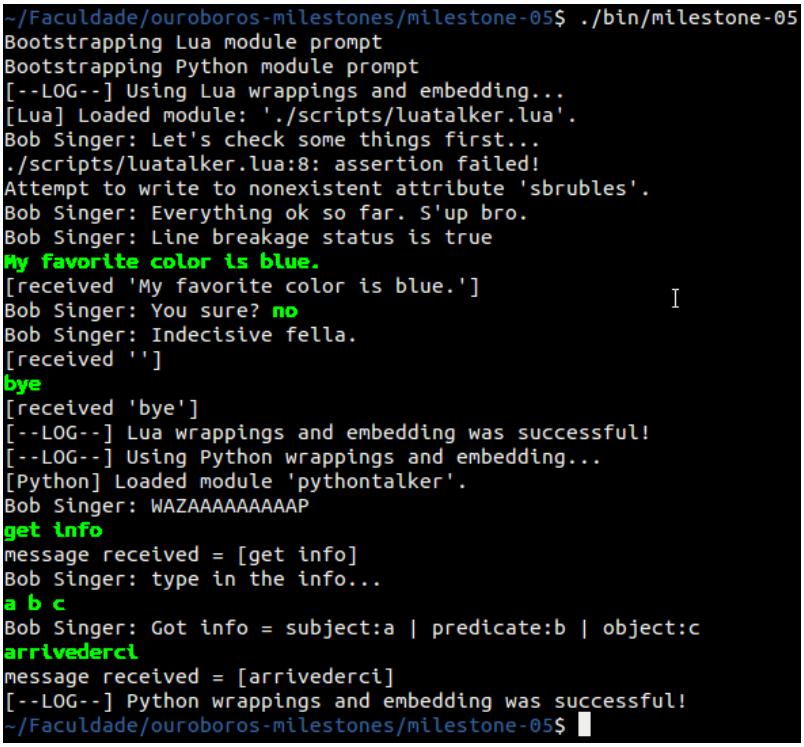
```
 1 // out.h
 2
 3 #ifndef OUROBOROS_MILESTONE_00_PROMPT_H_
 4 #define OUROBOROS_MILESTONE_00_PROMPT_H_
 5
 6 #include "info.h"
 7
 8 namespace out {
 9
10 class Sender {
11   public:
12
13     /// Avtivates line breaking.
14     bool break_line;
15
16     /// Constructor.
17     Sender();
18
19     /// Prints prompt output.
20     void SendMessage (const char* str);
21
22     /// Prints n prompt outputs.
23     void SendMultimessage (const char* str, int n);
24
25     /// Prints an information.
26     void SendInfo (Info* the_info);
27
28   private:
29     const char *const TALKER_NAME;
30 };
31 } // namespace out
32 #endif
```

Cada *milestone* processa dois *script*s, um em **Lua** e outro em **Python**, um em seguida do outro. Cada um deles estabelece um fluxo de comandos e respostas próprio. No caso do em **Lua**, ele também faz algumas verificações adicionais, para saber se os *wrappers* gerados detectam erros corretamente. Por isso, parece que ele solta mensagens de erro, mas elas são esperadas.

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
# pythontalker.py

from prompt.input import Receiver
import prompt

def main():
    rec = Receiver()
    out = prompt.out.Sender()

    out.SendMessage("WAZAAAAAAAAAP")
    while True:
        msg = rec.ReceiveMessage()
        print "message received = [%s]" % (msg)
        if msg in ["adios", "adieu", "arrivederci", "quit", "goodbye", "flw", "falou", "te
            mais", "te", "bye", "hasta"]:
            break

        elif msg == "hip hip":
            out.SendMultimessage("HOORAH!", 3)

        elif msg in ["vegeta", "scouter", "nappa"]:
            out.SendMessage("what was Goku power level reported from Vegeta scouter?")
            pl = rec.ReceiveNumber()
            if pl > 9000.0:
                out.SendMessage("correct.")
            else:
                out.SendMessage("dafuq dude?")

