Um idioma em C++ para a automatização da gestão de recursos

Luis Valente
Aura Conci

Universidade Federal Fluminense – Instituto de Computação
Rua Passo da Pátria, 156 Bloco E Sala 452 - 24.210-240 – Niterói – RJ

{lvalente,aconci}@ic.uff.br

Abstract. *Resource management is a common task held in any application. However, developers must manage this task directly in many cases. Therefore, as application complexity increases, resource management may become too complicated to be handled manually, and prone to errors. This paper presents a design pattern which exploits properties of the C++ programming languagem in order to automate resource management. This pattern was applied in the design of a game developement tool, the Guff framework.*

Resumo. *Gestão de recursos é uma tarefa comum que é realizada em qualquer aplicação. Entretanto, em muitos casos essa tarefa deve ser realizada manualmente pelo desenvolvedor. Com o aumento da complexidade das aplicações, esse tipo de tarefa pode tornar-se muito complicado e sujeito a erros. Este artigo apresenta um idioma que explora certas características da linguagem de programação C++, para automatizar a gestão de recursos. Este idioma foi aplicado no projeto de uma ferramenta destinada ao desenvolvimento de jogos para computador, o framework Guff.*

1. Nome

Gestão automatizada de recursos.

2. Intenção

Possibilitar que a gestão de recursos seja automatizada com o auxílio do compilador, para que o desenvolvedor não precise realizar a gestão manualmente. Esse idioma é específico para C++, pois explora certas propriedades encontradas nessa linguagem.

3. Motivação

A gestão de recursos é uma das tarefas mais comuns que devem ser executadas em uma aplicação. Um recurso pode ser definido como algo que deve ser requisitado ao sistema operacional, quando necessário, e que deve ser devolvido ao sistema após o uso. Essa definição é genérica o bastante para qualificar exemplos como memória, arquivos, *sockets* e travas para *mutexes*, entre outros.

O principal problema em relação a recursos é que o programador deve explicitamente devolvê-los ao sistema operacional quando não são mais necessários. Quando isto não é feito, uma série de problemas podem se manifestar, como perda de recursos (*resource leaks*), *deadlocks*, *bugs* "aleatórios" (um defeito que é difícil de se reproduzir), acessos inválidos à memória e uso excessivo de recursos (por exemplo, um programa utiliza mais memória do que deveria, porque não devolve blocos de memória alocados previamente), entre outros, dependendo do tipo do recurso. As linguagens de programação, em geral, não possuem funcionalidade padrão para tratar desse problema. Como exemplo, têm-se o trecho de código em C++:

```
void foo () {

  int * p = new int;
        p = new int;
}
```

No trecho de código, o primeiro bloco de memória é perdido na segunda atribuição. Quando o fluxo de execução do programa sai do escopo da função, o bloco de memória requisitado na segunda alocação também é perdido. Dessa forma, esse exemplo demonstra um erro conhecido como vazamento de memória. Isso ocorre porque o processo de devolução de recursos ao sistema não é automatizado.

Para resolver esse tipo de problema, este trabalho propõe um idioma que pode ser usado para minimizar erros relacionados a recursos e melhorar a qualidade do *software* produzido. Com o propósito de validar esse idioma, são discutidos exemplos que fazem parte da implementação do *framework* Guff [1].

O Guff é uma ferramenta destinada ao desenvolvimento de jogos para computador. Essa ferramenta é implementada em C++ e possui duas partes principais: a camada de aplicação e um *toolkit*. A camada de aplicação determina as interfaces e a arquitetura dos programas que utilizam essa ferramenta. O *toolkit* representa um conjunto de classes auxiliares que podem ser utilizadas para desenvolver os programas.

As forças que moldam este idioma são as seguintes:

- A gestão incorreta de recursos pode ocasionar erros diversos, como perdas de recursos, uso excessivo de recursos, *bugs* "aleatórios" e *deadlocks*, entre outros, dependendo do tipo de recurso;

- Um recurso pode ser referenciado por diferentes objetos, acidentalmente. Isso pode gerar problemas caso um deles devolva o recurso e um outro objeto tente usá-lo posteriomente;

- O compartilhamento de recursos pode ser uma característica necessária em uma aplicação. Nesse caso, é preciso estabelecer critérios que indiquem como o compartilhamento será controlado e quando o recurso será devolvido ao sistema.

