Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

INTRODUZINDO CONCORRÊNCIA EM JAVA

Rafael Machado Duarte

TRABALHO DE GRADUAÇÃO

Recife
25 de agosto de 2005

Universidade Federal de Pernambuco
Centro de Informática

Rafael Machado Duarte

INTRODUZINDO CONCORRÊNCIA EM JAVA

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Alexandre Cabral Mota*

Recife
25 de agosto de 2005

aos meus pais

AGRADECIMENTOS

Antes de todos gostaria de agradecer a minha família. Graças a ela tive a possibilidade de entrar na universidade e terminar minha graduação. Muito eu devo aos exemplos dados pelos meus pais, que sempre me incentivaram a ir adiante, a aprofundar minha formação. A eles dedico meus agradecimentos especiais.

Gostaria de agradecer a meu orientador, Alexandre Mota por sua dedicação, sempre me auxiliando a levar adiante o trabalho e esclarecendo minhas dúvidas. Gostaria de agradecer também ao professor Augusto Sampaio, que junto com Alexandre, me deram a oportunidade de iniciar minhas pesquisas em métodos formais e assim despertar meu interesse pela área de concorrência.

Agradeço também a todos meus amigos do CIn, por seu companheirismo e amizade, indispensáveis nos momentos difíceis que passamos durante mais de quatro anos juntos.

Dedico agradecimentos também a meus amigos de trabalho do projeto CENAS no CESAR, pelo ótimo ambiente de trabalho, engrandecedor profissionalmente e pessoalmente.

And you've been so busy lately
That you haven't found the time
To open up your mind
And watch the world spinning gently out of time
—DAMON ALBARN (Out of Time)

RESUMO

Java é uma das linguagens de programação mais populares na comunidade de desenvolvimento. Mesmo assim, nem todos seus recursos são usados; concorrência é um tal recurso.

A dificuldade em usar concorrência em Java se deve ao fato de os programadores estarem mais acostumados com o desenvolvimento puramente seqüencial. E os programas Java concorrentes, em geral, são dotados de problemas inerentes a esse paradigma; dentre os problemas mais comuns, encontramos *deadlocks*, *livelocks* e não-determinismo.

O objetivo deste trabalho é propor regras de reescrita de código Java intencionando introduzir concorrência de forma segura. Isto é, o aspecto concorrente é introduzido sem alterar o significado original do programa, bem como não criar programas com *deadlock*, *livelocks* e não-determinismo.

Como resultados principais de nosso esforço destacamos o potencial aumento de performance do programa e a obtenção de código seguro por construção.

Palavras-chave: Sistemas concorrentes, transformação de programas, Java

ABSTRACT

Java is one of the most popular programming languages nowadays. Nevertheless, not all its features are used; concurrency is one of these.

Using concurrency in Java is difficult because programmers are used to conventional sequential programming. Moreover, concurrent Java programs, in general, have the inherent problems of this paradigm; as common problems, we could cite deadlock, livelock and nondeterminism.

This work aims on proposing Java code rewriting rules to safely introduce concurrency. That is, the concurrent aspect is introduced without changing the original program meaning, as well as not creating programs with deadlock, livelock and nondeterminism.

As the main results of this work, we highlight the potential performance improvement and code safety ensured by construction.

Keywords: Concurrent systems, program transformation, Java

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1

INTRODUÇÃO

O desenvolvimento de sistemas concorrentes é bastante difícil. Java, apesar de possuir um modelo de concorrência bem avançado - como será visto no Capítulo 2, não foge à regra. Os desenvolvedores têm muita dificuldade em construir e manter sistemas concorrentes em Java, principalmente por causa das diferenças entre a programação seqüencial e concorrente. E quando se consegue construir o sistema, seu código não é confiável, pois é vítima de erros inerentes à concorrência, tais quais *deadlock*, *livelock* e não-determinismo.

Considere o seguinte exemplo de classe em Java:

```
public class Conta {
  ...
    private double saldo;

    public void debitar(double valor) {
        this.saldo = this.saldo - valor;
    }

    public void creditar(double valor) {
        this.saldo = this.saldo + valor;
    }
  ...
}
```

Pode-se perceber que se trata de uma classe bem simples, que representa uma Conta de um banco. Uma conta possui um atributo que representa o saldo e duas operações para manipulá-lo: debitar e creditar. Enquanto a execução for seqüencial, não há problema algum com a classe acima. Chamadas aos métodos nunca são executadas simultaneamente e o valor do saldo se manterá consistente.

Caso haja acesso concorrente à classe Conta, ou seja, mais de um thread chamando seus métodos, não há garantia de que os métodos não serão executados simultaneamente. O problema é que ao se compilar o código Java, é gerado um código similar a Assembly[1] e ao se executar um sistema concorrente, a máquina virtual Java terá que dividir o tempo de processamento entre os threads que estiverem executando. Assim sendo, a execução de um método por um thread pode ser interrompida para que outro thread possa executar. Este fato pode acarretar na corrupção do valor do saldo - um crédito de R$100 seguido de um débito de R$100 nem sempre deixará o saldo inalterado - algo inaceitável para o sistema.

[1] Também conhecido como código de máquina, ou seja, instruções para o processador

Um modo bastante simples de evitar este problema seria transformar o programa, adequando-o à execução concorrente. Para tanto, pode-se aplicar uma das regras propostas no Capítulo 3; esta regra é aplicada quando se deseja que algum método seja executado atomicamente, ou seja, sua execução não pode ser interrompida pelo sistema de execução para que outro thread possa executar. Ela consiste simplesmente em adicionar o modificador `synchronized` à declaração do método; assim sendo, os métodos mostrados anteriormente ficariam do seguinte modo:

```
...
  public synchronized void debitar(double valor) {
      this.saldo = this.saldo - valor;
  }
  public synchronized void creditar(double valor) {
      this.saldo = this.saldo + valor;
  }
...
```

Este foi apenas um pequeno exemplo dos problemas que acontecem ao se desenvolver um sistema concorrente. O Capítulo 3 mostrará uma série de regras que podem ser aplicadas durante esse desenvolvimento, enquanto que o Capítulo 4 trará um estudo de caso para ilustrar a aplicação de tais regras.

A introdução de concorrência em um sistema pode trazer diversos benefícios, dentre os quais podemos citar:

- Potencial melhora de desempenho
- Melhor aproveitamento do processador
- Respostas mais rápidas ao usuário

Esta é a motivação da abordagem proposta neste trabalho: introduzir concorrência em um sistema Java seqüencial, permitindo que ele usufrua dos benefícios de concorrência, mas garantindo que ele permaneça correto.

CAPÍTULO 2

PROGRAMAÇÃO CONCORRENTE EM JAVA

Este capítulo fará uma introdução aos conceitos de programação concorrente, descrevendo quais suas motivações, algumas abordagens e problemas comuns. Em seguida, será mostrado como Java suporta a programação concorrente e quais os recursos que a linguagem fornece para tal fim.

2.1 PROGRAMAÇÃO CONCORRENTE

Programação concorrente [And00, AS83] é o nome dado às notações usadas para expressar um paralelismo (potencial) na programação de aplicações. Ela também envolve as técnicas utilizadas para lidar com comunicação e sincronização entre entidades (potencialmente) paralelas.

A programação concorrente teve seu início em 1962, quando foram construídos os primeiros computadores capazes de terem a CPU executando instruções ao mesmo tempo que eram realizadas operações de entrada e saída (E/S). Desde então, os desenvolvedores de sistemas se depararam com um novo modo de se programar sistemas e, inevitavelmente, com as dificuldades inerentes à concorrência.