        elif msg == "ask me":
            out.SendMessage("vc eh bobo?")
            bobo = rec.ReceiveConfirmation()
            out.SendMessage(str(bobo)+"? Hmpf.")

        elif msg == "break":
            out.break_line = not out.break_line
            out.SendMessage("toggled break line - "+str(out.break_line))

        elif msg == "info around":
            out.SendMessage("type in the info...")
            inf = rec.ReceiveInfo()
            out.SendMessage("receiving and sending info...")
            out.SendInfo(inf)
        elif msg == "get info":
            out.SendMessage("type in the info...")
            inf = rec.ReceiveInfo()
            out.SendMessage( "Got info = subject:%s | predicate:%s | object:%s "%(inf.subject
                (), inf.predicate(), inf.object()) )
        elif msg == "send info":
            out.SendMessage("Type the info attributes in a row: subject, predicate and object
                .")
            s = rec.ReceiveMessage()
            p = rec.ReceiveMessage()
            o = rec.ReceiveMessage()
            inf = prompt.Info()
            inf.set_subject(s)
            inf.set_predicate(p)
            inf.set_object(o)
            out.SendMessage("Sending info...")
            out.SendInfo(inf)

        elif msg in ["help", "h"]:
            out.SendMessage(" hip hip, scouter, ask me, break, info around, get info, send
                info, quit")

    return True
```

```
-- luatalker.lua

local out     = require "prompt.out"
local input   = require "prompt.input"
local prompt  = require "prompt"

local function nogetter ()
  local test = input.Receiver()
  local check, err = pcall(function () assert(test.sbrubles) end)
  if check == false and err then
    print(err)
  else
    return false
  end
  return true
end

local function nosetter ()
  local test = out.Sender()
  local check, err = pcall(function () test.sbrubles = 42 end)
  if check == false and err then
    print(err)
  else
    return false
  end
  return true
end

