# 3
# Principais Usos de Referências Fracas e Finalizadores

Finalizadores e referências fracas recebem amplo suporte de algumas das principais linguagens de programação, entre elas C# (CLS07, Richter02), Java, Lua, Python, Perl (Schwartz05) e Haskell. No entanto, muitas vezes não precisamos de finalizadores quando a linguagem oferece suporte a referências fracas. Partindo desses fatos, decidimos investigar quais os usos típicos desses mecanismos através de uma pesquisa informal. Utilizamos como fonte de pesquisa a literatura, listas de discussões e dois mecanismos de busca em código na internet, Krugle (Krugle07) e Google Code Search (GCS07). Além disso, enviamos um questionário por e-mail a programadores tanto da área acadêmica quanto da indústria. Mais especificamente, enviamos o questionário para a lista de discussão da Linguagem Lua (LML07), que possui mais de 1.200 assinantes, e para a lista de discussão do laboratório Tecgraf da PUC-Rio (TCG07). Enviamos também mensagens individuais para 18 programadores da área acadêmica e 17 programadores da indústria. Uma parte do resultado de nossa pesquisa encontra-se na Tabela 3.1 e a outra parte refere-se aos usos encontrados para finalizadores e referências fracas que serão discutidos nas seções seguintes.

Ao analisar o resultado mostrado na Tabela 3.1, foi possível observar rapidamente que o mecanismo de referências fracas é pouco usado e pouco conhecido pela maioria dos programadores. Apenas 11% dos programadores que responderam a pesquisa já usaram referências fracas e menos da metade conhece esse mecanismo. Os número são mais generosos com relação a final-

| | **Lua** | **Tecgraf** | **Academia** | **Indústria** |
|---|---|---|---|---|
| Total de programadores | $> 1.200$ | 63 | 18 | 17 |
| Respostas obtidas | 4 | 5 | 18 | 17 |
| Conhecem finalizadores | 4 | 5 | 18 | 17 |
| Já usaram finalizadores | 2 | 1 | 18 | 17 |
| Conhecem referências fracas | 4 | 5 | 4 | 5 |
| Já usaram referências fracas | 2 | 3 | 0 | 0 |

Tabela 3.1: Resultado da pesquisa informal.

izadores: 86% já usaram e todos conhecem esse mecanismo.

Com o objetivo de mostrar o alcance do mecanismo de referências fracas, realizamos algumas consultas no Google Code Search e no Krugle. Procuramos por programas Java que contivessem a palavra "String" a fim ter uma idéia do tamanho da base. Foram encontrados 1.430.000 resultados na primeira ferramenta e 1.805.649 na segunda. Ao buscar por "WeakReference" obtivemos 4.000 resultados na primeira e 2.727 na segunda, ou seja, apenas 0,28% e 0,15% da base respectivamente. Podemos então observar quão pouco esse mecanismo é utilizado. Já no caso de finalizadores, efetuamos uma busca por "finalize" em cada ferramenta, pois para implementar um finalizador em Java é necessário sobrescrever o método `finalize` da classe `java.land.Object`, da qual todas as classes são derivadas. Note que, como esse identificador não é uma palavra reservada nem um tipo específico da linguagem, ele pode ser usado em outros contextos que não dizem respeito a finalização de objetos. Devido a isso, retiramos uma amostra de 5% dos resultados para verificar onde a palavra "finalize" era usada. Pudemos então observar que em ambas as amostras, "finalize" se referia unicamente à implementação do método `finalize`. Voltando então à busca, obtivemos 14.500 resultados no Google Code Search e 13.328 no Krugle. Isso nos mostra que finalizadores têm um alcance quase quatro vezes maior que referências fracas de acordo com a primeira ferramenta e quase cinco vezes maior de acordo com a segunda.

Na próxima seção descrevemos os principais usos de referência fracas encontrados. A seguir, na Seção 3.2, além de descrever cada um dos usos típicos encontrados para finalizadores, mostramos como eles podem ser implementados através de referências fracas. Após catalogar as respostas dos programadores ao questionário enviado e realizar uma pesquisa bibliográfica em busca de usos de finalizadores e referências fracas, observamos que todos os usos encontrados em nossa pesquisa estavam descritos brevemente no trabalho de Leal (Leal05). No nosso trabalho, detalhamos mais cada um dos usos e as respectivas implementações. Além disso, apresentamos novos exemplos e casos especiais.

## 3.1 Referências Fracas

Nesta seção, descrevemos os usos encontrados para o mecanismo de referências fracas, são eles:

- **Coleta de referências cíclicas** – Encontrado na literatura, nos trabalhos de Brownbridge (Brownbridge85) e Leal (Leal05).
- **Cache** – Dentre os 5 programadores que disseram já ter usado referências fracas, dois o fizeram para implementar uma cache de objetos. Também

encontramos uma descrição desse uso no trabalho de Leal (Leal05).