Há três principais motivações para a criação de programas concorrentes:

Utilizar totalmente o processador - processadores modernos executam em velocidades bem superiores as dos dispositivos de entrada e saída com quem interagem. Assim, um programa seqüencial que está esperando por E/S não é capaz de executar outras operações, desperdiçando poder de processamento. Considerando um programa que deseja realizar operações de E/S intensivas, como em uma animação, e ainda precisa calcular a próxima seqüência de imagens, realizar essas atividades seqüencialmente teria um efeito drástico na velocidade de execução do programa.

Permitir que mais de um processador resolva o problema - programas seqüenciais podem ser executados por apenas um processador (a não ser que o compilador os tenha transformado em concorrentes). Um programa concorrente é capaz de explorar um paralelismo real e conseguir uma execução mais rápida.

Modelar o paralelismo do mundo real - sistemas embarcados e em tempo real precisam controlar e interagir com entidades do mundo real (por exemplo, robôs, braços mecânicos, etc.) que são inerentemente paralelos. Refletir a natureza paralela do sistema na estrutura do programa o torna mais legível, de manutenção mais fácil e mais confiável.

Há uma grande dificuldade para programadores desenvolverem sistemas concorrentes. As principais causas estão relacionadas às diferenças entre os sistemas seqüenciais (aos quais estão

acostumados) e concorrentes (ver Tabela 2.1). Outro fator importante é como as linguagens suportam a programação concorrente. Linguagens mais antigas, como C, Fortran e Pascal, precisam do suporte do sistema operacional para usar concorrência; sendo C normalmente associado com UNIX e POSIX[1]. No entanto, as linguagens mais modernas, como C#, Ada e Java, já possuem suporte direto a programação concorrente. Quando a concorrência é suportada diretamente pela linguagem, torna-se mais fácil usá-la, pois não há muita preocupação com detalhes específicos da plataforma.

Programas Seqüenciais	**Programas Concorrentes**
Podem usar apenas um processador	Podem aproveitar mais de um processador
Comportamento normalmente determinístico	Comportamento normalmente não determinístico
Testes que cobrem todo o código encontram a maioria dos erros	Cobertura de código é insuficiente; threads competindo pelo acesso à memória causam os erros mais problemáticos
As causas das falhas são encontradas pelo cuidadoso acompanhamento da execução (*tracing*)	Causas das falhas são encontradas pela suposição de uma execução que se adeqüe ao comportamento observado
Uma vez identificado, o correção de um erro é geralmente garantida	A causa do erro pode facilmente permanecer não identificada

Tabela 2.1. Características de programas seqüenciais e concorrentes

A execução concorrente de um sistema normalmente é realizada usando-se os conceitos de threads e processos. Tanto threads quanto processos são caracterizados a partir de linhas de execução independentes (são executadas em "paralelo"[2], podendo haver interação entre elas). A diferença entre threads e processos reside no fato de processos serem tipicamente independentes, carregarem uma quantidade considerável de informação sobre seu estado, terem endereços de memória separados e interagirem apenas através de APIs [3] para comunicação inter-processos providas pelo sistema operacional. Por outro lado, Threads compartilham as informações de estado de um único processo, bem como memória e outros recursos diretamente.

2.1.1 Comunicação entre processos/threads

A comunicação entre os processos/threads[4] pode ser realizada através de memória compartilhada ou troca de mensagens. No caso do uso de memória compartilhada, um thread escreve na área compartilhada enquanto o outro a lê. No outro caso, os threads enviam mensagens para os demais, que as recebem.

[1] Conjunto de padrões para a programação em sistemas UNIX - inclusive padrões para concorrência.
[2] paralelismo real ou virtual, de acordo com o hardware subjacente.
[3] *Application Programming Interface* - conjunto de serviços providos por algum componente de software.
[4] Até o final da presente seção os termos thread e processos serão utilizados como sinônimos.

O modelo de memória compartilhada não distingue a memória para uso local (como a maioria das variáveis locais) da memória que está sendo usada para realizar comunicação entre threads (como variáveis globais e memória heap). Como a memória que é potencialmente compartilhada precisa ser tratada com mais cuidado do que memória local a um thread, é bem fácil cometer algum erro.

Já no modelo de troca de mensagens, não há compartilhamento de memória. Toda a comunicação entre os diferentes threads é realizada via troca de mensagens. Este modelo possui a vantagem de evitar acesso concorrente à memória, mas apresenta um desempenho inferior ao compartilhamento de memória. O modelo de troca de mensagens é bem popular em sistemas distribuídos, onde cada processo executa em uma máquina diferente, cada qual com sua memória.

Outro conceito importante é o de comunicação síncrona e assíncrona. Em um modelo de comunicação síncrona, ao enviar uma mensagem, o emissor fica bloqueado até receber a resposta. Já no modelo assíncrono, as mensagens são enviadas e o emissor continua sua execução normalmente.

2.1.2 Dificuldades

Uma das maiores críticas à programação concorrente é que ela introduz *overhead*, resultando em uma execução mais lenta quando o programa é executado em apenas um processador. Este *overhead* é causado principalmente pela troca de contextos realizada quando um thread é pausado e outro inicia sua execução (nos casos onde o número de processadores é menor que o de threads). Apesar disso, questões relacionadas à Engenharia de Software (por exemplo, modularidade, manutenabilidade, legibilidade) superam esses problemas, do mesmo modo que questões de eficiência são superadas pelas vantagens de se programar em uma linguagem de alto nível em vez de Assembly.

A criação de programas concorrentes pode introduzir problemas inexistentes em versões seqüenciais correspondentes, isto advém, em parte, da necessidade de coordenação entre as atividades concorrentes, caso desejem cooperar para a solução de um problema. Esta coordenação pode envolver padrões intrincados de comunicação e sincronização. Caso não sejam gerenciados apropriadamente, eles podem adicionar uma complexidade significativa aos programas e resultar no acontecimento de novas condições de erro. É possível citar algumas como exemplos:

- Pode ocorrer deadlock[5] quando cada atividade concorrente está esperando outra para executar uma operação.
- Pode haver interferência quando duas ou mais atividades concorrentes tentam atualizar o mesmo objeto; podendo ocasionar a corrupção dos dados do objeto. Este problema é conhecido pelo nome de *race condition*.
- Pode haver estagnação (*starvation*) caso uma ou mais atividades concorrentes estiverem continuamente negando recursos como resultado das ações de outros.

[5]Situação onde dois threads estão esperando um pelo término do outro, fazendo com que nenhum dos dois termine.

O comportamento desejado de um programa concorrente é normalmente expresso com duas propriedades: segurança (*safety*) e *liveness*. A propriedade de segurança expressa o requisito de que "nada ruim acontece". Em outras palavras, as atividades concorrentes não interferem nas outras causando corrupção dos dados. A propriedade de *liveness* expressa o requisito de que "algo bom acontecerá". Em outras palavras, todas as atividades concorrentes são capazes de progredir com sua execução, sem o acontecimento de deadlock ou estagnação. Algumas considerações sobre programação concorrente podem ser observadas na Tabela 2.2.

