GObject: Implementação de funcionalidades de programação orientada a objetos em linguagem C

Gabriel Falcão Gonçalves de Moura[1]
Nesley Jesus Daher de Oliveira[2]

Resumo

O paradigma da orientação a objetos vem sendo utilizado em escalas cada vez maiores no contexto de desenvolvimento de *software*. Este artigo aborda aspectos e funcionalidades da biblioteca ***GObject***, uma implementação livre para a linguagem C e, apresenta algumas técnicas que permitem a utilização do paradigma de orientação a objetos em detrimento do paradigma imperativo nativo desta linguagem. O trabalho ainda apresenta uma série de justificativas que validará a utilização da linguagem C, no desenvolvimento de *softwares* orientados a objetos.

Palavras-chave: gobject, glib, orientação a objetos, linguagem C.

Introdução

As linguagens de programação orientadas a objetos possuem características sintáticas que têm como objetivo mesclar, de maneira sinérgica, as funcionalidades de orientação a objetos com a sintaxe de seu código, evitando o que, em tradução literal, *Fred Brooks* chama de **"erros acidentais"**, ou seja, não garantem que um software terá um projeto estruturalmente orientado a objetos, que usa as propostas deste paradigma da melhor forma possível ou que o programador terá capacidade de implementar modelos conceitualmente complexos. Em síntese, *Brooks* define que **"tarefas acidentais"** estão relacionadas com **"tarefas essenciais"**, pois a construção de softwares passa por atividades inerentes à elaboração modelos conceituais para entidades abstratas (tarefas essenciais), e por atividades de transcrição de modelos conceituais em linguagens de programação. (tarefas acidentais).

No contexto de desenvolvimento de *software* é comum a utilização de conjuntos de funções que auxiliam tarefas, tais como, manipular caracteres, exibir informações formatadas, manipular estruturas de dados etc. Um conjunto dessas funções de auxílio é conhecido como biblioteca de *software*. À medida que *softwares* livres surgem, funções não-específicas do *software* são modularizadas e distribuídas livremente na forma de bibliotecas de *software*, permitindo sua reutilização.

Entre as bibliotecas de *software* livre disponíveis para a linguagem C, está a ***GLib***, que possui uma

1 Aluno do curso superior de Gestão de Tecnologia da Informação do Centro Universitário de Belo Horizonte. Email:gabriel@nacaolivre.org

2 Professor do curso superior de Gestão de Tecnologia da Informação do Centro Universitário de Belo Horizonte. Email: nesdaher@gmail.com

quantidade muito grande de funções e estruturas de dados prontas para a utilização durante os processos de desenvolvimento de sistemas ou programas. No entanto, dentro da ***GLib***, há um auto-denominado "sistema de objetos", chamado ***GObject*** ou ***GLib*** *object-system*. Desta manera, o ***GObject*** trata-se basicamente de um sistema de funções, estruturas de dados e convenções de programação que pode ser utilizado como ambiente orientado a objetos para projetos que necessitem estar apoiados no paradigma orientado a objetos(OO) e concomitantemente no uso da linguagem C.

O paradigma OO propõe implementação de práticas e funcionalidades em código que permitem a representação abstrata de objetos reais, pois dessa maneira é possível descrever o comportamento e o funcionamento do *software*, de maneira mais próxima à sua finalidade, o que permite o mapeamento de objetos reais para um modelo abstrato que possibilita a definição da interação entre esses objetos, bem como seus comportamentos e estados. Assim, de acordo com a proposta do próprio paradigma OO a aplicação deste conjunto de práticas e funcionalidades deve permitir o desenvolvimento de *software* flexíveis, reusáveis e extensíveis [Dean 1996].

A ***GObject*** é então, um módulo da ***GLib*** que implementa, em linguagem C, funcionalidades que permitem o uso desta linguagem para atender ao paradigma OO. Desta maneira, de acordo com [Warkus 2004] ela pode permitir o uso de características tais como, tipagem dinâmica, tipos de dados voltados à introspecção, emissão de sinais e integração com o *sistema de tratamento de exceções da GLib*.

O tema escolhido para o desenvolvimento deste estudo, se justifica, pois, a linguagem de programação C não é de fato orientada a objetos, muito pelo contrário, foi proposta e construída para dar suporte ao paradigma imperativo, também conhecido com procedural. No entanto, com a utilização da ***GObject*** é possível criar *softwares* orientados a objetos em C, o que possibilita a utilização de práticas de análise de projetos, arquitetura e engenharia de *software* que necessitam que a implementação do código do sistema seja OO, tão necessário no cenário de desenvolvimento de software em âmbito mundial. Com isso, é justificada então a proposta apresentada, já que a mesma trata de um estudo para aplicação de uma técnica que revitaliza uma linguagem de programação muito difundida e que tem um grande parque aplicações desenvolvida, atualizando-a e possibilitanto sua sobrevida no mercado tecnológico atual.

Desta maneira nesta introdução foi apresentada uma breve introdução do trabalho de forma possibilitar a contextualização em cima do qual ele foi construído, traçando assim um perfil que mostra a motivação para adoção do tema e uma justificativa para isso. Na segunda parte são abordados assuntos pertinentes ao estudos como referência teórica da pesquisa desenvolvida. Em seguida na terceira parte, é apresentado um estudo aplicando assim as técnicas da ***GObject*** para demonstrar sua capacidade em construir código em C com perspectivas OO. Por fim, é apresentado a conclusão do trabalho onde também são sugeridas algumas continuidades a ele como proposta de trabalhos futuros.

2. Linguagens de Programação

Para que os computadores possam ser utilizados, é necessário executar instruções lógicas que permitem o uso dos recursos de processamento para criar resultados esperados. Tais instruções lógicas são

chamadas de programas e suas aplicações em computadores abrangem desde usinas elétricas e nucleares até a armazenagem de registros de talões de cheque pessoais. Devido a essa diversidade de contextos e possibilidades de aplicações, diferentes linguagens de programação, foram e continuam a ser criadas para encaixarem na medida certa da necessidade à que são destinadas.

No contexto de linguagens de programação, existem diferentes domínios, paradigmas e critérios de avaliação para cada tipo de linguagem. Para definir termos e explicar o ***GObject***, é necessário o entendimento de um conjunto de conceitos que são apresentados a partir deste ponto.

2.1. Legibilidade

A legibilidade é um critério de avaliação de linguagens de programação que determina a facilidade de entendimento do código escrito, o que reflete na manutenção e aprimoramento do código programa.

2.2. Ortogonalidade

Conforme Sebesta (2000, p.26), *ortogonalidade em uma linguagem de programação significa que um conjunto relativamente pequeno de construções primitivas pode ser combinado em um número relativamente pequeno de maneiras para construir as estruturas de controle e de dados da linguagem*.

Desta forma, em poucas palavras, é então uma característica de flexibilidade da linguagem de programação. Um exemplo aplicável ao contexto da implementação do ***GObject***, é justamente a própria linguagem C, cuja ortogonalidade possibilita sintaticamente a implementação de orientação a objetos; neste ponto é importante mencionar que isso acontece não pela perspectiva da programação orientada a objetos, mas pela maneira como implementar um sistema de tipos dinâmicos e outras características existentes no ***GObject***.