- **Tabelas de propriedades** – Encontramos descrições desse uso nos trabalhos de Hayes (Hayes97) e Leal (Leal05).
- **Conjuntos fracos** – Dois programadores, dentre os 5 que disseram já ter usado referências fracas, disseram ter usado o mecanismo para implementar conjuntos fracos. O trabalho de Leal (Leal05) também apresenta uma descrição desse uso.
- **Finalização** – As linguagens Modula-3, Python e Haskell utilizam referências fracas na implementação de finalizadores. Também encontramos esse uso através do questionário (um programador disse já ter usado referências fracas como mecanismo alternativo de finalização, para desalocar recursos externos).

### 3.1.1 Coleta de Referências Cíclicas

Coletores de lixo que usam exclusivamente contagem de referências para detectar objetos não mais alcançáveis pelo programa não são capazes de coletar referências cíclicas, ou seja, não são capazes de coletar objetos que se referenciam mutuamente, gerando vazamento de memória. De acordo com Brownbridge (Brownbridge85), essa deficiência foi uma das motivações iniciais para o desenvolvimento e uso do mecanismo de referências fracas. A solução para o problema consiste em quebrar o ciclo de referências através da substituição de algumas referências ordinárias por referências fracas, de tal forma que qualquer ciclo de referências seja composto por pelo menos uma referência fraca. É responsabilidade do programador garantir essa quebra. Um exemplo disso encontra-se na Figura 3.1. O número ao lado de cada objeto corresponde ao seu contador de referências. O objeto A é referenciado de um ponto fora do ciclo e também pelo objeto C. Como essa última referência é ordinária, mesmo que a primeira referência seja removida o contador de A continuará maior que zero e nenhum objeto no ciclo será coletado. Ao substituir a referência ordinária de C para A por uma referência fraca, o contador de A chega a zero assim que a referência vinda de fora do ciclo é removida. Sendo assim, A será coletado e, consequentemente, os outros objetos.

### 3.1.2 Cache

Uma maneira interessante de gerenciar recursos externos é manter os objetos proxy que os representam na memória, ao invés de removê-los sempre

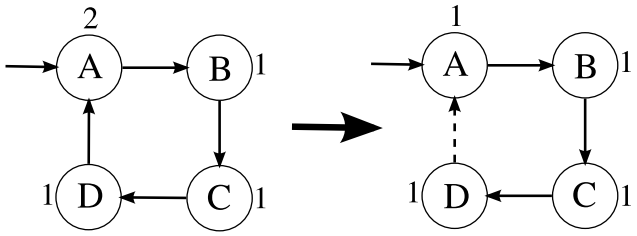


Figura 3.1: Coletando ciclos através de referências fracas

que não são mais necessários. Contudo, a existência de muitos recursos externos pode acabar sobrecarregando a memória, tornando interessante remover instâncias não utilizadas por um longo período. Além disso, pode ser necessário impedir que diferentes instâncias sejam mapeadas ao mesmo recurso externo, do contrário estaríamos criando uma dificuldade extra para manter uma sincronização entre essas instâncias. Essas funcionalidades podem ser oferecidas através de uma cache de instâncias e de uma função que retorne a instância do recurso desejado, ou crie uma nova caso ainda não exista uma instância para o recurso.

Podemos também utilizar uma cache de objetos em aplicações que fazem uso de estruturas de dados de tamanho significativo. Tais aplicações podem melhorar sensivelmente seu desempenho ao manter essas estruturas residentes na memória. Essa política contudo pode levar a um rápido esgotamento da memória livre. Uma cache pode ser usada para preservar os objetos apenas enquanto o nível de utilização da memória não for restritivo.

A dificuldade nessas duas abordagens está no fato de que uma entre duas coisas deve ser tolerada: ou a cache pode crescer indefinidamente ou existe alguma forma explícita de gerenciamento da cache na aplicação. Essa última pode ser bastante tediosa e levar a mais código do que é realmente necessário para resolver o problema. A primeira é inaceitável em processos com um longo tempo de execução ou mesmo em processos que fazem uso substancial da memória.

O uso de referências fracas possibilita a implementação de caches gerenciadas automaticamente. Considere como exemplo uma aplicação que utiliza uma tabela (um *result set*) com milhares de dados provenientes de um banco de dados. Após carregar os dados, a aplicação remove todas as referências para a tabela, exceto por uma referência fraca. Sempre que for necessário acessar algum dado na tabela, a aplicação primeiro verifica se a mesma ainda está acessível. Caso a tabela tenha sido coletada a aplicação recarrega os dados diretamente do banco de dados através de uma nova consulta.