Programas Seqüenciais	**Programas Concorrentes**
A memória é estável a não ser que seja explicitamente atualizada	A memória está em fluxo, exceto seja somente para leitura, local ao thread, ou protegida por um *lock*
Não há a necessidade de *locks*	*Locks* são essenciais
Invariantes das estruturas de dados precisam ser satisfeitas antes e depois da execução dos métodos que as manipulam	As invariantes das estruturas de dados precisam ser satisfeitas todo o tempo em que o *lock* para a estrutura de dados não estiver tomado
Não há considerações especiais quando se acessa duas estruturas de dados interdependentes	Se duas estruturas de dados são relacionadas, *locks* são necessários para se usar relacionamentos
Não há a possibilidade de *deadlock*	Há a possibilidade de *deadlock* quando são usados múltiplos *locks*

Tabela 2.2. Programação de sistemas seqüenciais e concorrentes

Deadlock é um dos problemas mais comuns em sistemas concorrentes. Há quatro condições necessárias para a ocorrência de deadlock, sendo elas usualmente expressas em termos de recursos alocados a um thread. As condições são as seguintes:

Exclusão mútua: apenas uma atividade concorrente pode usar um recurso por vez (ou seja, o recurso não é compartilhado ou ao menos possui um acesso concorrente limitado).

Adquirir e esperar: é necessária a existência de atividades concorrentes que estão usando recursos enquanto esperam pela aquisição de outros recursos. Usando-se *locks*, atividades concorrentes precisam manter *locks* enquanto esperam obter novos *locks*.

Ausência de preempção: um recurso pode apenas ser liberado voluntariamente por uma atividade concorrente; os *locks* adquiridos por uma atividade concorrente não podem ser forçadamente tomados por outra atividade.

Espera circular: uma cadeia circular de atividades concorrentes precisa existir de modo que cada atividade possua recursos (*locks* de outros objetos) que estão sendo requisitados pela próxima atividade da cadeia.

Não obstante, o problema de *deadlock* pode ser evitado. Há três diferentes abordagens para se lidar com deadlock:

Prevenção: o *deadlock* pode ser prevenido pela garantia de que pelo menos uma das condições necessárias a sua ocorrência não aconteça. Ao se usar recursos compartilháveis, por exemplo, nunca se deve manter um recurso enquanto se espera pela aquisição de outro, os recursos podem ser tornados preemptíveis, ou pode-se impor uma estrita ordem lógica à requisição de recursos, de modo que todos os threads requisitem recursos na mesma ordem.

Deadlock avoidance: caso se possua mais informações sobre o padrão de uso dos recursos, então é possível construir um algoritmo que permita que as quatro condições necessárias ao *deadlock* ocorram, mas garanta que o sistema nunca entrará em *deadlock*. Um algoritmo de *deadlock avoidance* examina dinamicamente o estado da alocação de recursos e toma as devidas ações para garantir que o sistema não entre em *deadlock*. Qualquer requisição de alocação de recursos potencialmente insegura é negada.

Detecção e recuperação: em muitos sistemas concorrentes de propósito geral, a alocação de recursos não é conhecida *a priori*. E mesmo que seja conhecida, o custo de *deadlock avoidance* é muitas vezes proibitivo. Conseqüentemente, muitos desses sistemas ignoram esse problema até que entrem em *deadlock*; eles realizam então ações corretivas para retornar à execução normal (por exemplo, abortar uma atividade concorrente e assegurar a preempção de seus recursos).

2.1.3 Métodos

Ao desenvolver um sistema concorrente, o desenvolvedor precisa decidir sobre quais processos serão utilizados, quantos usar, e como eles vão interagir. Estas decisões são influenciadas pela natureza da aplicação e pelo hardware subjacente onde ela será executada. Para quaisquer opções escolhidas, a chave para o desenvolvimento de um sistema *correto* é a garantia de que a interação entre os processos esteja devidamente sincronizada. Para tanto, foram desenvolvidos diversos métodos, dos quais citaremos alguns a seguir.

Locks

Um *lock* é um mecanismo para limitar o acesso a um recurso compartilhado em um ambiente onde há vários threads executando. Um *lock* pode ser visto como um *token* necessário para a realização de uma tarefa. Um thread só poderá executar determinada seção crítica quando estiver de posse do *lock* para tal seção. *Locks* são, portanto, considerados um dos modos mais simples de se implementar exclusão mútua.

Semáforos

Um semáforo é uma variável protegida (ou tipo de dados abstrato) e constitui um método clássico para se restringir o acesso a recursos compartilhados (por exemplo, armazenamento) em um ambiente multithreaded. Para obter acesso a um recurso compartilhado, o processo precisa acessar o semáforo para obter permissão; outro processo só obtém acesso ao recurso quando ele for liberado.

Semáforos ainda são comuns em linguagens que não suportam intrinsicamente outras formas de sincronização. Eles ainda são o mecanismo de sincronização em muitos dos sistemas operacionais em uso. A tendência, porém, é a adoção de formas de sincronização mais estruturadas, como monitores e canais. Além de sua inadeqüação ao lidar com *deadlock*, semáforos não protegem o programador de erros simples como obter um semáforo em poder do mesmo processo ou esquecer de liberar um semáforo obtido.

Monitores

Um monitor [Hoa74] é uma abordagem para sincronizar duas ou mais tarefas que usam um recurso compartilhado, usualmente um conjunto variáveis e/ou atributos.

Um monitor consiste basicamente em:

- Um conjunto de procedimentos que permitem a interação com os recursos compartilhados;
- Um *lock* de exclusão mútua;
- As variáveis associadas ao recurso;
- Um invariante do monitor que define as premissas necessárias para evitar *race conditions*.

Para obter acesso a um recurso protegido por um monitor, o thread precisa obter seu *lock* de exclusão mútua, liberando-o após o término da seção crítica.

Barreiras

Barreiras são um mecanismo de sincronização onde a execução é bloqueada até que um determinado conjunto de threads termine sua execução. Barreiras são bem comuns quando se possui um conjunto de tarefas que podem ser realizadas em paralelo, mas o resultado de cada uma é necessário para uma computação posterior.

2.2 PROGRAMAÇÃO CONCORRENTE EM JAVA

Java é um linguagem de programação recente. Ela foi concebida em 1991, originalmente projetada para produtos eletrônicos. No entanto, em 1994, quando teve seu foco redirecionado para a internet, ela começou a receber muita atenção da comunidade de desenvolvedores. Desde aquele tempo, a popularidade de Java vem aumentando, superando todas as expectativas. Assim como aconteceu com C, devido a popularidade de sistemas operacionais UNIX. Entre suas características, podemos citar: [GM96]

Simples, orientada a objetos e familiar - orientação a objetos ainda é o paradigma dominante; o modelo de Java é simples e apresenta muitas semelhanças com C++;

Robusta - verificações em tempo de compilação e execução asseguram a confiabilidade dos programas.

Distribuída e segura - uma linguagem e sistema de execução projetados para executar em ambientes distribuídos precisam incorporar aspectos de segurança;

Interpretada, arquitetura neutra e portável - o uso de uma máquina virtual que interpreta *byte codes* [6] independentes de arquitetura garante que os programas são portáveis de uma plataforma para a outra.

Alta performance - técnicas como compilação *just-in-time* permitem que códigos de performance críticas sejam compilados para código nativo, contrabalanceando os *overheads* associados à interpretação de byte codes;

Dinâmica - classes só são carregadas quando necessárias, e novas classes podem ser ligadas à medida que o sistema evolui.

Multithreaded - suporte da linguagem para programação concorrente permite que aplicações *multithreaded* portáveis sejam construídas.

2.2.1 Modelo de Concorrência

Atualmente, Java é um dos mais interessantes desenvolvimentos na área de programação concorrente e orientada a objetos [Wel04]. Como se tratava de uma linguagem nova, seus criadores foram capazes de projetar um modelo de concorrência dentro do *framework* de orientação a objetos sem se preocupar com questões de compatibilidade. Tal integração se deu através de uma adaptação do conceito de objetos ativos[7].