- Um dos conceitos principais em gestão automatizada de recursos é a definição de uma política de propriedade sobre recursos. Em termos gerais, o proprietário de um recurso é aquele que é responsável por sua devolução ao sistema. Esse conceito é muito importante, pois é preciso estabelecer claramente critérios para a determinação da propriedade sobre o recurso, para que a gestão de recursos possa ser realizada corretamente.

- Um programa pode gerar diversas exceções durante a sua execução. Quando isso ocorre, o fluxo de execução deixa o bloco de código no trecho onde a exceção foi gerada. Dessa forma, podem ocorrer perdas de recursos.

A linguagem de programação C++ possui certas propriedades que podem ser usadas para automatizar a gestão de recursos, como escopo em blocos de código, construtores e destrutores de classes.

Os construtores e destrutores são métodos especiais das classes cuja execução é garantida pelo projeto da linguagem C++ [Stroustrup 1997]. O construtor é executado na criação de uma instância de uma classe e o destrutor é executado quando a instância é destruída.

A automatização da gestão de recursos pode ser realizada ao combinar-se os conceitos de escopo, construtores e destrutores de classes, através da aplicação da seguinte regra: alocar recursos nos construtores e devolver os recursos nos destrutores correspondentes. Essa regra é definida em [2], como Regra Primeira de Aquisição e citada como "resource acquisition is initialization" em [3]. Dessa forma, é possível utilizar o compilador para automatizar o processo de criação e destruição de instâncias de classes.

Instâncias de classes que implementam esse paradigma também são conhecidas como objetos automáticos.

4. Aplicabilidade

Este idioma pode ser utilizado quando deseja-se:

- Evitar perdas de recursos em situações que possam potencialmente gerar exceções;
- Evitar perdas de recursos quando a aplicação torna-se complexa demais para ser gerida manualmente;
- Evitar que um recurso seja referenciado acidentalmente por mais de uma classe;
- Promover o compartilhamento de recursos;
- Obter flexibilidade para implementar outras semânticas para gestão de recursos.

5. Estrutura

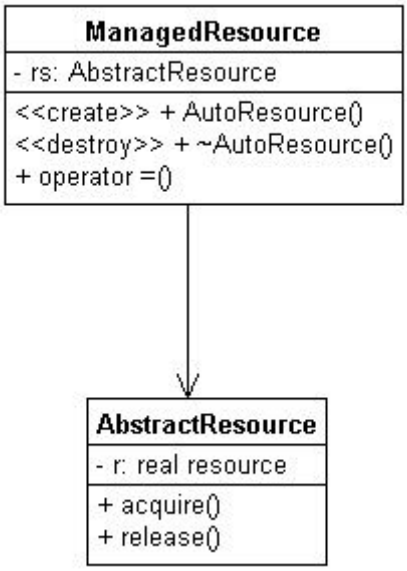

Figura 1. Estrutura do idioma

6. Participantes

A classe `AbstractResource` é responsável por encapsular os procedimentos de requisição e devolução do recurso ao sistema operacional, através dos métodos `acquire` e `release`. O campo `r` representa o recurso real (um arquivo, por exemplo).

A classe `ManagedResource` é responsável por definir e aplicar uma política para a gestão automatizada do recurso. Essa política é determinada pelo construtor, destrutor e pelo operador de atribuição. O construtor cria uma instância do recurso a ser gerido. O destrutor decide o que fazer com o recurso gerido quando uma instância de `ManagedResource` está para ser destruída. O operador de atribuição é responsável por aplicar a política de propriedade sobre o recurso (essa tarefa também é de responsabilidade do construtor de cópias).

7. Colaborações

O cliente utiliza `ManagedResource` para obter acesso a `AbstractResource`.

8. Consequências

Esse idioma também funciona quando exceções são geradas em um programa. Isso se deve ao fato de que os destrutores dos objetos locais (automáticos) também são executados depois que o fluxo de execução deixa um bloco de código, em virtude de uma exceção. Sendo assim, é possível realizar corretamente a gestão de recursos na presença de erros inesperados.