Como exemplo de ortogonalidade, a linguagem C têm a capacidade de dar suporte à manipulação de ponteiros através de referenciamento e "derreferenciamento", que basicamente consiste em obter o endereço de memória de uma variável, e obter o conteúdo da variável através de seu endereço em memória. No exemplo do código em C abaixo, os operadores * e &, para manipulação de ponteiros foram utilizados de maneira redundante o que sem o nível de ortogonalidade que a linguagem permite, poderia ocasionar um erro de sintaxe.

```
void mostrar (char **x, int *i)
{
   printf ("Nome: %s\n", **&*&x);
   printf ("Idade: %i\n", **&*&i);
}
int main (int argv, char **argc)
{
   char *nome = strdup("Juquinha");
   int num = 10;
   int idade;
   idade = *&num;
```

```
    mostrar (&nome, &idade);
    return 0;
}
```

3. História do *GObject*

Conforme [Steiner 2008] em Julho de 1995, Peter Mattis idealizou um *software* livre para edição de imagens chamado GIMP[3] que foi inicialmente desenvolvido para trabalhar em ambientes baseados em sistema unix. Assim, para projetar e construir um programa gráfico que utilize os conceitos de Janelas, botões, campos de texto etc é necessário utilizar alguma biblioteca que disponibilize funcionalidades para tal. Segundo Noble, bibliotecas com essa finalidade são denominadas **GUI** (ou *graphical user interface toolkits*).

As primeiras versões do GIMP utilizavam o toolkit gráfico Motif[4] que era o padrão para sistemas Unix. Apesar disso, segundo [Steiner 2008], a adoção deste toolkit gerou alguns problemas:

- A utilização do toolkit causava vários erros críticos no programa devido à sua incompatibilidade com o sistema de *plugins*[5] implementado no GIMP.
- A licença do Motif era incompatível com a licença GNU/GPL[6], pela qual, o GIMP é coberto.

A partir da versão 0.60 do GIMP, o *software* foi divulgado entre listas de discussão de universidades, ganhando reconhecimento e interesse, principalmente pelos estudantes. Como qualquer *software* livre, permitiu que programadores interessados pudessem contribuir com o seu desenvolvimento. Durante esse período, Peter Mattis, então mantenedor do projeto, resolveu criar sua própria toolkit gráfica: *GTK (ou GIMP Tool Kit).* [Steiner 2008].

A GTK se tornou, então, uma biblioteca para a criação de elementos gráficos de

---

3 Software livre e multiplataforma, para processamento de imagens. Site oficial: http://gimp.org

4 Motif é um toolkit gráfico comercial e proprietário para a criação de aplicações em ambiente Unix

5 Plugins são componentes de *software* que podem ser instalados pelo usuário, para adição de funcionalidades ao *software*.

6 GPL é uma licença de *software* livre que garante a visualização, modificação, utilização e redistribuição do *software*, um trecho desta licença se encontra no apêndice A

forma a propiciar um meio para interação com o usuário, baseada uma sub-biblioteca chamada *GDK (ou GIMP Drawing Kit)*, responsável por *"desenhar"* elementos gráficos como quadrados, círculos e outras formas geométricas, além de trabalhar com manipulação de arquivos de imagem. Além disto, permite também o controle da tela do computador em nível de sistema operacional.

Tanto o GIMP quanto a GTK e a GDK foram escritos em linguagem C, mas devido à necessidade dos desenvolvedores usarem conceitos de orientação a objetos, foi implementado um sistema interno de orientação a objetos dentro da GTK, o que fez o toolkit suportar um nível maior de abstração quanto à relação e interação entre seus componentes de interface gráfica. Técnicamente, cada componente de interface gráfica (User Interface - UI) se tornou uma classe, e o suporte a herança e encapsulamento, que consistem em duas fortes características da orientação a objetos, começaram então a ser introduzidos nesta biblioteca.

A GTK manteve o sistema de objetos embutido até a verão 1.2, quando seus desenvolvedores resolveram separar do sistema de objetos, um conjunto de funções utilitárias e estruturas de dados do toolkit gráfico, resultando em uma nova biblioteca e um sistema de objetos plugável, denominadas de *GLib* e *GObject*, respectivamente.

4. A linguagem C

De acordo com Kernighan e Ritchie (1988), por volta da década de 1970, o sistema operacional UNIX ainda era desenvolvido em linguagem de montagem, também conhecida como linguagem Assembly, que se trata de uma linguagem de baixo nível, baseada em instruções que simbolizam códigos numéricos inerentes ao processador do computador, que são montadas numa sequência lógica para produzir o resultado desejado. Assim, para cada plataforma de hardware que precisasse trabalhar com esse sistema, seriam necessárias grandes mudanças nas instruções de montagem, foi quando Dennis Ritchie criou a linguagem C, baseada na linguagem B[7], de seu amigo e co-desenvolvedor do UNIX, Ken Thompson.

Meyer (2000) define três características a serem consideradas para a linguagem C: i) *"alto-nível"*, por possuir estruturas de controle comparáveis a Algol e Pascal; ii)

7 Predecessora da linguagem C, desenvolvida por Ken Thompson na Bell Labs

orientada a máquina, por permitir a manipulação de dados no nível mais elementar, através do uso de endereços, apontadores e bytes; e iii) *portável*, pois sua orientação a máquina é uma abstração a instruções específicas de cada plataforma de hardware.

Dessa forma, a linguagem C é escolhida como preferencial para a criação de programas feitos para o ambiente Unix, entretanto a simplicidade da linguagem exige que o programador crie estruturas de dados complexas, tais como listas encadeadas e árvores enárias, ou utilizar bibliotecas que já possuem estruturas de dados pré-definidas. Portanto, a utilização do C como linguagem de programação requer um conhecimento básico sobre estruturas dados complexas, algoritmos de manipulação de dados e de acesso à endereços de memória.

5. Orientação a Objetos

Segundo [Jacobson 2004], a orientação a objetos consiste em uma técnica de modelagem, onde um sistema é composto por objetos que por sua vez interagem entre si. A orientação a objetos permite projetar sistemas onde a abstração do funcionamento pode ser fácilmente explicada e modelada e os objetos podem literalmente ser desenhados para demonstrar o funcionamento do sistema. Esse paradigma permite também a implementação de um sistema guiado por componentes que tem como característica a grande facilidade de manutenção e evolução, haja vista que possibilita de forma pragmática a aplicação do conceito de reusabilidade, onde uma estrutura pode ser reutilizada sem que necessite de alterações estruturais, ao contrário, "apenas" é utilizada. Isso potencializa a capacidade evolutiva do paradigma mencionada. Um sistema modelado com técnicas propostas pela orientação a objetos são, em geral, simples de entender, pois podem ser comparados à realidade graças à idéias propostas pela OO, como: Herança, Polimorfismo, Encapsulamento etc.

5.1. Objeto

No contexto de OO, um objeto é uma entidade capaz de guardar estados(informações) e de realizar operações(comportamento) que podem se utilizar dos estados, bem como modificá-los. Segundo [Dean 1996], as operações de um objeto são denominadas *métodos* e suas informações são denominadas *atributos*.

Para considerar um sistema orientado a objetos é necessário levar em conta pelo menos 4 características:

- Classificação;
- Herança;
- Polimorfismo comportamental;
- e Encapsulamento.

5.2. Classe e instância

Em função da definição do que é um objeto, é possível identificar a possibilidade da existência de vários objetos com o mesmo comportamento (estados e operações), assim, a partir de um molde, objetos são criados, e após as definições de suas operações e estados iniciais, tal molde é denominado então **Classe**.

Ivar Jacobson diz que em algumas ocasiões classes são chamadas de tipos-objeto, no entanto, um tipo é relacionado às possibilidades de manipulação com determinado tipo de dado, enquanto uma classe define, além da forma de se manipular as informações(estados), o comportamento do objeto e suas operações. Sendo assim, é plausível conceber uma classe como sendo uma implementação específica de um tipo.

Um objeto pertencente a uma classe é, também, chamado de **instância**. Sendo assim, enquanto uma classe descreve o comportamento e as informações de uma estrutura, a instância (objeto) possui suas informações manipuladas através de suas operações.

Assim, concluindo a definição de classe e de instância, de forma análoga a objetos do mundo real, a *classe* é um molde para criar novos objetos, enquanto *instância* é um sinônimo de objetos.