Em Java, objetos ativos podem ser obtidos de duas maneiras: herdando da classe `Thread` ou implementando a interface `Runnable`, como pode ser observado na Figura 2.1. Um thread é criado quando o objeto associado a ele é criado e quando o método `start` é chamado, o novo thread começa sua execução chamando o método `run`.

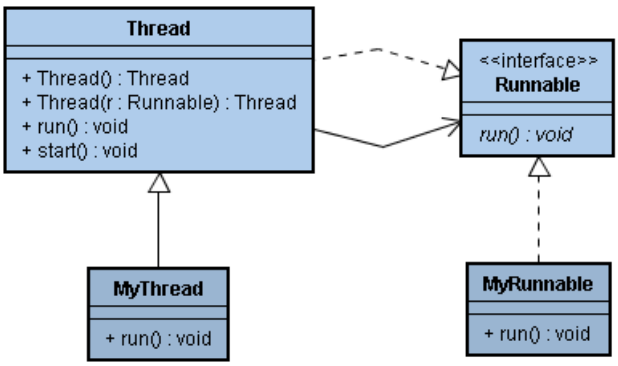

Figura 2.1. Threads em Java

[6] Código intermediário executado pela máquina virtual Java - sistema necessário para executar programas Java.

[7] Objetos com comportamento próprio, em oposição a objetos passivos - que apenas respondem requisições.

Criar subclasses de `Thread` e sobrescrever o método `run` permite que uma aplicação possua objetos ativos. Alternativamente, o método `run` pode ser passado ao thread no momento de sua criação, usando a interface `Runnable`. As duas abordagens para a criação de threads em Java serão discutidas a seguir.

Para criar um thread usando herança, primeiro é necessário criar uma classe que herde da classe pré-definida `Thread` e redefina o método `run`. O método `run` define o comportamento do thread; em seu corpo devem estar todos os comandos a serem executados, pois este método será chamado quando o thread for posto em execução pela máquina virtual. Após a criação da classe e redefinição do método `run`, cria-se um novo objeto da classe e ao chamar o método `start` o thread inicia sua execução - como pode ser visto na Figura 2.2.

```
public class OlaMundoThread extends Thread {
    public void run() {
        System.out.println("Olá Mundo!");
    }

    public static void main(String[] args) {
        OlaMundoThread olaMundo = new OlaMundoThread();
        olaMundo.start();
    }
}
```

Figura 2.2. Criação de Thread usando herança

Na outra abordagem para a criação de Threads, a classe que deverá ser executada concorrentemente implementa a interface `Runnable`. Esta interface possui apenas o método `run`, que deve ser implementado pela classe. Esta abordagem é a alternativa para os casos onde a classe já herda de outra. Como Java não permite herança múltipla, não é possível herdar de mais de uma classe. A Figura 2.3 mostra como o exemplo anterior seria feito nesta abordagem.

```
public class OlaMundoRunnable implements Runnable {
    public void run() {
        System.out.println("Olá Mundo!");
    }
    public static void main(String[] args) {
        OlaMundoRunnable olaMundo = new OlaMundoRunnable();
        Thread thread = new Thread(olaMundo);
        thread.start();
    }
}
```

Figura 2.3. Criação de Thread usando a interface Runnable

O ciclo de vida de um thread pode ser observado na Figura 2.4. Um thread, ao ser criado, fica em um estado inerte até que o método `start` seja chamado, momento no qual o thread entra no estado *running*. No estado *running*, o thread pode ceder a execução a outro com o uso do método `yield`, mantendo-se no mesmo estado; ou pode ir para o estado *not runnable* quando o método `wait` ou `sleep` são executados. Quando a execução do método `run` termina, o thread entra no estado *dead*.

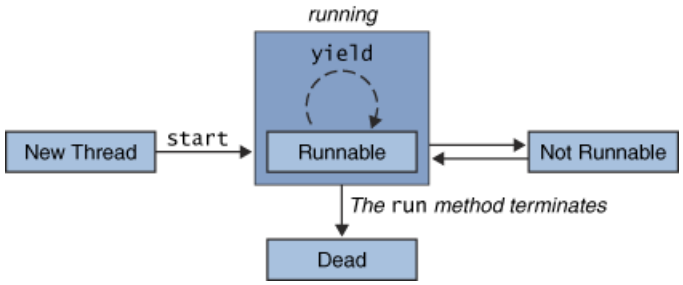

Figura 2.4. Ciclo de vida de um thread

A classe Thread apresenta métodos bem úteis para controlar sua execução, dentre os quais podemos citar os seguintes:

start() inicia a execução do thread como uma atividade independente, chamando o método `run` definido pelo programador. Quando o método `run` termina sua execução, o thread é finalizado.

stop() pára a execução do thread. Este método está depreciado porque é propício a *deadlock* [Mic99].

isAlive() retorna *true* caso o thread tenha iniciado sua execução, mas ainda não terminado.

sleep() suspende a execução do thread por um tempo determinado.

join() espera até o término da execução do thread. Este método possui uma variação onde se estabelece um *timeout* para que não se espere por um tempo indefinido.

yield() suspende a execução do thread, permitindo que outros threads executem ("cede a vez").

2.2.2 Controle de Concorrência

A comunicação entre threads é realizada pela leitura e escrita de objetos compartilhados. Como visto anteriormente, esses objetos precisam ser protegidos de atualizações simultâneas, de modo a evitar interferências e subseqüentes inconsistências. Em Java, toda classe herda implicitamente da classe `Object`, a qual define um `lock` de exclusão mútua. Conseqüentemente, todo objeto criado possui um `lock` em potencial (eles são criados apenas quando necessário). O controle de concorrência pode ser realizado com o uso de diversos recursos da linguagem, descrevemos a seguir o principais mecanismos existentes em Java.

Sincronização

É o principal recurso de controle de concorrência presente na linguagem. Associado a cada objeto há um *lock* de exclusão mútua; o acréscimo de sincronização evita a execução concorrente com o uso desse *lock*. Há duas maneiras de se usar sincronização em Java: usando o modificador `synchronized` na declaração do método e criando-se blocos sincronizados.

Quando um método possui o modificador `synchronized` na sua declaração, o acesso ao método só pode ser realizado quando o *lock* associado ao objeto for obtido. Sendo assim, métodos sincronizados proporcionam acesso mutuamente exclusivo aos dados encapsulados pelo objeto, caso eles sejam manipulados apenas por métodos sincronizados. Métodos não sincronizados não requerem *locks*, e podem portanto serem chamado a qualquer momento. De tal modo, para prover uma exclusão mútua completa, todos os métodos que acessem dados encapsulados devem ser sincronizados (quando chamados por múltiplos threads). O uso do modificador `synchronized` pode ser observado na figura 2.5

```
public synchronized void write(int newValue) {
    theData = newValue;
}
```

Figura 2.5. Exemplo de método sincronizado

Blocos sincronizados provêem um mecanismo onde um bloco de código pode ser marcado como sincronizado. A palavra-chave `synchronized` usa como parâmetro um objeto cujo *lock* precisa ser obtido antes de se continuar a execução. Usando-se blocos sincronizados, o método mostrado anteriormente (Figura 2.5) ficaria como mostrado na Figura 2.6.

```
public void write(int newValue) {
    synchronized(this) {
        theData = newValue;
    }
}
```

Figura 2.6. Exemplo de bloco sincronizado

Java provê outro recurso para o controle de acesso a variáveis compartilhadas: o modificador `volatile`. De acordo com a especificação de Java [GJSB05], um atributo volatile não é mantido na memória local e todas as leituras e escritas são feitas diretamente na memória principal. Além disso, as operações realizadas em atributos voláteis devem ser realizadas exatamente na ordem que um thread requisitar. Outra regra define ainda que atributos voláteis dos tipos double e long são lidos e escritos atomicamente[8].