Entretanto, existem duas situações (que representam detalhes de implementação do C++) onde esse paradigma não funcionará. Essas situações correspondem ao uso das funções `exit` e `abort`. Um programa em C++ pode terminar, normalmente, das seguintes maneiras [3]:

- Quando a execução da função principal do programa (`main`) termina;
- Quando a função `exit` é chamada;
- Quando a função `abort` é chamada;
- Quando uma exceção que tenha sido gerada pela aplicação não é tratada.

A chamada às funções `exit` ou `abort` faz com que o fluxo de execução deixe a função no ponto onde a chamada ocorreu. Isso significa que os destrutores de objetos locais da função (e de outras funções que a chamaram) não serão executados.

A diferença entre `exit` e `abort` é que esta última causa o término imediato do programa, enquanto a primeira permite a realização de outras tarefas.

Como a gestão automatizada de recursos utiliza a premissa de que os destrutores serão usados para devolver os recursos, caso essas funções sejam usadas os destrutores de objetos locais não serão executados e, portanto, o idioma irá falhar.

9. Implementação e exemplos

Em geral, uma instância de uma classe pode ser criada em um programa desenvolvido em C++ das seguintes formas [3]:

- Um objeto é declarado como uma variável local em um bloco de código. Nesse caso, a instância será criada quando o fluxo de execução passar pelo ponto da declaração. A instância será destruída quando o fluxo de execução deixar o bloco de código;
- A instância é criada na memória através do operator **`new`**. A destruição da instância poderá ser realizada através do uso do operador **`delete`**.
- Um objeto B é declarado como membro de outro objeto A. Quando o objeto A for criado, seu construtor irá criar uma instância de B. Essa instância será destruída pelo destrutor do objeto A, quando este for destruído.

- Como um elemento de um *array*. Quando o *array* for criado, os construtores padrões de cada elemento serão executados, criando-se as instâncias. Quando o *array* for destruído, os destrutores de cada elemento serão executados, destruindo-se as instâncias.

- Um objeto é declarado como estático em uma função. Quando a função for executada pela primeira vez, o objeto será criado. Esse objeto será destruído ao término do programa.

- Um objeto é declarado como uma variável global, constante global ou membro estático de uma classe. A instância do objeto será criada no início da execução do programa e será destruída ao término da aplicação.

- Uma instância é criada como um objeto temporário, o que pode ocorrer, por exemplo, durante a avaliação de uma expressão. A instância será destruída quando a avaliação da expressão terminar.

Com exceção da segundo caso de formas de criação de instâncias, o escopo do objeto é bem definido (ou seja, sabe-se onde o construtor e o destrutor deverão ser executados). Para automatizar a gestão de recursos para o segundo caso, pode ser utilizada uma combinação com as outras abordagens.

Uma possível declaração de classe que aplica a gestão automática de um recurso hipotético é descrita por este trecho de código:

```
class ManagedResource {
  public:

     ManagedResource () { acquire (); }
    ~ManagedResource () { release (); }

  private:
    AbstractResource mRes;
};
```

Essa abordagem, entretanto, é ligeiramente rígida. Uma simples variação pode ser usada para se obter uma maior flexibilidade de uso:

```
class ManagedResource {
  public:

   ManagedResource (AbstractResource rs)
    : mRes (rs)
   {
     mRes.acquire ();
   }

   void reset (AbstractResource rs) {
     mRes.release (); mRes = rs;
   }
   ... // sem alterações

};
```

Esse exemplo não viola as regras, desde que o método `acquire` de `rs` não seja usado antes do momento em que o objeto é passado para `ManagedResource`.

Para que a gestão automatizada de recursos funcione corretamente, é preciso estabelecer claramente o proprietário do recurso. Um objeto é considerado propretário de um recurso se o objeto for o responsável por devolvê-lo ao sistema.