5.3. Herança

Uma das características providas pela orientação a objetos é a da possibilidade do uso de herança para resolver o problema de classes compartilharem características iguais. Tomando como exemplo a definição de classes: Animal, Carnivoro, Herbívoro, ambas

Carnivoro e Herbívoro são variações de animal. Animais, sejam carnívoros ou herbívoros, podem se locomover, dormir e acordar, por exemplo. Assim a herança, conforme Jacobson (2004), consiste na possibilidade de definir classes que herdam características de uma classe base, também denominada de superclasse, classe mãe ou classe pai, onde as primeiras são denominadas de classes derivadas, subclasses ou classes filhas. Desta forma, a principal vantagem da herança é evitar a duplicidade de definição de operações e de dados, podendo simplesmente herdar características de uma super-classe e especializar métodos e atributos.

5.4. Polimorfismo

A característica de polimorfismo é responsável por permitir a interação com objetos sem a necessidade de conhecer sua classe origem, através de comportamento dinâmico. Em poucas palavras, classes diferentes podem possuir métodos homônimos que possuem comportamento diferentes e trazem resultados diferentes. Jacobson (2004) ressalta a importância do polimorfismo na comunicação entre objetos, pois há uma uniformidade de invocação de métodos em diferentes objetos de diferentes classes, e a definição da lógica fica limitada ao método que é invocado (receptor), ao invés de ser estendida ao objeto que invoca(transmissor). Assim, o polimorfismo, junto com o encapsulamento, que será abordado a seguir, exerce os importantes papéis de tornar o código mais claro e inteligível, além aumentar o nível de manutenibilidade na dinâmica proposta pela orientação a objetos.

5.5. Encapsulamento

Segundo [Dean 1996] e [Jacobson 2004], o encapsulamento é a capacidade e definir restrições de visibilidade, de acesso e modificação de atributos e métodos de classes. Encapsular métodos e atributos faz com que haja uniformidade de acesso às informações e operações de um objeto e permitindo a execução de sub-operações antes, durante e depois do acesso, bem como a possibilidade de modificação da lógica das operações de um objeto sem a necessidade de modificar os objetos dependentes de tais operações.

5.6. Introspecção

Segundo (SAGAR, HILERIO, 2008), instrospecção é uma técnica de reutilização de

códigos e dinamismo de desenvolvimento que se baseia em obter dados sobre o objetos em tempo de execução[8].

Em poucas palavras, quando uma linguagem ou ambiente provê introspecção ao desenvolvedor, é possível obter nomes de objetos, métodos, atributos, classes, super-classes etc. Tal característica pode ser aproveitada, por exemplo, para serializar um objeto, gravando seu estado atual num arquivo XML, e posteriormente recuperar os dados serializados e recriar um objeto dinâmicamente em tempo de execução.

6. Orientação a objetos em ambientes não orientados a objeto

A construção de *software* passa por dois tipos de tarefas:As tarefas regidas por ações **acidentais** e as tarefas regidas por ações **essenciais** [Brooks 1995]. A primeira são aquelas dependentes de ambiente, ferramentas e quaisquer características facilitadoras, isto é, linguagens de sintaxe simplificada, editores e ferramentas de desenvolvimento com checagem de sintaxe e até mesmo de semântica, recursos do tipo arrastar-e-soltar e quaisquer outras utilidades que tornem o desenvolvimento de *software* mais confortável e menos sucetível a erros comuns. A segunda consiste naquelas tarefas que independem de ferramentas e recursos facilidadores de desenvolvimento, e estão relacionadas ao conhecimento de análise, engenharia e arquitetura de desenvolvimento de *software*.

Segundo [Brooks 1995], linguagens de programação orientadas a objetos só resolvem os problemas classificados como *acidentais* e ressalta que *software* podem ser desenvolvidos usando as propostas da OO mesmo em linguagens de programação que não a suportam sintáticamente. Para tanto, é necessário que o desenvolvedor possua maturidade o suficiente para utilizar um ambiente OO implementado por alguma biblioteca [Meyer 2000].

7. O *GObject* object-system

Segundo [Hendrickx 2004], ***GObject*** é uma implementação de funcionalidades de orientação a objetos em linguagem C. Com ***GObject*** é possível alcançar um nível produtivo de desenvolvimento de *software* usando orientação a objetos. *Matthias Warkus* explica que tal nível de produção de *software* com ***GObject*** pode ser alcançado

8 É o tempo de vida de um programa após a sua execução.

com a ajuda de **Boilerplate Codes**, que são **códigos-molde** ou **templates** a serem modificados para criar um *GObject*.

7.1. *GType*, o sistema de tipos dinâmica do *GObject*

Ao criar classes a partir do ambiente provido pelo sistema de objetos *GObject*, o desenvolvedor estará criando novos tipos, que são estruturas de dados comuns à linguagem C, mas que possuem convenções abordadas a seguir inferindo no funcionamento de toda a dinâmica do *GObject*.

Segundo a documentação oficial (CHAPLIN, TAYLOR, 2008), na estrutura da *GLib*, biblioteca que sustenta o *GObject*, está um sistema que implementa um suporte a registro dinâmico de tipos, denominado *GType*. Assim, o *GType* funciona como um "cartório" de tipos, que faz registro e cria uma identificação interna à biblioteca *GLib*. O funcionamento desse sistema se dá através de funções da biblioteca que servem de interface para o registro de tipos, e assim permite criar uma coleção de meta-dados sobre cada tipo registrado como forma de recuperá-los posteriormente em caráter de instrospecção.

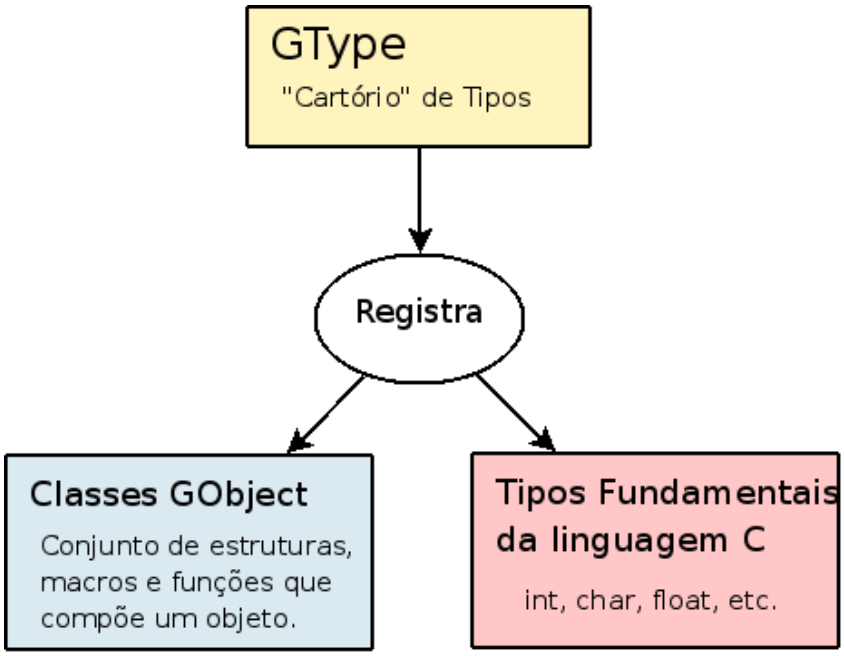

Figure 1: A relação entre o registro de tipos fundamentais e classes *GObject no sistema GType. Fonte: autoria própria*

Reiterando a afirmação de [Brooks 1995], a orientação a objetos não é uma característica dependente de linguagem, é um paradigma associado com disciplina e conhecimentos em arquitetura de *software*. Neste caso, o ***GType*** é um exemplo de estrutura que visa prover um caminho a ser trilhado pelo programador, dando todo o suporte necessário para a utilização do sistema de tipos.