[8] A especificação de Java permite que estes tipos, por possuírem 64 bits, sejam lidos e escritos de forma não atômica. É recomendado porém, que os implementadores de máquinas virtuais o façam de modo atômico.

Ainda há métodos para realizar interação entre threads, usados principalmente para sincronização condicional - algo bem comum em programação concorrente. Os métodos mostrados a seguir estão definidos na classe `Object` - a superclasse padrão de todas as classes Java.

wait() faz com que o thread fique esperando pelo monitor do objeto. O monitor é liberado através dos métodos de notificação.

notify() notifica um thread que esteja esperando pelo monitor do objeto, para que ele retorne à execução.

notifyAll() similar ao anterior, só que a notificação é feita a todos os threads que estão esperando pelo monitor.

O mecanismo de *wait/notify* permite que um thread espere por uma notificação de outro para prosseguir. Tipicamente, o primeiro thread checa uma variável e verifica que o valor ainda não é o necessário. O primeiro thread então chama o método `wait()` e entra em um estado ocioso (sem usar praticamente nenhum recurso do processador) até que seja notificado que algo foi mudado. Eventualmente, um segundo thread modifica o valor da variável e invoca `notify()` (ou `notifyAll()`) para sinalizar ao thread ocioso que a variável foi alterada.

O mecanismo de *wait/notify* não necessita que a variável seja checada por um thread e modificado por outro. Contudo, é geralmente uma boa idéia usar esse mecanismo em conjunção com apenas uma variável. Deste modo diminui a possibilidade de perderem-se notificações.

2.2.3 Java Concurrency Utilities

O pacote Java *Concurrency Utilities* [JSR04][Mah05] foi introduzido na versão 5.0 do J2SDK[9]. Este pacote possui classe projetadas para serem reutilizadas na construção de classes e sistemas concorrentes. Assim como o *framework* de coleções simplificou a organização e manipulação de dados em memória, provendo implementações de estruturas de dados freqüentemente usadas, o pacote de utilitários para concorrência tem o objetivo de simplificar o desenvolvimento de classes concorrentes por meio de implementações de recursos comumente usados em projetos concorrentes. O pacote inclui um *pool* de threads flexível e de alta performance; um *framework* para execução assíncrona de tarefas; diversas classes de coleções otimizadas para acesso concorrente; utilidades de sincronização como semáforos com contadores; variáveis atômicas; *locks*; e variáveis condicionais.

O uso do pacote de utilitários para concorrência, em vez do desenvolvimento de seus próprios componentes, oferece as seguintes vantagens:

- Esforço de programação reduzido. É bem mais fácil usar uma classe pré-definida que desenvolver uma própria.
- Aumento de performance. A implementação do pacote de utilitários para concorrência foi desenvolvida e revisada por especialistas em concorrência e performance; essas implementações tendem a ser mais rápidas e escaláveis que implementações típicas.

[9] Java 2 Standard Development Kit - conjunto de ferramentas para o desenvolvimento de programas Java

- Aumento de confiabilidade. O desenvolvimento de classes concorrentes é difícil - as primitivas de baixo nível providas pela linguagem (`synchronized, volatile, wait(), notify(), and notifyAll()`) são difíceis de usar corretamente, e erros ao usar esses recursos podem ser difíceis de detectar e corrigir. Ao se usar componentes para concorrência padronizados e exaustivamente testados, muitas fontes de problemas como *deadlock*, estagnação, *race conditions* e excessiva mudança de contexto são eliminadas. O pacote de utilitários para concorrência foi cuidadosamente checado em relação a *deadlock*, estagnação e *race conditions*.

- Melhor manutenabilidade. Programas que usam classes da biblioteca padrão são mais fáceis de entender e manter do que os baseados em classes criadas pelo próprio programador.

- Aumento de produtividade. Desenvolvedores costumam já entender a biblioteca de classes padrão, então não é necessário aprender a API e comportamento de componentes de concorrência *ad-hoc*. Além disso, aplicações concorrente são bem mais fáceis de corrigir quando construídas com o uso de componentes confiáveis e bem testados.

O conjunto de utilitários para concorrência inclui:

Framework para agendamento de tarefas - O *framework* Executor se destina a padronizar a invocação, *scheduling*, execução e controle de tarefas assíncronas de acordo com um conjunto de políticas de execução. Implementações podem permitir que tarefas sejam executadas pelo thread que as submeteu, em um único thread em *background* (como acontece com os eventos em Swing[10]), em um thread recém criado ou em um *pool* de threads, permitindo ainda que os desenvolvedores definam suas próprias políticas de execução. As implementações padrão oferecem políticas configuráveis como filas de tamanho limitado e políticas de saturação - que podem melhorar a estabilidade de aplicações ao prevenir o consumo de recursos excessivos.

Coleções Concorrentes - diversas novas classes de coleções foram adicionadas, inclusive as novas interfaces `Queue` e `BlockingQueue`; e implementações concorrentes de alta performance das interfaces `Map`, `List` e `Queue`.

Variáveis Atômicas - classes para a manipulação atômica de variáveis (tipos primitivos ou referências), provendo um conjunto de métodos de alta performance para operações aritméticas, de comparação e atribuição. A implementação de variáveis atômicas em `java.util.concurrent.atomic` proporciona melhor performance que a disponível quando se usa sincronização (na maioria das plataformas), tornando-a útil para a implementação de algoritmos concorrentes de alta performance, assim como implementar convenientemente contadores e geradores de números seqüenciais.

Sincronizadores - classes para sincronização de propósito geral, incluindo semáforos, *locks* de exclusão mútua, barreiras, *latches* e *exchangers*, os quais facilitam a coordenação entre threads.

[10] Biblioteca para a construção de interfaces gráficas em Java

Locks - apesar de *locks* já serem parte de Java, há uma série de limitações aos *locks* monitores padrão. O pacote `java.util.concurrent.locks` provê uma implementação de alta performance de *locks*, com a mesma semântica de memória da sincronização, com o adicional de possibilitar a especificação de um *timeout* quando se tenta obter um *lock*, múltiplas variáveis condicionais por *lock*, *locks* fora de escopo léxico e suporte para a interrupção de threads que estão esperando para a aquisição de *locks*.

Temporização com precisão de nanosegundos - O método `System.nanoTime` permite acesso a uma fonte de tempo com precisão de nanosegundos, com o objetivo de realizar medidas de tempo. Métodos que aceitam *timeouts* (como `BlockingQueue.offer`, `BlockingQueue.poll`, `Lock.tryLock`, `Condition.await`, e `Thread.sleep`) podem obter valores em nanosegundos. A precisão atual de `System.nanoTime` é dependente de plataforma.

CAPÍTULO 3

REGRAS DE TRANSFORMAÇÃO

Este capítulo apresenta um conjunto de regras que transforma um programa Java seqüencial em um explorando concorrência. Cada regra é constituída por dois trechos de código Java. O esquerdo se refere ao código antes da aplicação da regra e o direito seu resultado. Além disso, algumas regras podem necessitar de certas restrições para a sua aplicação ser bem sucedida. Aspectos relacionados com a preservação da semântica após a aplicação de uma regra são discutidos informalmente, baseando-se principalmente na especificação de Java [GJSB05].