Uma outra aplicação interessante de caches são funções memorizadas

(*memorizing functions*) (Ierusalimschy06, Jones00). O esforço computacional necessário para atender à invocação de uma função complexa pode ser reduzido salvando-se o resultado de cada invocação. Caso a função seja chamada novamente com o mesmo argumento, ela retorna o valor salvo. A utilização de um cache evita que a memorização dos resultados leve a um esgotamento da memória, preservando os resultados acessados mais recentemente (e possivelmente mais frequentemente).

Vale observar que em sistemas com coletores de lixo que usam contagem de referência os objetos são coletados imediatamente após sua contagem atingir zero, o que impede a implementação de caches usando-se apenas o mecanismo básico de referências fracas. Uma solução seria criar um segundo tipo de referência fraca que impediria a coleta do objeto enquanto a memória não atingisse níveis críticos.

### 3.1.3 Tabelas de Propriedades

Por várias razões, pode ser necessário associar atributos adicionais a um objeto sem modificar a sua estrutura interna. Em aplicações orientadas a objetos por exemplo, podemos querer associar dinamicamente um ou mais atributos a um objeto independente dos atributos de sua classe. Essa associação pode ser feita via uma estrutura de dados externa, como uma tabela associativa, que nesse caso é mais comumente chamada de tabela de propriedades. O objeto em questão deve então ser usado como chave nessa tabela, mapeando em seus atributos extras. Contudo, isso irá acarretar um problema: a referência na tabela impedirá que o objeto seja coletado. Vimos na Seção 2.3 que tabelas fracas podem ser usadas para representar uma tabela de propriedades, corrigindo o problema. Isso ocorre porque, ao invés de usar como chave de busca o próprio objeto, usamos uma referência fraca. Assim, quando o objeto não for mais usado pelo programa, a referência fraca poderá ser removida e o objeto poderá ser coletado.

Na linguagem Lua referências fracas são usadas através de tabelas fracas. O programador pode criar uma tabela onde apenas a chave é fraca, apenas o valor é fraco, ou ambos são fracos. Isso torna a implementação de tabelas de propriedades bastante simples.

### 3.1.4 Conjuntos Fracos

Conjuntos fracos (weak sets) podem ser entendidos como um conjunto de objetos cuja pertinência a esse conjunto não acarrete uma referência para o

objeto. Ou seja, sempre que objetos tem que ser processados como um grupo, mas sem interferir no ciclo de vida de cada um, podemos usar weak sets como uma solução de implementação.

Um exemplo bastante comum é o padrão de projeto *Observer* (Gamma95) que define uma dependência de um-para-vários entre um objeto observado e os objetos observadores. Quando o estado de um objeto observado se modifica, todos os seus observadores são notificados e tem o seu estado atualizado automaticamente. Geralmente, essa notificação é feita pelo objeto observado, que portanto precisa conhecer seus observadores, mantendo referências para os mesmos. O uso de referências fracas para mapear o conjunto de observadores permite manter um acoplamento mínimo entre os objetos e não impede que os observadores sejam coletados.

Note que, para implementar conjuntos fracos, caches e tabelas de propriedades, precisamos construir uma estrutura formada de referências fracas. No caso de conjuntos fracos, temos um conjunto de referências fracas que apontam para os observadores. A implementação de uma cache requer um conjunto de referências fracas para manter os objetos na memória. Por fim, no caso da tabela de propriedades, construímos uma tabela onde as chaves são mantidas por referências fracas. Caso a linguagem utilizada já forneça suporte a essas estruturas, a implementação desses usos é bastante simplificada. A linguagem Lua, por exemplo, fornece referências fracas através de tabelas fraca, como vimos na Seção 2.3. O programador pode criar uma tabela onde apenas a chave é fraca para representar uma tabela de propriedades, ou pode criar uma tabela com índices numéricos onde apenas os valores são fracos para representar um conjunto fraco ou uma cache.



### 3.1.5 Finalização

Referências fracas, acrescidas de um mecanismo de notificação, também podem ser utilizadas para informar o programa cliente que um objeto foi coletado, eventualmente disparando automaticamente rotinas associadas a tal evento. Como já foi visto, essa dinâmica constitui um exemplo do mecanismo de finalização baseado em coleções, e evita alguns dos problemas associados a finalizadores baseados em classes. Esse importante uso de referências fracas será abordado na próxima seção.