Por exemplo, este trecho de código demonstra uma situação problemática em relação a essa questão:

```
ManagedResource r1;
ManagedResource r2;
r2 = r1;
```

Nesse exemplo, são utilizados dois objetos que armazenam recursos distintos. Entretanto, quando a operação de atribuição é invocada, não é possível distinguir claramente o proprietário sobre o recurso. A operação de atribuição original possui uma semântica de cópia, ou seja, quando ocorre uma operação desse tipo, o objeto alvo recebe uma cópia do objeto fonte, *bit* a *bit* (*bitwise*). Dessa forma, a definição sobre o proprietário do recurso passa a ser ambígua. O proprietário poderia ser `r1`, `r2` ou ambos. Por exemplo, se o recurso fosse memória, os dois ponteiros referenciariam o mesmo endereço. Para resolver esse problema, é necessário estabelecer critérios para a determinação dos proprietários sobre os recursos.

A alternativa nesse caso é modificar a semântica da operação de atribuição, tornando-a responsável por aplicar a política que rege a propriedade sobre o recurso. Por exemplo, uma política que determine que deva existir apenas um propretário para um dado recurso, poderia ser implementada desta forma:

```
class ManagedResource

{
    void operator = (ManagedResource & obj) {
       mRes.release ();
       mRes = obj.transferResource ();
     }

      ManagedResource (ManagedResource & obj)
        : mRes (obj.transferResource () )
        {}

  private:
    AbstractResource transferResource () {

      AbstractResource tmp = mRes;
      mRes.null ();
      return tmp;
    }

    AbstractResource mRes;

};
```

Nesse caso, a operação de atribuição passa a ter uma semântica de transferência de recursos. No exemplo, `mRes.null` é uma operação que atribui ao recurso um valor neutro. Caso o recurso gerido fosse memória, esse valor seria 0 (`NULL`).

9.1. Estudo de caso: contadores de referências

Em várias aplicações, pode ser desejado compartilhar recursos. No Guff, por exemplo, todos os recursos podem ser compartilhados. Dessa forma, a política que determina a propriedade sobre os recursos deve considerar a propriedade coletiva (ao contrário do que acontece no exemplo anterior).

Uma solução é o uso de contadores de referências. Nesse caso, vários objetos podem referenciar um mesmo recurso ao mesmo tempo, e o contador é utilizado para guardar o número de referências para o recurso existentes. Quando a última referência for removida, o recurso deverá ser devolvido ao sistema.

Uma possível implementação é descrita neste exemplo:

```
class RefCounter {
  public:
    RefCounter ()  : mCounter (1) {}
    void increase () { ++mCounter; }
    void decrease () { --mCounter; }
    int  getCount () { return mCounter; }

  private:
    int mCounter;
};
```

A classe `RefCounter` implementa um contador de referências e a classe a seguir, `SharedRes`, representa uma referência compartilhada para um recurso.

```
class SharedRes {
 public:

  SharedRes (AbstractResource rs)
   : mCounter (new RefCounter), mRes (rs) {
     mRes.acquire ();
   }

  ~SharedRes () {
    decRef ();
  }

  SharedRes (SharedRes & obj) {
    addRef (obj);
  }

  void operator = (SharedRes & obj) {
    if (mRes != obj.mRes) {
      decRef ();
      addRef (obj);
    }
  }

 private:
```

```
  void addRef (SharedRes & obj) {
    obj.mCounter->increase ();

    mCounter = obj.mCounter;
    mRes     = obj.mRes;
  }

  void decRef () {
    mCounter->decrease ();

    if (mCounter->getCount () == 0) {
      mRes.release ();
      delete mCounter;

    }
  }

 private:
   AbstractResource mRes;
   RefCounter *     mCounter;
};
```

Nesse exemplo, o operador de atribuição possui uma semântica de compartilhamento de recursos, sendo responsável por atualizar o contador de referências e devolver o recurso ao sistema, se for o caso.

A operação de atribuição e o construtor de cópias são responsáveis por atualizar automaticamente o contador de referências e devolver o recurso ao sistema, se for apropriado. Dessa forma, o modo de uso de uma classe desse tipo é natural para o usuário, como neste exemplo:

```
SharedRes r1 (..);
SharedRes r2 (..);
r2 = r1;
```

Nesse exemplo, são alocados dois recursos distintos. Na operação de atribuição, o recurso inicialmente gerido por `r2` é devolvido ao sistema, e o recurso gerido inicialmente por `r1` passa a ser referenciado por ambos os objetos.