7.1.1. Funcionamento do *GType*

O ***GType*** possui um sistema interno de registro de tipos, onde cada novo tipo criado deverá ser registrado usando funções que o ***GType*** provê para tal papel. São elas:

1. **g_type_register_dynamic ()**
   Faz o registro de tipos que serão implementados em bibliotecas compartilhadas e utilizados no programa corrente. É uma maneira de registrar tipos que estarão cobertos pelo sistema de plugins do ***GObject***.
2. **g_type_register_static ()**
   Faz o registro de tipos estáticos ao programa corrente. É através dela que novas classes ***GObject*** criadas serão registradas.
3. **g_type_register_fundamental ()**
   Faz o registro dos tipos básicos da linguagem C, fundamentais à ***GLib***:
   - *gchar:*
     redefinição do tipo ***char***;
   - *guchar:*
     redefinição do tipo ***unsigned char***;
   - *gint:*
     redefinição do tipo ***int***;
   - *guchar:*
     redefinição do tipo ***unsigned int***;
   - *gboolean:*
     redefinição do tipo ***gint*** com a diferença de que será tratado como booleano, e utiliza macros para manter a convenção: 0 para FALSE e 1 para TRUE;
   - *glong:*
     redefinição do tipo ***long***;

- *gulong:*
  redefinição do tipo ***unsigned long***;
- *gint64:*
  Um tipo definido na ***Glib*** garantido a ser 64 bits em qualquer plataforma que suporte à biblioteca;
- *guint64:*
  É o tipo gint64 mas que suporta 64 bits de um inteiro sem sinal;
- *gfloat:*
  Redefinição do tipo ***float***;
- *gdouble:*
  Redefinição do tipo ***double***;
- *gchararray:*
  Redefinição do tipo ***char ****;
- *gpointer:*
  Redefinição do tipo ***void ****;

Os tipos redefinidos na ***Glib*** precisam ser registrados no sistema GType para ganharem representações internas à biblioteca ***GLib***, que vai refletir principalmente no suporte à introspecção da ***GLib***.

Além de denominar o sistema de tipos da ***GLib***, ***GType*** é um tipo de variável que guarda um número inteiro maior que zero, tal número é o identificador individual de cada tipo criado e registrado com as funções citadas acima, em poucas palavras, ***GType*** é o tipo dos tipos, no entanto, por ser uma descrição complexa, o exemplo abaixo dá uma melhor compreensão do que é o tipo de dados ***GType***:

```
/* Trecho de um registro de um
   GObject hipotético,
   chamado Pessoa
*/
```

```
/* identificador do meu GObject
   dentro do sistema de tipos
   da GLib */
   identificador = 0;

identificador = g_type_register_static (G_TYPE_OBJECT, "Pessoa", &info, 0);
```

No exemplo hipotético acima, a variável *identificador* é do tipo ***GType***, e após receber o valor dado pela função de registro, guardará um número de identificação, pelo qual é possível uma posterior recuperação de dados a respeito do tipo que foi registrado. O ***GType*** se baseia num registro interno de tipos, mas como é uma implementação sem suporte direto da linguagem, é necessário que o desenvolvedor faça uma chamada à função **g_type_init ()**, que funciona como um gatilho que vai desencadear o registro interno dos tipos básicos que foram criados [Chaplin e Taylor 2008].

Assim, o ***GType*** é uma maneira de permitir o registro das classes ***GObject*** criadas por um desenvolvedor e de criar instâncias a partir do número identificador daquela determinada classe.

7.2. O sistema de objetos do *GObject*

Utilizar o sistema ***GObject*** para de fato, criar classes e utilizar objetos requer do desenvolvedor mais esforço e mais linhas de código que em outras linguagens que possuem sintaxe apropriada para tal tarefa, mas apesar de sintáticamente isso ser uma desvantagem do uso de ***GObject***, é possivel ignorar a parte de utilização de properties, mas que não é indicado por questões de perda de funcionalidades providas pelo sistema de objetos do ***GObject***.

Enfim, quando um desenvolvedor optar por utilizar orientação a objetos através do ***GObject***, os seguintes pontos devem ser levados em consideração:

7.2.1. Utilização de códigos-molde

a) Apesar de ser necessária uma quantidade relativamente grande de código para criar uma classe ***GObject***, é possível e perfeitamente plausível o uso de códigos molde, ou seja, um esqueleto pronto de ***GObject*** pronto para sofrer pequenas modificações e que no final vai culminar no resultado desejado.

7.2.2. Automatizações de tarefas relativas à gerência de memória

b) Por se tratar de linguagem C, os desenvolvedores devem se preocupar em cuidar de liberar a memória que foi alocada por eles mesmos. No entanto, *GObject* provê um sistema interno de contagem de referências e gerenciamento de alocação de memória, dando um suporte diferencial para a tarefa de gerenciamento de memória, feita pelo desenvolvedor (KERNIGHAN, RITCHIE, 1988).

7.3. A estrutura de um *GObject*

Um *GObject* é composto por duas estruturas de dados, sendo uma para classe e outra para instância. A estrutura de classe serve para guardar métodos de sinais e outros métodos que o desenvolvedor achar conveniente. A estrutura de instância possui os atributos do objeto e tudo o mais que será mantido em memória para cada instância existente.

Uma peculiaridade nas estruturas de classe e instância, é que o primeiro membro de cada estrutura, deve ser a estrutura mãe de classe e instância, caracterizando parte da organização feita pelo sistema de objetos para prover suporte à herança. Logo, todo objeto que for criado deve herdar diretamente de um *GObject*.

Exemplo:

```
typedef struct _Pessoa {
   GObject parent;
   /* outros membros, atributos etc */
} Pessoa;

typedef struct _PessoaClass {
   GObjectClass parent_class;
   /* protótipos de sinais, etc */
} PessoaClass;
```

7.4. Métodos

Os métodos de objetos *GObject* se baseiam numa sintaxe de convenção de *namespaces*[9]:

9 Nomes que simbolizam um local abstrato, reservado para conter um conjunto de identificadores ucomo funções, variáveis, classes etc.

```
ns_objeto_nome_do_metodo (Objeto *,
  ...)
```

Onde:

- ***ns*** é o nome do namespace ao qual o objeto pertence
- ***Objeto ** *** é uma referência à instância na qual o método faz parte

Tais convenções garantem que o ***GObject*** criado não terá métodos conflitantes com os de outros objetos ***GObject*** diferentes, o que também reflete no polimorfismo, e que todo método receba como primeiro parâmetro, a instância ao qual ele faz parte, possibilitando, de fato, que o método haja como um método de objeto, e não simplesmente como uma função.

7.4.1. Métodos privados

Na linguagem C, o modificador **static** faz com que uma função só esteja acessível dentro do módulo atual[10]. A documentação oficial do ***GObject*** propõe a definição de classes ***GObject*** em módulos diferentes, permitindo utilizar o modificador **static** para tornar um método privado.

7.5. Properties

Pelo fato de um ***GObject*** ser, basicamente, uma estrutura alocada em memória, basta dicionar membros arbitrários à estrutura, e os atributos estarão acessíveis e visíveis. Mas além de isso não garantir o encapsulamento, perde-se o suporte à introspecção que o sistema de objetos provê. Assim, para criar atributos na classe que suportem as funcionalidades oferecidas pelo sistema de objetos ***GObject*** é preciso criar *properties*.

Bem como a criação de um ***GObject*** básico, a criação de properties demanda uma quantidade relativamente grande de código, mas é plenamente justificável através dos benefícios oferecidos por sua arquitetura.

7.5.1. *GValue*, *containers* genéricos para variáveis

Para uniformizar a manipulação das variáveis de cada *property*, e para dar suporte à instrospecção, o sistema de objetos faz uso do tipo ***GValue***. O ***GValue*** é uma estrutura

10 Arquivo que contém a definição de código corrente.

de dados que atua como envólucro para a variável que será a *property*, e possui dados sobre qual é o tipo daquela variável e qual o seu tamanho.