## 3.2

### Finalizadores

De acordo com nossa pesquisa, o uso de finalizadores é bastante comum se comparado ao uso de referências fracas. Nesta seção, mostramos os usos encontrados para finalizadores tradicionais e fornecemos implementações alternativas que utilizam referências fracas. Os usos encontrados para finalizadores são:

- **Desalocação de Memória** – Esse uso foi primeiramente encontrado através do questionário enviado aos programadores. Em particular, quando programas em Lua fazem referência a bibliotecas em C, vimos que é bastante comum utilizar finalizadores para reciclar a memória alocada por essas bibliotecas. Posteriormente, constatamos que alguns trabalhos (Boehm03, Leal05) já haviam descrito esse uso. Boehm (Boehm03) o chama de *Legacy Libraries* e diz que talvez seja o uso mais comum de finalizadores. Nossa pesquisa também chegou a essa conclusão, pois 39% dos programadores que disseram já ter usado finalizadores o fizeram para desalocar memória.
- **Desalocação de Recursos Externos** – Esse uso foi encontrado através do questionário enviado aos programadores. Entre os programadores que disseram já ter usado finalizadores, 14% responderam que o fizeram para desalocar recursos externos. Existem dois casos particulares desse uso que foram encontrados no trabalho de Boehm (Boehm03), são eles: o uso de finalizadores para remoção de arquivos temporários e para obtenção de informações de conectividade[1].
- **Coleta de Referências Cíclicas** – A linguagem Perl usa finalizadores para coletar referências cíclicas (Christiansen03). Também encontramos uma descrição desse uso no trabalho de Leal (Leal05).
- **Fallback** – Encontrado apenas no trabalho de Leal (Leal05).
- **Reciclagem de Objetos** – Esse uso de finalizadores também foi encontrado apenas no trabalho de Leal (Leal05).

Como um de nossos objetivos, mostramos que finalizadores e callbacks possuem características semelhantes. Além disso, comparamos esses mecanismos com o mecanismo de notificação passiva. Contudo, antes de iniciar qualquer discussão sobre as implementações, gostaríamos de identificar características importantes da linguagem utilizada. Essas características podem influenciar na escolha da melhor abordagem.

[1]Como a memória pode ser vista como um recursos externo, o uso de finalizadores na desalocação de memória também pode ser considerado um caso particular de desalocação de recursos externos.

Uma das características mais importantes diz respeito aos mecanismos de concorrência e sincronização oferecidos pela linguagem. Como vimos na Seção 2.2, a execução de callbacks e finalizadores pode introduzir linhas de execução concorrentes na aplicação. O uso de mecanismos de sincronização associados a callbacks e finalizadores é importante e mesmo inevitável em muitas situações, sobretudo quando esses últimos acessam variáveis globais. Infelizmente, o uso de mecanismos de sincronização tende a deteriorar o desempenho da aplicação. Esse uso também é uma fonte constante de erros de programação, incluindo deadlocks. A falta de mecanismos de concorrência e sincronização suficientemente expressivos dificulta bastante a implementação de programas que usam finalizadores e callbacks. Além disso, a complexidade inerente à concorrência e a sincronização desses mecanismos pode levar até mesmo a erros de implementação da linguagem. Para evitar esse problema, é preferível usar um mecanismo de notificação passiva, onde o próprio programa é responsável por invocar a rotina de finalização de um objeto em particular. Outra característica importante do contexto refere-se ao algoritmo de coleta de lixo utilizado pelo coletor da linguagem. Existem duas abordagem básicas que devemos levar em consideração: contagem de referências e rastreamento. A primeira abordagem recicla a memória alocada para um objeto assim que esse não pode mais ser acessado pelo programa. A segunda abordagem não é determinística e pode haver uma demora até o objeto ser removido. Dependendo do algoritmo, devemos considerar extensões do mecanismo de referências fracas para não inviabilizar a implementação.

Outra característica a ser considerada é o paradigma da linguagem de programação utilizada. Independente do tipo de notificação, passiva ou ativa, a implementação de finalizadores através de referências fracas constitui uma finalização baseada em coleções onde a rotina de finalização é totalmente desacoplada do objeto. No entanto, finalizadores totalmente independentes do estado do objeto a ser finalizado não são muito úteis, pois as rotinas de finalização normalmente dependem de informações armazenadas pelos objetos finalizados. Para ter acesso a essas informações, uma implementação de finalizadores via referências fracas pode armazená-las numa estrutura de dados alternativa. Contudo, em linguagens orientadas a objetos isso significa uma quebra no encapsulamento. Um modo de atenuar esse problema é passar o objeto como parâmetro do callback, no caso da notificação ativa, ou armazenar o objeto na fila de notificações, no caso da notificação passiva. Porém, ainda assim dependemos da visibilidade das informações e da existência de métodos de acesso e controle. Por fim, a última característica que devemos considerar refere-se à garantia de execução dos finalizadores. Algumas linguagens de pro-

gramação não garantem que todos os finalizadores serão executados antes do término da aplicação. No caso de referência fracas, essa garantia pode não existir quanto à execução dos callbacks que implementam rotinas de finalização. Essa falta de garantia pode dificultar o uso de callbacks ou finalizadores para liberar recursos que não são liberados pelo sistema operacional ao término da aplicação, como veremos no caso de remoção de arquivos temporários.