É importante notar que a implementação sugerida não leva em consideração aspectos relacionados a programação concorrente.

9.2. Estudo de caso: gestão de texturas

Em visualização tridimensional, uma textura representa uma imagem que pode ser aplicada em malhas de polígonos. As texturas são recursos limitados pela memória disponível nas placas gráficas. Uma placa gráfica típica atual possui entre 64 MB e 256 MB de memória. Dessa forma, a memória de vídeo é um recurso precioso que precisa ser gerido cuidadosamente, para que os jogos possam ser executados de maneira eficiente.

Para resolver esse problema, o Guff oferece diversas classes relacionadas a texturas e um gerente de texturas. Essas classes são retratadas pela figura 2, e foram projetadas com os seguintes requisitos:

- Executar o processo de leitura dos dados das imagens em disco, transformação desses dados em texturas e posterior devolução dos recursos associados ao sistema;

- Não permitir que as texturas sejam carregadas mais de uma vez, evitando o desperdício de recursos.

Para cumprir esses requisitos, essas classes implementam o paradigma de gestão automatizada de recursos.

O OpenGL [4] é a biblioteca de visualização tridimensional utilizada pelo Guff para implementar o seu módulo de visualização. No OpenGL, existe uma abstração denominada objeto de textura, que representa uma série de propriedades relacionadas às texturas, como os dados da imagens e filtros a serem usados nas operações de mapeamento de texturas. De maneira análoga a outros tipos de recursos, os objetos de textura devem ser requisitados ao OpenGL antes de serem usados, e devolvidos mais tarde.

A classe `TextureObject2` é a responsável pela gestão automática dos objetos de textura do OpenGL. No construtor dessa classe, um objeto de textura é requisitado ao OpenGL, e no destrutor esse objeto é devolvido ao sistema.

A classe `TextureParameters` representa um conjunto de diversos parâmetros relacionados a texturas, como os filtros a serem usados nas operações de mapeamento de textura.

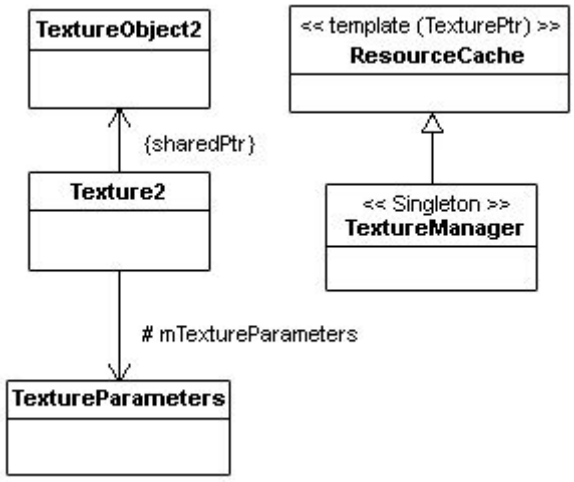

Figura 2. Diagrama de classes para texturas

A classe `Texture2` representa a textura e suas propriedades, como largura, altura e quantidade de *bits* por *pixel*, entre outras. A propriedade mais interessante é uma referência para um objeto de tipo `TextureObject2`. Essa referência é uma instância de uma classe que implementa contadores de referências, como exemplificado no estudo de caso anterior. Dessa forma, essa classe é, naturalmente, um objeto automático. Como exemplo, têm-se este trecho de código:

```
void foo ()
{

 Texture2 t1;
 Texture2 t2 ("abc.jpg");
```

```
 // t1 carrega a textura padrão do Guff, que representa uma
 // imagem xadrez (esse é o comportamento do construtor padrão).
 // t2 carrega a imagem abc.jpg


 t2 = t1;

 // t2 devolve os recursos associados com abc.jpg e
 // adiciona uma referência para a textura padrão que
 // foi carregada por t1.

 t1.load ("xyz.tga");

 // t1 carrega a imagem xyz.tga e t2 ainda referencia
 // a imagem padrão

}
 // os recursos associados com xyz.tga e com a textura padrão
 // são devolvidos automaticamente ao sistema quando a função
 // é encerrada
```

A classe `TextureManager` representa um *cache* para texturas. Sua principal responsabilidade é gerir a leitura de arquivos de imagem de modo que uma determinada textura não seja carregada mais de uma vez. Essa classe é implementada segundo o padrão de projeto *Singleton* [5]. Isso implica que existe somente uma única instância do gerente de texturas no sistema.