function main ()
  local send = out.Sender()
  local receive = input.Receiver()
  local info = prompt.Info()
  info:set_subject "It"
  info:set_predicate "does not"
  info:set_object "matter"
  send:SendMessage("Lets check some things first...")
  if not nogetter() or not nosetter() then
    return false
  end
  send:SendMessage("Everything ok so far. Sup bro.", "unused");
  send:SendMessage("Line breakage status is "..tostring(send.break_line), "unused");
  while true do
    local msg = receive:ReceiveMessage()
    print("[received \"..msg..\"]")
    if msg == "bye" then return true end
    if msg == "dude" then
      send:SendMultimessage("Say what?", 5)
    elseif msg == "sqrt" then
      send:SendMessage("Of...?")
      local arg = receive:ReceiveNumber()
      send:SendMessage("Tis "..math.sqrt(arg))
    elseif msg == "My favorite color is blue." then
      send.break_line = false
      send:SendMessage("You sure? ")
      send.break_line = true
      local answer = receive:ReceiveConfirmation()
      if answer then
        send:SendMessage("Lame.")
      else
        send:SendMessage("Indecisive fella.")
      end
    elseif msg == "Learn info" then
      send:SendMessage "What info?"
      info = receive:ReceiveInfo()
      send:SendMessage "Ok!"
    elseif msg == "Tell info" then
      send:SendMessage "Here is what I know:"
      send:SendInfo(info)
    end
  end
end
```

Figura 5.1



O texto em verde são os comandos enviados pelo usuário.

Parte Subjetiva

Capítulo 6

Fernando Omar Aluani

6.1 Desafios e Frustrações

Acho que o maior desafio foi mesmo simplesmente a quantidade absurda de trabalho a ser feito. Mesmo considerando o que já tinhamos feito anteriormente, tivemos muito mais trabalho ainda a ser feito para substituir o SWIG e implementar as funcionalidades que queríamos. Desafios relacionados a problemas encontrados, melhor jeito para implementar algo ou como definir uma boa interface de alguma parte apareciam de tempos em tempos mas então nós nos reuníamos um dia e facilmente resolvíamos ele juntos.

Frustações... Primeiramente a de que demoramos um pouco no primeiro semestre do trabalho de formatura para colocar as coisas rodando, e começar a implementar o OPWIG. Inicialmente perdemos muito tempo para desenvolver o *parser* de forma que ele conseguisse reconhecer coisas comuns de serem encontradas em código `C++`. E também demoramos para desenvolver a estrutura básica de metadados, como gerá-los e armazená-los.

Outro ponto meio frustante era quando um integrante da dupla atrasava em suas tarefas, e aí o outro não tinha muito o que fazer (isso aconteceu de ambos lados). Isso ocorreu diversas vezes por falta de tempo, normalmente decorrida por conta de outras matérias sendo cursadas concorrentemente.

6.2 Relação entre o trabalho de formatura e disciplinas do BCC

Aqui está uma relação de matérias do BCC que mais afetaram o desenvolvimento deste trabalho de formatura, em minha opinião:

MAC0122 - `Princípios de Desenvolvimento de Algoritmos`

Como o próprio nome da disciplina já diz, essa matéria foi muito útil para aprender a desenvolver algoritmos, usar estruturas básicas de computação e a aprender a mexer em `C`. Com isso e um conhecimento prévio de `C#` e `Java`, foi fácil aprender a mexer com `C++`, a linguagem principal usada no desenvolvimento desse trabalho.

MAC0211 - `Laboratório de Programação I`

Aqui começamos a aprender a como organizar e modularizar melhor nosso código, simplificando o desenvolvimento. E também vimos e aprendemos a mexer com as ferramentas *bison* e *flex*, que foram indispensáveis para o desenvolvimento do *parser* `C++` usado pelo OPWIG.

MAC0323 - `Estrutura de Dados`

O conhecimento de estruturas de dados complexas, como criá-las eficientemente, como usá-las e até como generalizar partes dessas estruturas foi muito importante no desenvolvimento de diversas partes do trabalho.

MAC0414 - `Linguagens Formais e Autômatos`

Os tópicos de gramáticas e autômatos aprendidos nesta matéria foram bem úteis durante o

desenvolvimento do *parser* de C++, facilitando o nosso entendimento de como o *parser* funcionava e como alterá-lo para chegar no nosso objetivo.

USPGD001 - USPGameDev

Enquanto não é uma matéria (esta sigla foi tirada de um antigo grupo ainda existente no PACA), e sim um grupo de desenvolvimento e pesquisa, o USPGameDev ajudou bastante a:

- Aprender diversas coisas relacionadas a várias matérias, em alguns casos melhor do que a disciplina correspondente.