### 3.2.1 Desalocação de Memória

Por vezes, é necessário que programas escritos em diferentes linguagens interajam. O trabalho de Muhammad (Muhammad06) discute uma série de questões de projeto ligadas à interoperabilidade de linguages de programação. Dentre as questões apresentadas, uma particularmente interessante é a diferença entre os modelos de gerenciamento de memória das linguagens de programação, mais especificamente, a interação entre linguagens com gerenciamento automático de memória e linguagens sem esse tipo de suporte. Um exemplo dessa interação ocorre quando programas escritos em Java (SUN04) fazem referência a bibliotecas escritas na linguagem C++, que não possui gerenciamento automático de memória. As funções dessas bibliotecas podem acabar alocando regiões de memória que não poderão ser liberadas automaticamente pelo coletor de Java. A solução é utilizar finalizadores para liberar a memória alocada com uma rotina nativa como `malloc`, ampliando assim a região de memória gerenciada pelo coletor de lixo.

A reciclagem da memória alocada por uma rotina nativa, escrita numa linguagem que não possui coleta de lixo automática, também pode ser feita através de um mecanismo de referências fracas que ofereça suporte a notificação ativa ou passiva. Para isso, o programa deve referenciar os objetos criados através das bibliotecas nativas via uma referência fraca. No caso de optarmos por notificação ativa, a rotina de finalização deve ser inserida em um callback associado à referência fraca. Essa implementação é bastante semelhante a finalizadores tradicionais, pois os callbacks também são executados automaticamente pelo coletor de lixo. Contudo, como vimos anteriormente, existem dois problemas quando utilizamos finalizadores ou callbacks para desalocar memória. O primeiro deles refere-se à possível introdução de linhas de execução concorrentes na aplicação. Caso a linguagem utilizada não ofereça um suporte adequado a concorrência e sincronização, o uso desses mecanismos pode levar a inconsistências durante a execução do programa. O segundo problema refere-se à impossibilidade de determinar quando finalizadores ou callbacks serão executados em coletores de lixo baseados em rastreamento.

Atrasos na liberação da memória podem ser gerados caso esses mecanismos não sejam executados no momento correto e esses atrasos podem afetar o desempenho da aplicação.

A outra abordagem que pode ser utilizada é a notificação passiva. Nesse caso, quando o coletor identifica que existem apenas referências fracas para o objeto que representa uma biblioteca em C ou C++, ele insere esse objeto em uma fila de notificações. Quando o programa achar adequado, por exemplo, quando o nível de memória ultrapassar certos limites, ele pode acessar explicitamente a fila de notificações e, a seguir, o programa pode executar a rotina de finalização adequada para cada objeto encontrado na fila. Apesar de perder um pouco de automação, o uso de notificação passiva elimina os problemas de concorrência. O indeterminismo ainda permanece, mas em menor escala. Isso se deve ao fato de que o coletor ainda é responsável por inserir o objeto na fila de notificações, após determinar que não existem mais referências ordinárias para ele. Contudo, ao contrário do que ocorre com finalizadores e callbacks, o próprio programa é responsável por invocar as rotinas de finalização e não o coletor de lixo.



### 3.2.2 Desalocação de Recursos Externos

Determinados recursos externos, como descritores de arquivos e conexões, são liberados pelo sistema operacional assim que o programa termina sua execução. Contudo, tais recursos não são infinitos e caso o programa não os libere a medida que não são mais necessários, pode haver um esgotamento dos mesmos. Geralmente, recursos externos são representados através de um objeto proxy, cujo ciclo de vida segue o padrão de utilização do recurso correspondente. Sendo assim, podemos alocar e desalocar automaticamente o recurso associado ao proxy através do uso de construtores e finalizadores. Para isso, basta inserir a rotina para alocação do recurso dentro do construtor e a rotina de desalocação dentro do finalizador referentes ao proxy, estendendo assim a capacidade de gerenciamento do coletor de lixo. Como sabemos, a rotina de finalização também pode ser implementada através de um callback associado a uma referência fraca para o proxy.

Assim como na seção anterior, finalizadores e callbacks nem sempre são adequados quando a linguagem utilizada possui um coletor de lixo baseado em rastreamento. Nesse caso, não é possível determinar quando a coleta será iniciada, consequentemente, também não é possível determinar quando os finalizadores serão executados e quando os recursos serão liberados. Atrasos na liberação dos recursos podem afetar negativamente o desempenho da aplicação.

Esses atrasos podem ser gerados quando os finalizadores não são executados no momento correto. Em alguns casos porém, é possível atenuar esse problema forçando explicitamente a execução do coletor de lixo, por exemplo, chamando o método `System.gc()` em Java. O coletor de lixo deve ser invocado sempre que a disponibilidade de recursos externos cair abaixo de níveis pré-definidos.