A operação de criação de texturas através do *cache* é bastante simples: inicialmente, o *cache* verifica se a textura requisitada está presente em seu repositório. Caso exista, é retornada uma referência para a textura armazenada no repositório (sob a forma de um objeto automático com contador de referências). Caso não exista, o gerente de texturas cria uma nova textura com os dados da imagem requisitada, armazena esse novo objeto no repositório e retorna uma referência para a textura.

Uma consequência interessante da aplicação da gestão automatizada de recursos nesse caso, é que o gerente de texturas não precisa devolver os recursos alocados para as texturas ao sistema explicitamente, pois as classes `Texture2` e `TextureObject2` farão isso automaticamente.

9.3. Estudo de caso: gestão de bibliotecas

O Guff utiliza bibliotecas externas, sempre que possível, para implementar vários de seus serviços. O objetivo dessa abordagem é evitar uma prática conhecida como "not built here" [6]. Esse tipo de prática, comum em desenvolvimento de jogos, indica a decisão de se implementar novamente todas as funcionalidades necessárias para a criação de um jogo, a cada projeto, em vez de reutilizar funcionalidades já existentes (na forma de bibliotecas e outros tipos de ferramentas).

Algumas das bibliotecas usadas pelo Guff precisam ser inicializadas antes de serem utilizadas e finalizadas ao término da aplicação.

Esse problema representa um caso que pode ser resolvido através da aplicação do paradigma de gestão automática de recursos. Nesse exemplo, o recurso a ser gerido é o estado de uso das bibliotecas (inicializado ou finalizado). Uma possível implementação para a gestão desse recurso é descrita aqui:

```
class Initializer
{
 public:

  Initializer () {
    if (library not initialized) {
      initializeLibrary ();
    }

    if (initialization failed)
      // raise an exception


    useCounter++;
  }

  Initializer::~Initializer () {
    useCounter--;

    if (useCounter == 0) {
      finalizeLibrary ();
    }

  }

 private:
   static int useCounter;

};
```

Existem duas vantagens interessantes ao se usar essa abordagem. A primeira é a possibilidade de inicializar e finalizar as bibliotecas corretamente, mesmo que ocorram exceções (erros fatais) durante o ciclo de vida da aplicação. A segunda vantagem é a possibilidade de lidar com bibliotecas que tenham dependências entre si, de maneira simples. Por exemplo, se uma biblioteca A necessita que uma biblioteca B esteja inicializada para que possa ser usada, basta encapsulá-las como objetos automáticos. Dessa forma, a finalização (que se dá na ordem contrária, primeiro B e depois A) também será realizada na ordem correta.

9.4. Estudo de casos: acesso a regiões críticas

Em programação concorrente, vários processos competem por acesso a recursos limitados do sistema. Quando não é possível satisfazer uma requisição, o processo é bloqueado até que seja possível satisfazê-la. O *deadlock* pode ocorrer quando processos de um determinado grupo permanecem bloqueados indefinidamente, porque estão à espera de recursos retidos por outros processos do grupo, entre si. Existem quatro pré-condições para a ocorrência de *deadlocks* [7]:

- Exclusão mútua: existe um determinado tipo de recurso só pode estar alocado a um único processo em um determinado instante;
- Aquisição gradativa de recurso: existe um processo que retém um recurso e está à espera de um outro;
- Não-preempção: um recurso só pode ser liberado explicitamente por um processo;

- Espera circular: dois ou mais processos formam uma fila circular, onde cada processo espera por um recurso que está sendo retido pelo próximo processo na fila.