- Aprender e adquirir experiência com as linguagens envolvidas neste trabalho, principalmente C++.
- Trabalhar melhor em grupo.

Capítulo 7

Wilson Kazuo Mizutani

7.1 Desafios e Frustrações

Pessoalmente, meu primeiro desafio foi escolher um trabalho de formatura. Eu tinha pelo menos umas três ideias diferentes, e sinto que se tivesse me decidido antes, eu poderia ter começado mais cedo e o trabalho estaria bem mais adiantado. No final, escolhi o que eu escolhi principalmente porque eu poderia fazer em dupla com meu amigo, e portanto eu teria duas vezes mais motivos para me esforçar e fazer tudo direito.

Depois disso, passamos por maus bocados tentando substituir o SWIG. Reconhecer a gramática da linguagem C++ não é fácil, e causou bastantes dores de cabeça. No final das contas, se tivéssemos usado a metodologia de *milestones* desde o começo, não precisaríamos ter feito tanto esforço com essa parte do projeto, pelo menos não tão cedo e não tão intensamente como ocorreu.

Outra dificuldade foi o código que eu mesmo escrevi. A verdade é que aprendemos muito durante a graduação, e nossas habilidades como programadores melhoram incrivelmente rápido. A parte da OPA que eu havia escrito há dois anos no USPGameDev hoje me atormenta. Por que eu tinha que fazer as coisas de um jeito tão complicado e confuso!? Até hoje sinto uma preguiça instantânea de ter que mexer naquela seção do nosso código...

Eu diria que minha maior frustração é que temos uma data final para entregar esse projeto. Não porque seja difícil atingir nossas metas, mas sim porque temos que ter metas. Sem elas, perdemos a objetividade e o trabalho não rende. Eu preferiria ter desenvolvido com mais liberdade, experimentando e explorando as possibilidades. Esse projeto é particularmente significativo para mim, porque ele abre muitas portas para metaprogramação em C++. Tanto que definitivamente continuarei trabalhando nele depois que entregá-lo. Só fico desanimado por ter tido que podar muitas das minhas ideias em prol da entrega final. Por outro lado, eu aprendi muito sobre como ser objetivo em um projeto com uma escala bem maior que a de um exercício-programa, e isso com certeza é um aprendizado que usarei para o resto da minha vida.

7.2 Relação entre o trabalho de formatura e disciplinas do BCC

Seguem algumas opiniões minhas sobre como algumas disciplinas que cursei ao longo da minha graduação ajudaram nesse trabalho. Muitas vezes, o que fez mais diferença foi o professor que ministrou a disciplina, seja pelo que ele passou a mais ou pelo que ele passou a menos com relação às ementas originais.

MAC0122 - Princípios de Desenvolvimento de Algoritmos

É a primeira matéria onde tive contado com C, e na qual aprendi o essencial para desenvolver meus próprios algoritmos. Os exercícios-programa ajudaram muito a incentivar minha curiosidade como programador, assim como minha vontade por tentar entender mais a fundo algoritmos de diversos tipos e as maneiras de torná-los mais eficientes.

MAC0211 - Laboratório de Programação I

Essencialmente pela experiência de fazer um projeto de médio porte, e pelas diversas ferramentas ensinadas ao longo da ementa: Makefile, LaTeX, flex e bison, principalemente.

MAC0323 - Estrutura de Dados

Além de aprender a ideia por trás das estruturas de dados clássicas, um aspecto interessante da matéria foi que o professor ressaltou a importância de se fazer boas APIs. Ao longo de todo nosso trabalho, uma das minhas principais preocupações sempre foi garantir que o usuário do nosso sistema tivesse facilidade de entender e usar os serviçoes que forneceríamos.

MAC0316 - Conceitos Fundamentais de Linguagens de Programação

Nessa disciplina desenvolvemos um interpretador para uma linguagem de programação bem simples, montando a máquina virtual dela, apesar de não chegarmos a chamar ela disso naquela época. Isso ajudou muito a entender o que tem por trás de uma linguagem interpretada, o que facilitou o uso das APIs de `Lua` e `Python` em nosso trabalho.

MAC0335 - Leitura Dramática

Ensinou boas técnicas para se dirigir a um público, que eu deveria ter usado melhor na nossa apresentação...

MAC0342 - Laboratório de Programação Extrema

As práticas de programação extrema foram muito úteis em nosso trabalho, e acredito que teríamos rendido muito mais se tivéssemos usados elas direito desde o começo (infelizmente eu cursei essa disciplina durante o primeiro semestre desse trabalho, então só comecei a aplicar o que aprendi no meio do caminho). O mais importante que aprendi foi saber priorizar aquilo que agrega mais valor na hora de desenvolver um *software*, evitando a perda de tempo com código que dá trabalho e não acrescenta nada.