O problema de desalocação de recursos externos também pode ser abordado via um mecanismo de notificação passiva. Para isso, associamos uma referência fraca a cada proxy e quando a memória atingir níveis críticos o programa acessa então a fila de notificações. Contudo, como essa fila é preenchida automaticamente pelo coletor, é possível que nem todos os objetos finalizáveis estejam na fila de notificações. Sendo assim, caso a finalização dos objetos na fila não seja suficiente, pode ser necessário forçar explicitamente a execução do coletor para que os objetos finalizáveis que não se encontram na fila sejam inseridos nela. Essa abordagem pode ser vantajosa caso o acesso à fila de notificações seja feito logo após uma coleta, pois assim, todos os objetos finalizáveis estarão na fila. Porém, caso exista um longo intervalo de tempo entre uma coleta e um posterior acesso à fila de notificações, a utilização de finalizadores ou callbacks pode ser mais eficaz, pois os recursos já terão sido liberados enquanto que na notificação passiva o programa ainda precisará acessar a fila de notificações. Não devemos nos esquecer, no entanto, que o mecanismo de notificação passiva, ao contrário de finalizadores e callbacks, não apresenta problemas de concorrência e sincronização, o que o torna de fato mais vantajoso.



### Remoção de Arquivos Temporários

Existem alguns recursos externos, como arquivos temporários criados pelo programa, que não são liberados pelo sistema operacional assim que o programa termina sua execução. Nesse caso, o próprio programa deve ser responsável pela remoção dos arquivos. Durante sua execução, o programa pode remover alguns arquivos temporários que não serão mais utilizados, porém, por vezes é necessário prover uma remoção mais confiável desses arquivos antes do término do programa. Finalizadores ou callbacks podem ser usados para tal fim, porém com algumas ressalvas. Dependendo da linguagem, não existe garantia de que esses mecanismos serão executados ao final do programa, consequentemente, alguns arquivos temporários poderão ainda permanecer.

A solução mais simples é garantir a execução dos finalizadores, ou dos callbacks, forçando uma coleta de lixo especial ao final do programa. Outra solução, descrita no trabalho de Boehm (Boehm03), sugere a implementação de uma rotina alternativa que deve ser chamada ao final da execução do programa,

enquanto finalizadores removem arquivos temporários durante a execução. No caso de descritores de arquivos e conexões, é irrelevante se a linguagem executa ou não os finalizadores ou os callbacks ao final do programa, pois o sistema operacional se encarrega de liberar os recursos.

Assim como ocorre quando usamos simplesmente finalizadores ou callbacks, o mecanismo de notificação passiva também não garante que todos os arquivos serão removidos ao final do programa. Sempre que necessário, podemos acessar a fila de notificações durante a execução do programa e remover os arquivos referentes aos objetos encontrados. Porém, quando acessamos a fila ao final de execução, o coletor pode não ter inserido todos os proxys que representam arquivos temporários na fila de notificações. Isso irá impossibilitar que todos os arquivos temporários sejam removidos.

Existe uma solução bastante simples, via referência fracas, que não requer a invocação de uma coleta de lixo especial ao final do programa. Essa solução consiste em manter os proxys em um conjunto fraco e utilizar notificação passiva ou ativa para remover os arquivos durante a execução do programa. À medida que os proxys são finalizados, suas respectivas referências contidas no conjunto fraco são removidas. Ao final da execução do programa, restam no conjunto fraco apenas os proxys referentes aos arquivos temporários que não foram removidos. Podemos então acessar esse conjunto e remover os arquivos temporários explicitamente.



### Obtenção de Informações de Conectividade

Em alguns casos, a desalocação explícita de recursos externos pode ser uma tarefa bastante complexa e ineficiente. Contudo, podemos nos beneficiar das informações sobre a conectividade dos objetos fornecidas pelo coletor de lixo a fim de facilitar o trabalho de liberação de recursos.

O trabalho de Boehm (Boehm03) descreve uma aplicação que usa grafos acíclicos direcionados (DAGs) contendo descritores de arquivos em seus nós terminais. Em função do tamanho e complexidade das estruturas de dados utilizadas, bem como do padrão de acesso da aplicação, é extremamente difícil rastrear explicitamente todas as referências para cada descritor de arquivos, a fim de fechá-los explicitamente quando necessário. A solução encontrada foi associar finalizadores aos nós terminais, fechando automaticamente os arquivos — como sabemos, isso também pode ser feito através de callbacks. Obviamente, essa solução pode levar os arquivos a ficarem abertos mais tempo do que o necessário, devido ao indeterminismo da coleta de lixo em linguagens com coletores baseados na técnica de rastreamento.