7.5.2. *GParamSpec*, estruturas com metadados sobre parâmetros

Para possibilitar a especificação de detalhes de cada *property*, relativos à permissões, valor padrão, valor mínimo e máximo, e o tipo ***GType*** que tal *property* manipulará, existe a estrutura ***GParamSpec***.

Com o ***GParamSpec*** é possível determinar se uma *property* será somente leitura, somente escrita, leitura e escrita, definição restrita ao momento de instanciação do objeto, etc.

7.6. Sinais

Além de possuir suporte a métodos para a troca de mensagens entre objetos, o sistema do ***GObject*** possui uma forma adicional para troca de mensagens entre objetos, que consiste expor uma lista de sinais que podem ser emitidos pelo objeto, e permitir que uma fila de uma ou mais *callbacks*[11] sejam executados antes, durante e após a emissão de quaisquer sinais.

Assim, sinais podem ser concebidos como eventos que ocorrem em uma instância de classe ***GObject***, e que quando ativados, se propagam por outros objetos precisam interagir com tais eventos.

8. Exemplo de utilizacão de *GObject*

Para fins de exemplificação de manipulação de objetos ***GObject*** neste trabalho, uma classe foi denominada *GCarro* foi implementada. A entidade conceituada é a de um carro, que possui 2 estados: a) ligado e b) desligado, um atributo inerente à velocidade, e quatro ações: a) ligar, b) desligar, c) acelerar, d) reduzir, sendo as ações ***a*** e ***b***, referentes ao estado do carro, e as ações ***c*** e ***d*** relacionadas à mudança da velocidade do carro.

O código de definição da classe GCarro se encontra no anexo A deste artigo, e o trecho de código abaixo é um exemplo da manipulação do objeto GCarro.

---

11 Callbacks são funções a serem executadas através do acionamento de um gatilho. Exemplo: Numa interface gráfica, quando o botão *"fechar"* for clicado, um callback que fecha a janela será acionado.

```
#include "g-carro.h"

int
main(int argc, char **argv){

  g_type_init ();

  GCarro *carro1, *carro2;

  carro1 = g_object_new (G_TYPE_CARRO, NULL);
  carro2 = g_object_new (G_TYPE_CARRO, NULL);

  /* manipulando o objeto carro1 */
  g_print ("\n= Manipulando o objeto: carro1 =\n");
  g_carro_ligar (carro1);
  g_carro_acelerar (carro1, 40);
  g_carro_reduzir (carro1, 20);
  g_carro_desligar (carro1);

  /* manipulando o objeto carro2*/
  g_print ("\n= Manipulando o objeto: carro2 =\n");
  g_carro_acelerar (carro2, 10);
  g_carro_ligar (carro2);
  g_carro_acelerar (carro2, 20);
  g_carro_reduzir (carro2, 20);

  g_print ("\nLiberando memoria...\n");
  /* destruindo objetos */
  g_object_unref (carro1);
  g_object_unref (carro2);

  return 0;
}
```

Após a compilação e execução, o código acima gera o seguinte resultado:

```
= Manipulando o objeto: carro1 =
```

```
Carro ligado!
Carro andando a 40 KM/h
Carro andando a 20 KM/h
Carro deligado!

= Manipulando o objeto: carro2 =
Ligue o carro antes de acelerar !
Carro ligado!
Carro andando a 20 KM/h
O carro morreu!

Liberando memoria...
Objeto GCarro eliminado!
Objeto GCarro eliminado!
```

9. Comparação entre classes Python e *GObject*

Ainda utilizando a classe GCarro como ponto de comparação, uma implementação na linguagem de programação python[12] será utilizada para comparar aspectos entre tarefas essenciais e acidentais, com o intuito de validar a possibilidade de utilização GObject para uma abordagem OO, tanto quanto é possível fazê-la na linguagem Python.

O código que segue será usado como ponto de comparação entre classes Python e ***GObject***, e trata-se da manipulação de objetos da classe GCarro definidos através da linguagem Python, localizados no anexo B deste artigo, que por sua vez são uma implementação em Python da classe GCarro, originalmente implementada em ***GObject***.

Dentre as várias linguagens de alto-nível e orientadas a objetos existentes no mercado, Python foi escolhida pela sua simplicidade de entendimento e pequenas semelhanças sintáticas com a definição e uso de classes em GObject.

```
carro1 = GCarro()
carro2 = GCarro()
```

12 Linguagem de programação de alto nível, orientada a objetos, que prima pela sintaxe simples e intuitiva. Website: http://www.python.org

```
print "\n= Manipulando o objeto carro1 ="
carro1.ligar()
carro1.acelerar(40)
carro1.reduzir(20)
carro1.desligar()

print "\n= Manipulando o objeto carro2 ="
carro2.acelerar(10)
carro2.ligar()
carro2.acelerar(20)
carro2.reduzir(20)

print "Python é uma linguagem de alto nível e vai liberar a memória automáticamente"
```

Após a execução, o código acima gera o seguinte resultado:

```
= Manipulando o objeto carro1 =
Carro ligado!
Carro andando a 40 KM/h
Carro andando a 20 KM/h
Carro desligado!

= Manipulando o objeto carro2 =
Ligue o carro antes de acelerar
Carro ligado!
Carro andando a 20 KM/h
O carro morreu!
Python é uma linguagem de alto nível e vai liberar a memória automáticamente
Objeto GCarro eliminado!
Objeto GCarro eliminado!
```

9.1. Análise comparativa entre a implementação em Python e *GObject* 9.1.1. Inicialização de objetos

A inicialização de objetos em Python é feita através do método __init__, que trata-se de um método reservado da linguagem Python, e sua definição é opcional. A inicialização

de objetos em *GObject* é feita através de um método cujo nome é a composição ns_cls_init, onde *ns* é o namespace do objeto, correspondido pela letra "g" de *GCarro* e *cls* é o nome do objeto, correspondido pela palavra "carro" de *GCarro*. A obrigatoriedade da definição do método em GObject está relacionada a como a classe é definida: *a)* Obrigatória, através da macro *G_DEFINE_TYPE* ou *b)* através da definição de uma função detalhada para registro de tipo, explicitando os métodos de inicialização e destrução, caso necessários.

9.1.2. Encapsulamento

Em Python o encapsulamento de atributos e métodos é feito através da utilização de dois caracteres "_" (*underscore*) como prefixo de seus nomes, tordando os membros da classe acessíveis somente dentro do seu escopo. Em *GObject*, a abordagem de utilização de encapsulamento de atributos é possível através da utilização de properties, que por sua vez possuem um conjunto de metadados GParamSpec, determinando seu nível de encapsulamento. Na implementação da classe GCarro as properties usam os parâmetros *G_PARAM_READABLE* e *G_PARAM_WRITABLE*, determinando que possuem permissão de acesso e modificação externa, caracterizando as como públicas.

9.1.3. Definição de métodos

A definição de métodos em Python é feita através da palavra-chave def, e todos os métodos recebem obrigatoriamente como primeiro parâmetro, uma referência ao objeto. De maneira similar, métodos de classes *GObject* também recebem como primeiro parâmetro uma referência ao objeto, com a diferença desse parâmetro ser opcional, uma vez que é apenas uma convenção em razão da linguagem C não suportar orientação a objetos e concomitantemente uma sintaxe específica para definição de métodos.

9.1.4. Definição de *properties*

A definição de *properties* em Python é feita através da criação de um método responsável por modificar o valor de um atributo, um método responsável por obter o valor de um atributo e uma chamada à função property, que por sua vez recebe como parâmetro os 2 métodos previamente definidos.