Vale salientar que os códigos Java usados nas regras podem conter algumas simplificações, de modo a possibilitar a abstração de detalhes irrelevantes às transformações em questão.

3.1 REGRAS PARA CONTROLE DE CONCORRÊNCIA

A regra a seguir introduz o modificador `synchronized` na declaração de um método. Ela deve ser aplicada quando for necessário que toda a execução do método seja atômica, de modo mutuamente exclusivo.

Regra 1 ⟨Introduzir sincronização na definição do método⟩

```
m() {
    stmtList;
}
```

→

```
synchronized m() {
    stmtList;
}
```

□

A regra anterior é suficiente para garantir a não introdução de problemas relacionados ao acesso de dados compartilhados. Entretanto, não há ganho qualquer de performance, pois todos os métodos se tornarão atômicos quando na realidade somente os trechos de código que manipulam objetos compartilhados é que precisam ser atômicos.

Deste modo, a Regra 2 é necessária para garantir uma execução concorrente segura, movendo a sincronização para o interior do método, fazendo uso de um bloco sincronizado. Ela deve ser aplicada quando apenas um trecho do método precisar ser sincronizado. Isto acontece, por exemplo, nos casos onde apenas uma parte dos comandos do método manipulam atributos acessados por mais de um thread. Esta é a maior diferença em relação à Regra 2, que garante execução atômica de todo o corpo do método, usando o lock do objeto possuidor do método ou da classe (no caso de métodos estáticos).

O uso da Regra 2 permite um maior controle sobre a granularidade da sincronização, pois é possível sincronizar apenas um trecho do corpo do método. Além disso, há a possibilidade de sincronizar outros objetos, passando-os como parâmetro do bloco.

Regra 2 ⟨Introduzir bloco sincronizado⟩

```
m() {
    stmtList1;
    stmtList2;
}
```

→

```
m() {
    stmtList1;
    synchronized(obj) {
        stmtList2;
    }
}
```

desde que
(→) `stmtList1` não acesse atributos compartilhados
(→) `stmtList2` acesse atributos compartilhados

□

O acréscimo do modificador `volatile` em um atributo faz com que as leituras e escritas de seu valor sejam atômicas. Este recurso é bastante útil quando os atributos são dos tipos long ou double, pois a leitura/escrita de atributos desses tipos podem não ser atômicas. O uso do modificador `volatile` em tipos não-primitivos afeta apenas a referência para o objeto, sendo assim de pouco uso para o controle de concorrência.

Regra 3 ⟨Inserir modificador volatile⟩

```
type att;
```

→

```
volatile type att;
```

desde que
(→) type seja um tipo primitivo

□

Um resultado equivalente ao anterior pode ser obtido encapsulando o atributo como privado e usando o modificador synchronized nos métodos de acesso e modificação do atributo (*get* e *set*). Usar atributos com o modificador `volatile` é considerado mais eficiente que usar sincronização [Lea99, Blo01]; na pior da hipóteses, não é mais custoso. Apesar disso, no caso onde atributos voláteis são constantemente acessados dentro do corpo de métodos, a performance pode ser pior do que seria caso o método fosse sincronizado, pois será necessário obter constantemente o *lock* para o atributo. A declaração de atributos voláteis é especialmente útil quando se deseja que seus valores possam ser acessados corretamente por múltiplos threads, não sendo necessário usar o lock por outras razões. Isto pode ocorrer nos seguintes casos:

- O atributo não precisa obedecer invariantes em respeito a outros.
- Escritas no atributo não dependem de seu valor atual.
- Nenhum thread escreve um valor ilegal de acordo com a semântica pretendida.
- As ações dos leitores não dependem dos valores de outros atributos não-voláteis.

O uso de atributos voláteis faz sentido quando se sabe que apenas um thread pode modificar o valor do atributo, mas diversos outros podem acessar seu valor.

As três regras apresentadas anteriormente devem ser aplicadas com cuidado, e só nos casos onde for realmente necessário. Isto se deve ao fato de o acréscimo de sincronização piorar o desempenho do programa, pois o compilador incluirá código para a obtenção e liberação do lock em cada em trecho que for sincronizado.

Caso um atributo seja volátil, sua leitura e escrita são garantidamente atômicas. Mesmo assim há casos onde métodos de acesso e atualização do atributo são sincronizados. As duas regras a seguir têm o propósito de remover essa sincronização desnecessária, melhorando assim o desempenho dos métodos.

Regra 4 ⟨Remover synchronized de métodos de acesso a atributos voláteis⟩

```
synchronized getAtt() {
    return this.att;
}
```
→
```
getAtt() {
    return this.att;
}
```

desde que
(→) att seja declarando como `volatile`
(→) att seja de um tipo primitivo

□

Regra 5 ⟨Remover synchronized de métodos de atualização de atributos voláteis⟩

```
synchronized setAtt(param) {
    this.att = param;
}
```
→
```
setAtt(param) {
    this.att = param;
}
```

desde que
(→) att seja declarando como `volatile`
(→) att seja de um tipo primitivo

□

A classe `ArrayList` é uma das mais utilizadas quando há necessidade de um vetor dinâmico. Ela funciona perfeitamente em uma execução seqüencial, mas quando posta num ambiente concorrente, não há garantia que ela manterá um estado consistente. Uma maneira bem simples de resolver este problema é trocar o `ArrayList` por um `Vector`, que é sincronizado. A substituição é possível porque todos os métodos oferecidos pelo `ArrayList` também são oferecidos por `Vector` - o contrário já não é possível, pois `Vector` possui mais métodos que `ArrayList`. Esta opção é mais interessante do que realizar a sincronização por conta própria, pois as classes da API Java têm uma garantia maior quanto ao seu correto funcionamento. A regra a seguir faz uso do recurso de *generics*[1] de Java 1.5; caso se deseje aplicar a regra em versões anteriores, deve-se apenas desconsiderar a definição do tipo da coleção.

[1] Classes parametrizadas.

Regra 6 ⟨Modificar tipo da coleção de ArrayList para Vector⟩

```
ArrayList<Type> att;   =   Vector<Type> att;
```

desde que
(←) apenas sejam usados os métodos que `ArrayList` possui
(←) não haja acesso concorrente ao `Vector`

□

3.2 REGRAS PARA INSERÇÃO DE CONCORRÊNCIA

A regra a seguir cria uma versão concorrente para um método existente na classe. O método criado simplesmente inicia a execução de um novo thread, inserindo no seu método `run()` uma chamada ao método que se deseja executar concorrentemente.

Regra 7 ⟨Criar versão concorrente de método⟩

```
void m() {
  stmts;
}
```

→

```
void m() {
  stmts;
}

void run_m() {
  new Thread() {
    public void run() {
      m();
    }
  }.start();
}
```

desde que
(→) a classe não possua um método com a mesma assinatura de `run_m()`
(→) `m` seja devidamente sincronizado caso manipule dados compartilhados

□

É importante notar que o método `m()` é void, ou seja, não retorna nenhum valor como resultado de sua execução. Isto se faz necessário porque a execução do thread é assíncrona, mesmo que o método `m()` não tenha terminado, o método `run_m` é finalizado.