Da mesma forma que nos usos anteriores, podemos utilizar notificação

passiva. Nesse caso, o coletor irá inserir o descritor do arquivo na fila de notificações quando não existirem mais referências ordinárias para ele. Sempre que necessário, o programa pode acessar a fila e fechar os arquivos. Atrasos no fechamento dos arquivos ainda podem ocorrer, pois para fechar um arquivo o proxy que o representa já deve ter sido inserido na fila pelo coletor. Contudo, os problemas de concorrência e sincronização são evitados.

### 3.2.3 Coleta de Referências Cíclicas

Coletores de lixo que utilizam exclusivamente contagem de referências são incapazes de liberar a memória ocupada por estruturas de dados cíclicas. Na linguagem Perl, o suporte a detecção de ciclos deve ser feito através de finalizadores, como mostram Christiansen e Torkington (Christiansen03). A solução proposta consiste em primeiro definir um *container* (um classe de objetos cujo propósito é conter outros objetos) para cada estrutura de dados recursiva ou potencialmente cíclica, como anéis, grafos e listas duplamente encadeadas. Todo acesso à estrutura deve ser feito através do seu container. A seguir, deve-se implementar um finalizador para cada container definido. A responsabilidade do finalizador é examinar a estrutura potencialmente cíclica armazenada em seu container a fim detectar e eliminar explicitamente os ciclos existentes. O finalizador é invocado quando a contagem de referências de seu respectivo container atinge zero. Isso garante que todos os objetos de uma estrutura potencialmente cíclica são coletados imediatamente após a última referência para seu container ser destruída.

A dificuldade associada a coleta de referência cíclicas em coletores que usam apenas contagem de referências foi uma das motivações iniciais para o desenvolvimento e uso de referências fracas (Brownbridge85). É bem mais intuitivo e adequado usar referências fracas para resolver esse problema do que finalizadores. A solução é simples: basta substituir referências ordinárias por referências fracas de tal forma que qualquer ciclo de referências seja composto por ao menos uma referência fraca, assim como visto na Seção 3.1.1.

### 3.2.4 Fallback

Aplicações que fazem uso intensivo de recursos computacionais limitados precisam liberar tais recursos à medida em que eles não são mais utilizados. Vimos na Seção 3.2.2 que finalizadores nem sempre são uma solução adequada para liberar recursos externos, pois coletores de lixo baseados em rastreamento podem atrasar a realização dessa operação. Foi sugerido nessa seção que a

coleta de lixo fosse explicitamente forçada sempre que disponibilidade de recursos caísse abaixo de níveis pré-estabelecidos. No entanto, ao invés de forçar toda a coleta de lixo, o ideal seria prover um método para realizar apenas a liberação do recurso. Como erros nesse processo de desalocação são bastante comuns, finalizadores seriam utilizados como um mecanismo de segurança (*fallback*) para liberar os recursos externos que deveriam ter sido liberados explicitamente. Ou seja, o finalizador verifica se o recurso já foi liberado e, caso ainda não tenho sido, executa uma rotina para desalocá-lo. Ainda que não existam garantias sobre quando ou mesmo se o finalizador vai ser invocado, essa pequena redundância é uma solução melhor do que depender apenas da liberação explícita dos recursos, que podem acabar não sendo liberado caso o programador cometa algum erro. Algumas classes da biblioteca padrão da linguagem Java, como `LogFileManager`, implementam essa solução (Venners98).

No geral, fallbacks são mecanismos utilizados para dar continuidade à execução do programa mesmo após a ocorrência de falhas. Uma facilidade bastante comum nas linguagens de programação e que caracteriza bem um fallback é o bloco `try-finally` (Sebesta02). Essa construção é utilizada basicamente para o tratamento de exceções; porém, ela também pode ser vista como um mecanismo de finalização baseado na estrutura sintática do programa. A Listagem 3 mostra a estrutura geral do bloco `try-finally`. Após sair do escopo definido pela cláusula `try`, mesmo que existam exceções ainda não tratadas, o fluxo de execução do programa é imediatamente transferido para a cláusula `finally`.

**Listagem 3** Bloco `try-finally` com tratadores.

```
try {
...
}
catch(...) {
...
}
... //Outros tratadores
finally{
...
}
```

Essa estrutura pode ser adaptada para funcionar como um finalizador no caso da desalocação de recursos externos tratado no primeiro parágrafo. Para isso, a cláusula `try` deve conter o bloco de código referente à criação e à utilização do recurso e a cláusula `finally`, por sua vez, deve assegurar

que o recurso será liberado. Mesmo que uma exceção interrompa prematuramente a execução do bloco `try`, o fluxo será transferido para a cláusula `finally`, garantindo a liberação do recurso. Um exemplo disso se encontra na Listagem 4.