Em *GObject*, a definição de properties é feita dentro da função de inicialização da classe, através de chamadas à função g_object_class_install_property, que instala cada

property dentro da classe. A modificação e recuperação dos valores de cada property, é feita através da criação de um único método resposável por modificação e outro para recuperação dos valores, e dentro de ambos é definida uma grande estrutura de decisão, onde a partir de um identificador, o atributo respectivo será utilizado.

As duas abordagens têm a vantagem de permitir implementação de algoritmos durante o acesso e modificação das properties, e a abordagem da GObject possui uma vantagem extra de usar um meio uniforme para alterar e acessar properties, através da função g_object_set e g_object_get, respectivamente.

10. Estudo de caso da LibAnvil

A LibAnvil trata-se de uma biblioteca, em construção[13], que servirá de apoio a *softwares* desenvolvidos em linguagem C utilizando a ***GObject*** para implementar a perspectiva OO.

Atualmente a LibAnvil implementa suporte a serialização de objetos criados com ***GObject*** de maneira modular e extensível, com suporte a formatos de persistência customizados através de implementações de extensão à biblioteca. Essa dinâmica é demonstrada na figura 3. O desenvolvimento da LibAnvil tomou como foco a tarefa de estudar características do ***GObject*** de maneira arbitrária, mas acabou tomando um caminho bem traçado, devido à demanda por uso de características específicas do sistema de objetos ***GObject***.

Os tópicos abordados foram:

- Uso de códigos-molde
- Utilização de properties
- Utilização de herança como forma de reuso de código
- Utilização massiva de introspecção

10.1. Uso de códigos-molde

Por se tratar de um projeto de *software* livre, foi possível utilizar um código-molde existente na documentação oficial da ***GLib*** para criar cada uma das 11 classes ***GObject*** que compõe a LibAnvil em seu estado atual.

13 a codificação da LibAnvil foi desenvolvida pelo autor como atividade pertinente ao trabalho desenvolvido na empresa AlfaiaTI - Tecnologia Sob Medida. Website: http://alfaiati.net

O código-molde utilizado possui 121 linhas de código, totalizando 1331 linhas de código somente para definir as classes, e registrar tipos. Desconsiderando inclusive, linhas de código referentes à definição de properties.

O projeto possui um total de 2417 linhas de código, o que conclui que atualmente mais linhas de código são gastas para definir as classes *GObject*, que para implementar os algoritmos de negócio.

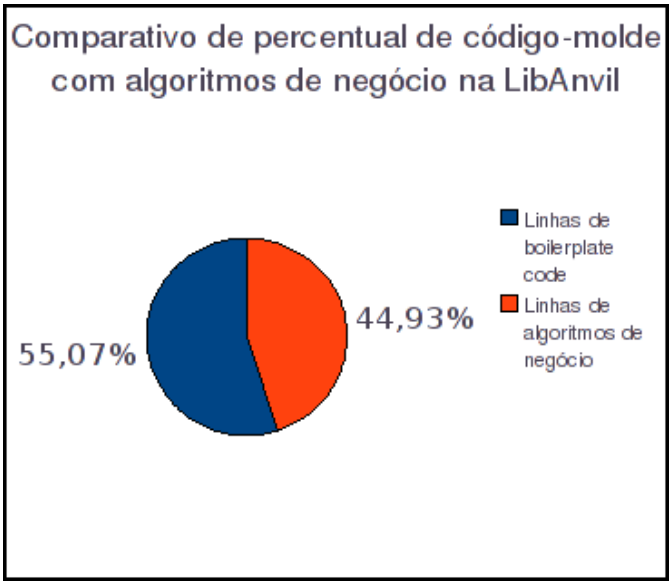

Figure 2: Gráfico comparativo entre linhas de código para definição de classes *GObject* e linhas de código responsáveis por regras de negócio. Fonte: Autoria própria

No estado atual, a LibAnvil possui ao todo 11 classes *GObject* definidas, e o mesmo código-molde foi utilizado para a criação de cada um deles, cujas linhas de código, somadas às linhas referentes aos algoritmos de negócio implementados, totalizam 2417.

Para todas cada objeto implementado, o mesmo código-molde foi utilizado. O código-molde utilizado, possui, sozinho, 121 linhas. Logo, conclui-se que das 2417 linhas de código da LibAnvil, 1331 linhas são referentes à definição de objetos e 1086 referentes aos algoritmos responsáveis pelo sistema de serialização de *GObject*.

10.2. Utilização de properties

A utilização de properties na LibAvil foi crucial para a uniformização e garantia de que a arquitetura definida para o projeto fosse cumprida.

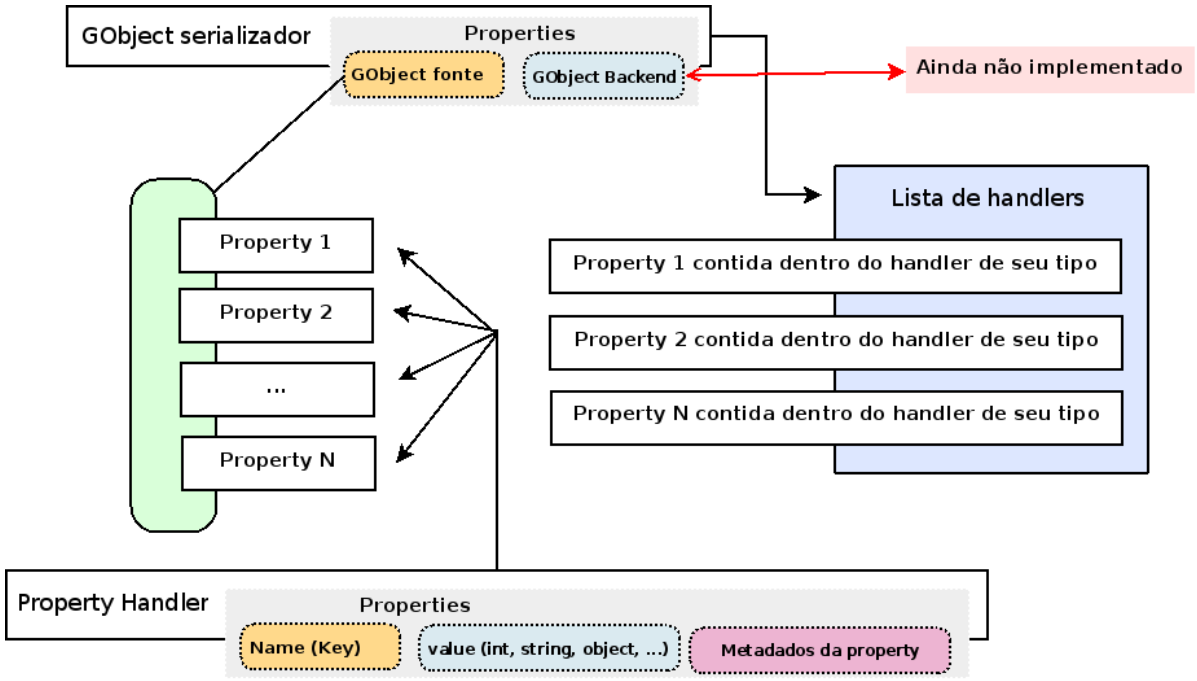

Figure 3: O fluxo de funcionamento do sistema de serialização implementado pela LibAnvil. Fonte: Autoria própria

O serializador é um ***GObject*** com duas properties: O objeto a ser serializado, e o *backend*[14] de serialização. O sistema de *backend* de serialização ainda não foi implementado, mas seu intuito é de prover serialização para qualquer formato, bastando criar novos *backends* para a LibAnvil, isso permitirá serializar, por exemplo, em XML, JSON e qualquer outro formato.

Quando o objeto fonte é definido no serializador, este se encarrega de invocar um método privado que faz introspecção no objeto fonte, iterando por cada uma de suas properties, identificando seu tipo e armazenando seu valor no *handler*[15] adequado para aquele tipo.