A gestão automatizada de recursos pode ser usada para controlar o acesso a regiões críticas, que estão relacionadas com a primeira pré-condição. Uma região crítica representa uma seção de código onde somente um processo pode executar por vez. O objetivo é evitar a retenção acidental dessas regiões, o que poderia provocar um *deadlock*. Como exemplo, têm-se a classe `Mutex`:

```
class Mutex
{
 public:
  void acquire () { ... }
  void release () { ... }

 private:

  // para evitar cópias acidentais
  Mutex (const Mutex &);
  void operator = (const Mutex &);
};
```

Essa classe encapsula o protocolo de entrada e saída de regiões críticas. O controle de acesso à região pode ser implementado através de um guarda, como a classe `Lock`:

```
class Lock
{
  public:
    Lock (Mutex & m)
     : _mutex (m)
     { _mutex.acquire (); }

    ~Lock ()
     { _mutex.release (); }

  private:
    Mutex & _mutex;

  private:

    // para evitar cópias acidentais
    Lock (const Lock &);
    void operator = (const Lock &);
};
```

O objetivo da classe `Lock` é evitar a retenção acidental da região crítica, mesmo que aplicação gere exceções. O modo de uso dessa classe é bastante simples:

```
  // supõe que mutex tenha sido declarado corretamente em alguma
  // parte do sistema
void foo ()
{
  Lock (mutex);

  // processamento ...
```

```
}
```

9.5. Outros exemplos

A denominação deste idioma no contexto de gestão de memória é popularmente conhecida como *smart pointers*. Esse tipo de especialização pode ser usado para compartilhamento de recursos, ou não. A classe `auto_ptr` [8] permite que apenas um objeto referencie um determinado conteúdo de memória, assim como é feito em [9]. As classes da da biblioteca boost [10] que simulam ponteiros, permitem gestão simples ou compartilhada de memória (com contadores de referências). A gestão de memória com contadores de referência é apresentada como o idioma *Counted body* em [11].

A gestão automatizada de recursos é referida sob o nome de "resource acquisition is initialization" em [3]. Outro trabalhos que apresentam essa técnica são [12, 2]. Entretanto, nenhum desses trabalhos descreve a gestão automatizada de recursos sob a forma de um idioma.

10. Padrões relacionados

- *Smart pointers;*
- *Counted body*;
- *Resource acquisition is initialization*.

11. Agradecimentos

Os autores fazem um agradecimento à equipe de *shepherding* do SugarloafPLop 2005, em especial a Giulano Mega, por sua valiosa orientação para atingir a versão atual deste trabalho. Os autores agradecem o apoio financeiro da CAPES. A segunda autora agradece também ao CNPq pelo contrato de pesquisa PQ No. 302436/2004-9.

12. Referências

[1] Valente, L., Guff: *Um Sistema para Desenvolvimento de Jogos*, Dissertação de Mestrado, Universidade Federal Fluminense, 2005.

[2] Milewsky, B., *C++ in Action: Industrial Strength Programming Techniques, Web Edition*, http://www.relisoft.com/book (22/03/05).

[3] Stroustrup, B., *C++ Programming Language, 3rd Edition*, Addison-Wesley, 1997.

[4] Segal, M., e Akeley, K, *The OpenGL Graphics System: A Specification - Version 2.0*, Documentação, 2004.

[5] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[6] Rollings, A., e D. Morris, *Game Architecture and Design: A New Edition*, New Riders, 2003.

[7] Silberschatz, A., and P. Galvin, Operating Systems Concepts, 5th Edition, Addison-Wesley, Reading, 1998.

[8] SGI, *Standard Template Library Programmer's Guide*, Disponível em http://www.sgi.com/tech/stl/ (23/03/05).

[9] Alexandrescu, A., *Modern C++ Design – Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001.

[10] Boost.org, *Boost C++ Libraries*, Disponível em http://www.boost.org (23/03/05).

[11] Coplien, J., "C++ Idioms Patterns". In Pattern Languages of Program Design 4, Editado por Brian Foote, Neil Harrison e Hans Rohnert, Addison-Wesley, 2000.

[12] Meyer, S., *More Effective C++*, Addison-Wesley, 1996.