MAC0414 - Linguagens Formais e Autômatos

Entender o que são gramáticas, suas limitações e como implementar programas que as reconhecem foi primordial na hora de fazermos nosso gerador que precisava analisar código em `C++`. Por mais que tenhamos usado o Flexc++ e o Bisonc++ para gerar os analisadores léxico e sintático, não teríamos sido capazes de entender as mensagens de erro que eles produziam, e portanto não consiguiríamos corrigir a nossa parte dos analisadores sem o conhecimento que essa disciplina nos trouxe — pelo menos não sem ter que pesquisar um monte e quebrar a cabeça pra caramba.

MAC0424 - O Computador na Sociedade e Empresa

Tirando as partes mais... *peculiares* dessa matéria, teve duas coisas relevantes para o trabalho que aprendi com ela. A primeira é a definição informal de uma máquina de Turing (infelizmente minha turma não teve tempo de ver isso em MAC0414 e eu não fiz nenhuma disciplina de Complexidade Computacional), que me ajudou a entender melhor o que é uma "máquina que executa programas". A outra foi uma discussão que tivemos sobre as diferenças entre *dado*, *informação*, *conhecimento* e *competência*, que me ajudou a compreender mais claramente as verdadeiras (in)capacidades do computador. Isso me permitiu ter uma visão mais objetiva da computação em geral, o que me ajudou a escrever várias partes dessa monografia.

Capítulo 8

Próximos passos

Para o projeto como um todo:

- ***Scriptception***: possibilitar uma linguagem de *script* usar diretamente outra linguagem de *script* facilmente.
- **Mais linguagens**: Adicionar suporte padrão a outras linguagens além de `Python` e `Lua`, como `Ruby`, `Perl` e `Octave`.
- **Multithread**: Suportar uso *multithreaded* do Ouroboros.
- **Tutoriais**: Alguns tutoriais sobre como usar as diferentes partes do sistema também seria bem útil para ajudar os usuários.

Para completar o OPWIG:

- **Pré-processador**: O pré-processador de `C`/`C++` é uma parte importante da linguagem, podendo até definir trechos de código condicionalmente. Colocar um pré-processador no *parser* possibilitaria reconhecer códigos `C++` bem mais complexos. Nós vamos analisar a dificuldade de implementar essa funcionalidade, e se acharmos que não gastará muito tempo vamos colocá-la - para tal provavelmente vamos procurar alguma ferramenta pronta que ajude nisso.
- **Exportar `Enum`s**: `Enum` é uma estrutura de dados de `C++` que representa um tipo e seus possíveis valores. O *parser* do OPWIG já é capaz de reconhecer essas estruturas, porém o gerador de código ainda não exporta ela para as linguagens de *script*.
- **Classes completas**: ainda falta bastante o que implementar para podermos exportar classes mais realistas.

Referências

[Aab96] Anthony A. Aaby. *Introduction to Programming Languages*. 1996. Disponível em: < http://www.emu.edu.tr/aelci/Courses/D-318/D-318-Files/plbook/ >. Acesso em: 16 Nov. 2013. 5

[Bro13a] Frank B. Brokken. Bisonc++, 2013. Disponível em: < http://bisoncpp.sourceforge.net/ >. Acesso em: 17 Nov. 2013. 35

[Bro13b] Frank B. Brokken. Flexc++, 2013. Disponível em: < http://flexcpp.sourceforge.net/ >. Acesso em: 17 Nov. 2013. 34

[Dar11] Vinícius Kiwi Daros. Scrum no desenvolvimento de jogos eletrônicos. Trabalho de formatura, Instituto de Matemática e Estatística da USP, 2011. 25

[GHJV95] E. Gamma, R. Helm, R. Johnson e J Vlissides. *Design Patterns*. Addison-Wesley, 1995. 29

[IdFC] R. Ierusalimschy, L. H. de Figueiredo e W. Celes. Lua 5.1 reference manual. Disponível em: < http://www.lua.org/manual/5.1/ >. Acesso em: 19 Nov. 2013. 8

[pyt] Python v2.7.6 documentation. Disponível em: < http://docs.python.org/2/ >. Acesso em: 19 Nov. 2013. 8

[S.13a] Amaya S. The c++ resources network - a brief decription, 2013. Disponível em: < http://www.cplusplus.com/info/description/ >. Acesso em: 17 Nov. 2013. 5

[S.13b] Amaya S. The c++ resources network - history of c++, 2013. Disponível em: < http://www.cplusplus.com/info/history/ >. Acesso em: 17 Nov. 2013. 7

[Sta10] William Stallings. *Computer Organization and Architecture*. Prentice Hall, 8ª edição, 2010. 5

[swi] Simplified wrapper and interface generator. Disponível em: < http://www.swig.org >. Acesso em: 2 Nov. 2013. 25

[Wal13] Larry Wall. Programming is hard, let's go scripting, 2013. Disponível em: < http://www.perl.com/pub/2007/12/06/soto-11.html >. Acesso em: 17 Nov. 2013. 8