14 Módulos adicionais que podem ser desenvolvidos para dar novas funcionalidades à biblioteca

15 Handler, do inglês "manipulador". No contexto na libanvil, é o objeto responsável por manipular dados contidos em propriedades de outros objetos.

Os *handlers* são instâncias de classes ***GObject*** que servem de interface para lidar com properties de qualquer tipo e armazenam informações de todos os metadados de cada *property*.

10.3. Utilização de herança como forma de reuso de código

Os *handlers* variam de um pro outro com pequenas diferenças relativas ao tipo no qual cada um manipula. No entanto, os metadados da *property* manipulada são guardados num ***GParamSpec***, isso significa que é uma *property* idêntica para todos os ***handlers***. Para evitar a duplicidade de código e implementar reutilização de código, foi criado o ***GObject*** abstrato *BaseHandler*, que contém unicamente a *property* de metadados. Assim, todos os ***handlers*** devem ser subclasses diretas do *BaseHandler*.

10.4. Utilização massiva de introspecção

Para que o serializador possa, de fato, saber quais as properties existentes no objeto-fonte, lidar com seu conteúdo e criar ***handlers***, o sistema de properties utiliza funções de introspecção do ***GObject*** que listam as properties e retornam cada um dos seus metadados.

11. Conclusão

No contexto de programação, a orientação a objetos é um paradigma independente de linguagem, e possui uma perspectiva que torna possível mapear objetos reais em entidades abstratas de software. Assim, conforme os estudos realizados neste trabalho, a *GObject* é uma implementação do paradigma OO em linguagem C que pode ser utilizada em projetos de software que possuam entre seus requisitos esta linguagem de programação, visto que a *GObject* suporta funcionalidades que possibilitam a construção de códigos para mapear objetos na linguagem C.

Também, foi possível verificar que a codificação de classes em GObject demanda uma grande quantidade de linhas de código, em razão da linguagem C não suportar uma sintaxe apropriada para agilizar o uso das funcionalidades propostas pelo paradigma OO. No entanto, a convenção de nomes de variáveis possibilita codificar uma classe em GObject e reutilizar seu código, facilitando e agilizando a codificação de novas classes.

Como continuidade a este trabalho são propostos estudos sobre a implementação do paradigma OO em linguagem C, existente na seção de sistema de arquivos virtual

(*VFS, Virtual File System*) do *kernel Linux*, fazendo análise comparativa com a implementação do GObject de modo a obter propostas para melhorias no GObject. Estudos sobre ferramentas que automatizem a tarefa de criar classes e objetos com GObject, tecnologias provedoras de interoperabilidade entre linguagens de programação que implementem GObject, e comparações entre vantagens e desvantagens de sua utilização.

GObject: Implementation of object-oriented programming in C language

Abstract

The object-oriented paradigm is being used in increasingly larger scales into the software development context. This article discusses aspects and features from the GObject library, which is an free implementation for the C programing language, and presents some technical that turns the utilization of the object-oriented paradigm possible, in detriment of the imperative one, native from that language. The work also shows a serie of justifications, that valids the utilization of the C programming language on object-oriented software development.

Keywords: gobject, glib, object-oriented, C programming language.

References

[Brooks 1995] BROOKS, F. *Mythical Man-Month, The: Essays on Software Engineering*. Annyversary. [S.l.]: Addison Wesley Professional, 1995. 336 p. ISBN 0-201-83595-9.

[Chaplin e Taylor 2008] CHAPLIN, D.; TAYLOR, O. *The Official GObject Reference Manual*. http://library.gnome.org/devel/gobject/stable/: gimp.org, Maio 2008. Acessado em 11 de Maio de 2008.

[Dean 1996] DEAN, J. *Whole-Program Optimization of Object-Oriented Languages*. Tese (Doutorado) — University of Washington, 1996.

[Hendrickx 2004] HENDRICKX, S. *Glib-C: C as an alternative Object Oriented*

Environment. 2004.

[Jacobson 2004] JACOBSON, I. *Object-oriented Software Engineering*. [S.l.: s.n.], 2004. ISBN 0-201-54435-0.

[Meyer 2000] MEYER, B. *Object-Oriented Software Construction*. Prentice Hall PTR, 2000. ISBN 0136291554. Disponìvel em: <http://www.amazon.ca/exec/obidos/redirect? tag=citeulike09-20&path=ASIN/0136291554>.

[Steiner 2008] STEINER, J. *A Brief History of GIMP*. http://www.gimp.org/about/ancient_history.html: gimp.org, Maio 2008. Acessado em 11 de Maio de 2008.

[Steiner 2008] STEINER, J. *GIMP Prehistory, before GIMP 0.54*. http://www.gimp.org/about/prehistory.html: gimp.org, Maio 2008. Acessado em 11 de Maio de 2008.

[Warkus 2004] WARKUS, M. *The Official GNOME 2 Developer's Guide*. 1. ed. San Francisco: William Pollock, 2004. 497 p. ISBN 1-59327-030-5.

APÊNDICE A - Licença dos códigos anexados ao trabalho

Os códigos anexados ao trabalho foram desenvolvidos pelo autor do artigo, e estão licenciados como software livre, sob a licença GNU/GPL versão 2. Entre os termos da licença, encontrados no site oficial da Free Software Foundation[16], está uma indicação para que cada arquivo de código-fonte possua um cabeçalho indicando os termos básicos de sua licença.

Em razão de evitar redundância de informação, o cabeçalho da licença de cada código deste artigo se encontra centralizado abaixo, em versão original, em inglês dos Estados Unidos.

```
This program is free software; you can
  redistribute it and/or modify it under the terms of the GNU
  General Public License as published by the Free Software
  Foundation; either version 2 of the License, or (at your option)
  any later version.
```

16 Fundação sediada em Boston/MA, EUA, que possui a missão de promover a liberdade dos usuários de computadores e defender os direitos do software livre. Website: http://fsf.org

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public
License along with this program; if not, write to the
Free Software Foundation, Inc., 59 Temple Place - Suite 330,
Boston, MA 02111-1307, USA.

ANEXO A - Definição da classe GCarro em linguagem C com *GObject*

Arquivo: *g-carro.h*

```
#ifndef _G_CARRO_H_
#define _G_CARRO_H_

#include <glib-object.h> /* Incluindo cabeçalhos da biblioteca \inglesN{GObject}*/

G_BEGIN_DECLS
#define G_TYPE_CARRO            (g_carro_get_type ())
#define G_CARRO(obj)            (G_TYPE_CHECK_INSTANCE_CAST ((obj), G_TYPE_CARRO, GCarro))
#define G_CARRO_CLASS(klass)    (G_TYPE_CHECK_CLASS_CAST ((klass), G_TYPE_CARRO,
GCarroClass))
#define G_IS_CARRO(obj)         (G_TYPE_CHECK_INSTANCE_TYPE ((obj), G_TYPE_CARRO))
#define G_IS_CARRO_CLASS(klass)  (G_TYPE_CHECK_CLASS_TYPE ((klass), G_TYPE_CARRO))
#define G_CARRO_GET_CLASS(obj)   (G_TYPE_INSTANCE_GET_CLASS ((obj), G_TYPE_CARRO,
GCarroClass))

typedef struct _GCarroClass GCarroClass;
typedef struct _GCarro GCarro;

struct _GCarroClass
{
 GObjectClass parent_class;
};
struct _GCarro
{
```

```
 GObject parent_instance;

 gboolean ligado;
 guint velocidade;
};

GType g_carro_get_type (void) G_GNUC_CONST;
void g_carro_ligar (GCarro *self);
void g_carro_desligar (GCarro *self);
guint g_carro_acelerar (GCarro *self, guint velocidade);
guint g_carro_reduzir (GCarro *self, guint velocidade);
G_END_DECLS

#endif /* _G_CARRO_H_ */
```