Para que realmente haja uma execução concorrente, é necessário substituir as chamadas a `m()` por chamadas a `run_m()`. A regra a seguir ilustra essa transformação:

Regra 8 ⟨Substituir chamadas seqüenciais por versão concorrente⟩

```
m1() {
 ...
    obj.m();
 ...
}
```

→

```
m1() {
 ...
    obj.run_m();
 ...
}
```

desde que
(→) `obj` possua o método `run_m()`
(→) todos os métodos que acessem variáveis usadas por m sejam sincronizados

□

Essa regra deve ser aplicada com muito cuidado. Antes de aplicá-la, é necessário realizar todo o controle de concorrência relacionado ao acesso a variáveis compartilhadas. Para tanto, podem ser usadas as regras apresentadas na seção anterior.

A transformação apresentada se mostra muito útil nos casos onde o sistema realiza alguma atividade que demore bastante, mas se deseja que o sistema continue respondendo a outras requisições. Ela é ainda mais eficiente quando os métodos que tiverem a execução concorrente trabalhem em conjuntos de variáveis disjuntos[2]. Nos casos onde há variáveis compartilhadas é importante verificar qual o grau de sincronização utilizado, caso se use muita sincronização de baixa granularidade (por exemplo, modificador `synchronized` na definição do método) há o risco de se perder a execução concorrente.

[2]Também conhecido como paralelismo vetorial.

CAPÍTULO 4

ESTUDO DE CASO

O estudo de caso será um sistema bancário bem simples, composto pelas classes `Banco`, `Conta`, `ContaCorrente` e `Poupanca`[1]. Os serviços oferecidos são basicamente a adição e remoção de contas e poupanças, operações de crédito e débito, a atualização dos saldos das poupanças de acordo com os rendimentos e o débito da taxa de manutenção das contas. O diagrama de classes do sistema pode ser observado na Figura 4.1.

A princípio será apresentada uma versão do sistema puramente seqüencial, sem nenhuma preocupação com controle de concorrência. A partir deste sistema, as regras serão gradativamente aplicadas, adeqüando-o a uma execução concorrente. As seções a seguir mostram os casos onde as Regras foram aplicadas.

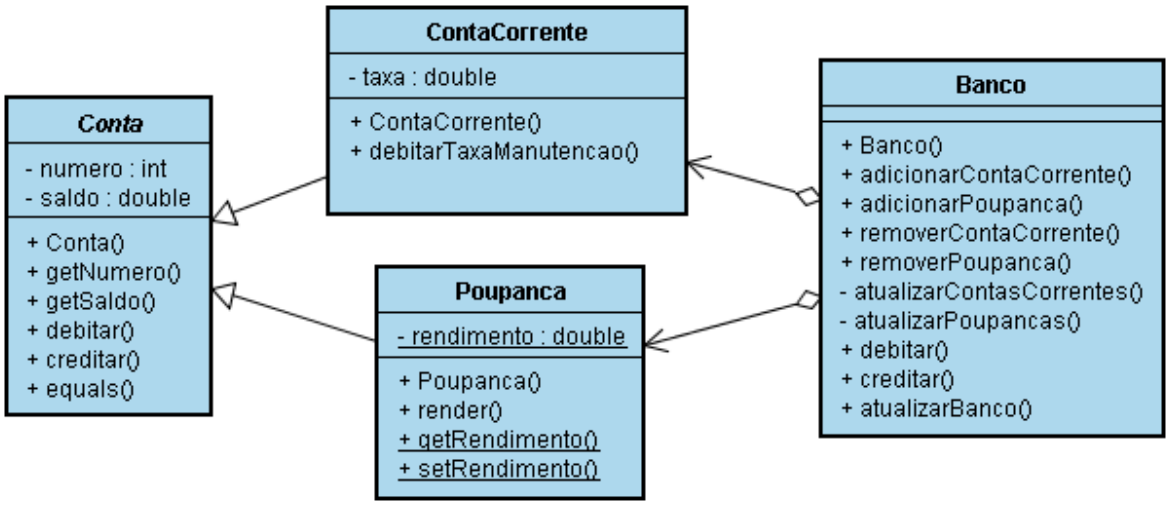

Figura 4.1. Diagrama de classes do sistema

4.1 CONTROLE DE CONCORRÊNCIA

Quando as classes do sistema eram usadas em uma execução seqüencial, hão havia nenhuma preocupação com o controle de concorrência. A partir do momento que se assume que as classes poderão ser acessadas por múltiplos threads simultaneamente, é essencial realizar um controle de concorrência eficaz.

[1] O código fonte do estudo foi bastante simplificado, aspectos como tratamento de exceções e verificação de parâmetros não foram contemplados apenas para facilitar sua apresentação.

A classe `Banco` possui dois atributos que representam as coleções de contas e poupanças; os tipos de ambos estão definidos como `ArrayList`. Além disso, há as operações de adição e remoção definidas para cada coleção.

```
public class Banco {
  ...
    private ArrayList<ContaCorrente> contasCorrentes;
    private ArrayList<Poupanca> poupancas;

    public void adicionarConta(Conta c) {
        this.contas.add(c);
    }
    public void adicionarPoupanca(Poupanca p) {
        this.poupancas.add(p);
    }
    public void removerConta(Conta c) {
        this.contas.remove(c);
    }
    public void removerPoupanca(Poupanca p) {
        this.poupancas.remove(p);
    }
  ...
}
```

Enquanto o paradigma seqüencial for usado, não há problema algum com o código acima. Entretanto, caso haja acesso concorrente aos atributos, não há garantia quanto à consistência da coleção (adições e remoções concorrentes podem corrompê-la). Uma solução bastante simples para este problema seria a aplicação da Regra 6.

```
public class Banco {
  ...
    private Vector<ContaCorrente> contasCorrentes;
    private Vector<Poupanca> poupancas;
  ...
}
```

Agora, por conta da classe `Vector`, o controle de concorrência da coleção é garantido, dispensando ainda o uso de sincronização nos métodos adicionar e remover definidos na classe `Banco`.

Outro problema que pode ocorrer na classe `Banco` é a execução concorrente dos métodos creditar e debitar. Uma operação de crédito em uma conta não pode acontecer ao mesmo tempo que uma operação de débito - elas devem ocorrer atomicamente.

```
public class Banco {
  ...
   public void debitar(int numeroConta, double valor) {
        for(int i = 0; i < this.contasCorrentes.size(); i++) {
            if(this.contasCorrentes.get(i).getNumero() == numeroConta) {
                this.contasCorrentes.get(i).debitar(valor);
            }
        }
    }
    public void creditar(int numeroConta, double valor) {
        for(int i = 0; i < this.contasCorrentes.size(); i++) {
            if(this.contasCorrentes.get(i).getNumero() == numeroConta) {
                this.contasCorrentes.get(i).creditar(valor);
            }
        }
    }
  ...
}

public abstract class Conta {
    private int numero;
    private double saldo;

    public void debitar(double valor) {
        this.saldo = this.saldo - valor;
    }
    public void creditar(double valor) {
        this.saldo = this.saldo + valor;
    }
  ...
}
```

A solução mais simples é aplicar a Regra 1 ou 2 nos métodos creditar e debitar da classe `Banco`, mas não seria uma solução muito eficiente, pois não seria possível executar os métodos concorrentemente em contas diferentes. Dessa forma, simplesmente aplicamos a Regra 1 nos métodos de crédito e débito da classe `ContaCorrente`, pois o problema acontece quando há acesso concorrente a uma mesma conta.

```
    public synchronized void debitar(double valor) {
        this.saldo = this.saldo - valor;
    }
    public synchronized void creditar(double valor) {
        this.saldo = this.saldo + valor;
    }
```