**Listagem 4** Liberando recursos com o bloco `try-finally`.

```
Proxy p;
try {
    p = new Proxy();
    foo(p);
}
finally{
    p.freeResources();
}
```

Infelizmente, o uso do bloco `try-finally` como um mecanismo de finalização para liberar recursos externos possui algumas desvantagens em relação a finalizadores tradicionais. Em primeiro lugar, é mais simples liberar um recurso através do finalizador do proxy que o representa do que inserir blocos `try-finally` em cada região do programa em que o proxy é criado. Em segundo lugar, a necessidade de finalização está quase sempre associada a tipos abstratos de dados, e não à estrutura sintática do programa (Hayes92, Schwartz81).

Pode parecer que no tratamento de exceções é vantajoso usar o bloco `try-finally` como finalizador. Porém, quando a implementação da linguagem está programada para executar um coleta de lixo sempre que ocorrer uma exceção, basta definir um finalizador para liberar o recurso, o que elimina a necessidade do bloco `try-finally`. Mesmo que uma exceção impeça o recurso de ser liberado explicitamente, a execução do coletor e, consequentemente, do finalizador irá garantir a liberação do recurso.

### 3.2.5 Reciclagem de Objetos

Determinados objetos possuem um custo de criação relativamente alto devido a sua complexidade. Uma aplicação pode obter ganhos de desempenho significativos se, ao invés de sempre criar esses objetos, ela reciclar os que não são mais utilizados. Uma forma de efetuar essa reciclagem é através da definição de finalizadores para todos os objetos recicláveis. A rotina de finalização deve decidir se o objeto correspondente será coletado ou reciclado, o que vai depender de alguns parâmetros relevantes, como o número de objetos

recicláveis disponíveis no sistema ou a quantidade de memória livre. Novos objetos só são criados quando não existirem objetos recicláveis livres.

Algumas linguagem, como Java, não permitem que o finalizador associado a um objeto seja executado mais de uma vez. Nesse caso, geralmente não é muito útil implementar esse tipo de solução, já que que um objeto poderá ser reciclado uma única vez.

A reciclagem de objetos com finalizadores dá origem ao que chamamos de *objetos ressucitáveis*. Um objeto ressucitável é gerado quando a aplicação não pode mais acessar o objeto a partir do momento em que este se torna desconexo, porém o objeto pode influenciar no comportamento futuro da aplicação. No mecanismo de notificação ativa isso ocorre apenas quando o coletor de lixo limpa a referência fraca antes de executar o callback associado a ela. No caso da notificação passiva, não há meios de existirem objetos ressucitáveis.

Para uma implementação via um mecanismo de notificação passiva, todos os objetos candidatos a serem reciclados devem possuir uma referência fraca. O coletor irá inserir o objeto candidato à reciclagem em uma fila de notificações quando não existirem mais referências ordinárias para ele. O programa é então responsável por acessar a fila e reciclar os objetos necessários.

Outra forma de implementar essa reciclagem é inserindo o objeto em uma tabela fraca, de forma que a tabela mantenha o objeto através de uma referência fraca. Nesse caso, quando não mais existirem referências ordinárias para o objeto, ele permanecerá um tempo armazenado na tabela fraca até ser coletado. Nesse tempo, ele pode ser reciclado se necessário. Porém, essa abordagem tráz limitações óbvias. Caso o coletor de lixo seja baseado exclusivamente em contagem de referências, o objeto é coletado imediatamente após a última referência ordinária ser removida. Nesse caso, nunca haverá tempo para ele ser reciclado. No entanto, isso pode ser contornado com a criação de um tipo especial de referências fracas que impede que o objeto seja coletado enquanto a memória não atingir níveis críticos.

## 3.3 Conclusões

Mesmo sendo pouco conhecido e pouco utilizado, o mecanismo de referências fracas consitui-se numa alternativa elegante para a implementação de um mecanismo de finalização. Vimos que callbacks e finalizadores tradicionais possuem características semelhantes; mais especificamente, ambos possuem o mesmo nível de automação na execução das rotinas de finalização e ambos apresentam problemas de concorrência, sincronização e indeterminismo. No

entanto, o mecanismo de notificação passiva apresenta algumas vantagens em relação a callbacks e finalizadores, pois elimina os problemas de concorrência e sincronização e diminui o indeterminismo. Porém, um pouco de automação é perdida, já que o programador é responsável por chamar as rotinas de finalização. Dependendo da aplicação, isso pode acabar trazendo algumas desvantagens. No entanto, para alguns dos usos de finalizadores encontrados é até mais adequado e intuitivo utilizar notificação passiva que callbacks ou finalizadores. Por esses motivos, acreditamos que toda implementação de referências fracas deveria fornecer suporte a um mecanismo de notificação passiva, mesmo que a implementação já ofereça suporte a finalizadores ou callbacks.

Com base no que foi discutido neste capítulo, decidimos implementar um mecanismo de notificação passiva para a linguagem Lua. Esse mecanismo será descrito no Capítulo 4.