Arquivo: *g-carro.c*

```
#include "g-carro.h"

enum
{
 PROP_0,

 PROP_VELOCIDADE,
 PROP_LIGADO
};

static void
g_carro_morrer (GCarro *self)
{
      /* método privado, pois possui o modificador static */
 g_print ("O carro morreu!\n");

 /* Mudando atributos da classe */
 self->ligado = FALSE;
 self->velocidade = 0;
}
```

```
G_DEFINE_TYPE (GCarro, g_carro, G_TYPE_OBJECT);

static void
g_carro_init (GCarro *self)
{
 /* Construtor da classe */
 self->ligado = FALSE;
 self->velocidade = 0;
}

static void
g_carro_finalize (GObject *object)
{
 /* Destrutor */

 g_print ("Objeto GCarro eliminado!\n");

 /* Executando o destrutor da classe mãe */
 G_OBJECT_CLASS (g_carro_parent_class)->finalize (object);
}

static void
g_carro_set_property (GObject *object,
                      guint prop_id,
                      const GValue *value,
                      GParamSpec *pspec)
{
 g_return_if_fail (G_IS_CARRO (object));

 switch (prop_id)
 {
 case PROP_VELOCIDADE:
  /* definindo o valor da velocidade do carro. */
  G_CARRO(object)->velocidade = g_value_get_uint (value);

  /* se a velocidade for zero, o carro deve morrer */
  if (G_CARRO(object)->velocidade == 0)
    g_carro_morrer (G_CARRO(object));
```

```
 break;
 case PROP_LIGADO:
  /* definindo o valor do estado do carro */
  G_CARRO(object)->ligado = g_value_get_boolean (value);
  break;
 default:
  G_OBJECT_WARN_INVALID_PROPERTY_ID (object, prop_id, pspec);
  break;
 }
}

static void
g_carro_get_property (GObject *object,
                      guint prop_id,
                      GValue *value,
                      GParamSpec *pspec)
{
 g_return_if_fail (G_IS_CARRO (object));

 switch (prop_id)
 {
 case PROP_VELOCIDADE:
  /* retornando a velocidade do carro */
  g_value_set_uint (value, G_CARRO(object)->velocidade);
  break;
 case PROP_LIGADO:
  /* retornando o estado do carro */
  g_value_set_boolean (value, G_CARRO(object)->ligado);
  break;
 default:
  G_OBJECT_WARN_INVALID_PROPERTY_ID (object, prop_id, pspec);
  break;
 }
}

static void
g_carro_class_init (GCarroClass *klass)
{
```

```
 GObjectClass* object_class = G_OBJECT_CLASS (klass);
 GObjectClass* parent_class = G_OBJECT_CLASS (klass);

 object_class->finalize = g_carro_finalize;
 object_class->set_property = g_carro_set_property;
 object_class->get_property = g_carro_get_property;

 /* Criando estrutura de properties */
 g_object_class_install_property (object_class,
                         PROP_VELOCIDADE,
                         g_param_spec_uint ("velocidade",
                                 "Velocidade",
                                 "Velocidade atual",
         0,
         G_MAXUINT,
                                 G_TYPE_UINT,
                                 G_PARAM_READABLE | G_PARAM_WRITABLE));

 g_object_class_install_property (object_class,
                         PROP_LIGADO,
                         g_param_spec_boolean ("ligado",
                                   "Ligado",
                                   "Se o estado atual do carro é ligado",
                                   FALSE,
                                   G_PARAM_READABLE | G_PARAM_WRITABLE));
}


void
g_carro_ligar (GCarro *self)
{
 g_object_set (self, "ligado", TRUE, NULL);
 g_print ("Carro ligado!\n");
}

void
g_carro_desligar (GCarro *self)
{
 g_object_set (self, "ligado", FALSE, NULL);
```

```
 g_object_set (self, "velocidade", TRUE, NULL);
 g_print ("Carro deligado!\n");
}

guint
g_carro_acelerar (GCarro *self,
                  guint velocidade)
{
 guint total;
 if (self->ligado == FALSE){
  g_print ("Ligue o carro antes de acelerar !\n");
  return 0;
 }


 /* obtendo valor da property de velocidade */
 g_object_get (self, "velocidade", &total, NULL);

 /* acrescentando velocidade passada pelo parametro */
 total += velocidade;

 /* definindo novo valor da property */
 g_object_set (self, "velocidade", total, NULL);

 g_print ("Carro andando a %d KM/h\n", total);
}

guint
g_carro_reduzir (GCarro *self,
                 guint velocidade)
{
 guint total;
 if (self->ligado == FALSE){
  g_print ("Ligue o carro antes de reduzir !\n");
  return 0;
 }
 /* obtendo valor da property de velocidade */
 g_object_get (self, "velocidade", &total, NULL);
```

```
 /* subtraindo velocidade passada pelo parametro */
 if (total - velocidade < 0)
  total = 0;
 else
  total -= velocidade;

 /* definindo novo valor da property */
 g_object_set (self, "velocidade", total, NULL);

 /* se a velocidade for igual a zero, o carro vai morrer,
  é preciso checar se está ligado antes de exibir sua nova velocidade */
 if (self->ligado == TRUE){
  g_print ("Carro andando a %d KM/h\n", total);
 }
}
```

ANEXO B - Definição da classe GCarro em Python

```
class GCarro(object):
    def __init__(self):
        # construtor da classe
        self.__velocidade = 0
        self.__ligado = False

    def __morrer(self):
        # método privado
        print "O carro morreu!"

        # mudando atributos da classe
        self.__velocidade = 0
        self.__ligado = False

    def __del__(self):
        # destrutor da classe
        print "Objeto GCarro eliminado!"

    # definindo a property "velocidade"
    def __set_velocidade(self, valor):
```

```
        self.__velocidade = valor
        if self.__velocidade == 0:
            self.__morrer()

    def __get_velocidade(self):
        return self.__velocidade

    velocidade = property(fget=__get_velocidade, fset=__set_velocidade)

    # definindo a property "ligado"
    def __set_ligado(self, valor):
        self.__ligado = valor

    def __get_ligado(self):
        return self.__ligado

    ligado = property(fget=__get_ligado, fset=__set_ligado)

    # métodos públicos
    def ligar(self):
        self.__ligado = True
        print "Carro ligado!"

    def desligar(self):
        self.__ligado = False
        self.__velocidade = 0
        print "Carro desligado!"

    def acelerar(self, velocidade):
        if self.ligado == False:
            print "Ligue o carro antes de acelerar"
            return 0

        total = self.velocidade
        total += velocidade

        self.velocidade = total

        print "Carro andando a %d KM/h" % total
```

```
def reduzir(self, velocidade):
    if self.ligado == False:
        print "Ligue o carro antes de reduzir"
        return 0

    total = self.velocidade
    if (total - velocidade < 0):
        total = 0
    else:
        total -= velocidade

    self.velocidade = total

    # se a velocidade for igual a zero, o carro vai morrer,
    # é preciso checar se está ligado antes de exibir sua
    # velocidade
    if self.ligado == True:
        print "Carro andando a %d KM/h" % total
```

Sinceros agradecimentos aos meus pais e irmãos, pelo apoio incessante, a todos os amigos pelo incentivo, especialmente aos amigos Gustavo Noronha e Lincoln de Sousa por me mostrarem "a magia do natal"*, ao orientador por apoiar o tema e nortear o trabalho, a todos os professores, e especialmente ao professor Antônio Ricardo Leocádio, cujos ensinamentos sobre software livre e didática brilhante foram essenciais ajudando a me tornar um desenvolvedor e contribuidor de projetos de software livre, culminando, inclusive, no conhecimento da tecnologia abordada no trabalho.*