Na classe Poupanca, é possível observar mais um problema de controle de concorrência: o acesso ao atributo rendimento. Ele pode ser acessado concorrentemente via três métodos: render(), getRendimento(), setRendimento() e durante a execução de todos se deseja que o valor do rendimento esteja consistente.

```
public class Poupanca extends Conta {
    private static double rendimento;

    public void render() {
        this.creditar(Poupanca.rendimento);
    }
    public static double getRendimento() {
        return rendimento;
    }
    public static void setRendimento(double rendimento) {
        Poupanca.rendimento = rendimento;
    }
  ...
}
```

Na tentativa de resolver este problema, pode-se aplicar a Regra 3, que adiciona o modificador volatile à definição do atributo. Assim, há a garantia de que os threads sempre acessarão valores consistentes do rendimento. Porém, esta regra não garante a execução correta do método render(), já que ele não se trata apenas de uma leitura ou escrita de atributo. Mas ao observar bem o método, percebe-se que ele consiste basicamente a uma chamada ao método creditar, que é sincronizado. Acrescentar sincronização ao método render traria apenas um *overhead* desnecessário.

Outro caso onde é necessário realizar um controle de concorrência é nos métodos de atualização de contas correntes e poupanças.

```
  ...
    private void atualizarContasCorrentes() {
        Iterator<ContaCorrente> contas = this.contasCorrentes.iterator();
        while (contas.hasNext()) {
            contas.next().debitarTaxaManutencao();
        }
    }
    private void atualizarPoupancas() {
        Iterator<Poupanca> poupancas = this.poupancas.iterator();
        while (poupancas.hasNext()) {
            poupancas.next().render();
        }
    }
  ...
```

Para realizar a sincronização neste caso, deve-se aplicar a Regra 2 com acesso mutuamente exclusivo ao atributo que cada método acessa. Desta forma, assegura-se que não haverá acesso às contas e poupanças enquanto elas estiverem sendo atualizadas. Há ainda a necessidade de assegurar que todas as poupanças serão atualizadas com o mesmo rendimento. Assim sendo, deve-se evitar mudanças no valor do atributo rendimento enquanto as poupanças estiverem sendo atualizadas, algo que pode ser alcançado com o uso da Regra 2 com exclusão mútua na classe Poupanca.

```
...
  private void atualizarContasCorrentes() {
      synchronized (this.contasCorrentes) {
          Iterator<ContaCorrente> contas = this.contasCorrentes.iterator();
          while (contas.hasNext()) {
              contas.next().debitarTaxaManutencao();
          }
      }
  }
  private void atualizarPoupancas() {
      synchronized (this.poupancas) {
          Iterator<Poupanca> poupancas = this.poupancas.iterator();
          synchronized(Poupanca.class) {
              while (poupancas.hasNext()) {
                  poupancas.next().render();
              }
          }
      }
  }
...
```

4.2 INSERÇÃO DE CONCORRÊNCIA

Periodicamente, o banco realiza a atualização dos saldos das poupanças e contas. As contas têm debitado o valor da taxa de manutenção e as poupanças têm creditado o seu rendimento. Essas operações são independentes, pois não há nenhuma variável compartilhada por ambas durante sua execução

```
...
  public void atualizarBanco() {
      atualizarContas();
      atualizarPoupancas();
  }
...
```

É possível portanto que as duas operações sejam executadas concorrentemente sem problemas. Para tanto, serão aplicadas as Regras 7 e 8, resultando no código a seguir:

```
private void run_atualizarContas() {
    new Thread() {
        public void run() {
            atualizarContas();
        }
    }.start();
}

private void run_atualizarPoupancas() {
    new Thread() {
        public void run() {
            atualizarPoupancas();
        }
    }.start();
}

private void atualizarBanco() {
    run_atualizarContas();
    run_atualizarPoupancas();
}
```

Após a transformação, cada atualização é executada por um thread diferente. Esta abordagem é muito interessante quando se tem métodos que costumam demorar bastante, pois ao executá-los concorrentemente é possível continuar a execução sem que seja necessário esperar seu término.

A regra pôde ser aplicada facilmente porque os métodos trabalham em conjuntos de dados distintos, o que proporciona um alto grau de paralelismo entre as atividades. A transformação também pode ser aplicada quando há variáveis compartilhadas, mas faz-se necessário bastante cuidado com a sincronização dos dados comums aos threads. Outro ponto importante é o grau de sincronização dos métodos; executar concorrentemente dois métodos totalmente sincronizados no mesmo objeto não traz benefício algum, pois a execução será seqüencial e ainda haverá o *overhead* causado pela sincronização.

CAPÍTULO 5

CONCLUSÕES

O presente trabalho apresentou um conjunto de regras para transformar um programa Java seqüencial em concorrente. Algumas regras surgiram da teoria acerca de concorrência, enquanto outras de características próprias de Java. As regras propostas são essencialmente sintáticas, sem necessidade de lidar-se com informações dinâmicas e semânticas do programa. Outro ponto interessante é que o código resultante da aplicação das regras se mantém bem parecido com o original, facilitando assim sua compreensão.

As regras propostas foram então aplicadas a um estudo de caso, com o objetivo de validá-las. Foi constatado que, após o uso das regras, o sistema continuou funcionando como esperado, apesar de não podermos garantir formalmente que seu significado permaneceu inalterado. O estudo de caso foi bastante simples, mas já tornou possível sentir o potencial do uso da presente abordagem.

5.1 TRABALHOS RELACIONADOS

Podemos citar como abordagens relacionadas para o desenvolvimento de sistemas concorrentes em Java, UML-RT [Lyo98, SR98] e JCSP [WAF02]. A primeira se trata de uma extensão de UML para sistemas concorrentes e em tempo real; ela provê diversos recursos para a modelagem de concorrência, tendo um suporte ferramental capaz de gerar código Java a partir do modelo. A segunda provê uma biblioteca de classes Java baseada no modelo de concorrência de CSP [Ros97], tendo como objetivo facilitar o mapeamento de uma especificação formal em CSP para um sistema concorrente em Java.

Percebe-se que a abordagem apresentada difere das demais por se basear em transformações, pois consideramos que já existe um programa Java seqüencial e se deseja torná-lo concorrente - seja para obter um melhor desempenho, seja para fazê-lo mais legível e intuitivo. Outro ponto importante é que as abordagens relacionadas se baseiam em um modelo de troca de mensagens, enquanto o presente trabalho se baseia no modelo de memória compartilhada. Isto é bastante interessante devido ao fato da maioria dos desenvolvedores estar habituada ao modelo de compartilhamento de memória.

Outro ponto importante é que trabalhamos sempre com a biblioteca padrão da linguagem. As outras abordagem se baseiam em bibliotecas proprietárias desenvolvidas por terceiros e, portanto, nem sempre gratuitas e de fácil obtenção.

5.2 TRABALHOS FUTUROS

Ainda há muito a ser feito para que as regras possam ser usadas eficientemente em um real ambiente de desenvolvimento; a seguir pode ser observada uma lista de alguns dos trabalhos que poderiam ser realizados:

Criação de novas regras. É interessante que sejam criadas novas regras que capturem outros aspectos de concorrência, criando-se assim um conjunto de regras mais completo. Há também um forte interesse na criação de regras que aproveitem as classes da nova biblioteca para concorrência do J2SDK 5.0

Estudos de caso com sistemas de maior porte. Faz-se necessário um estudo de caso com um sistema maior, de preferência real. Isto é muito importante para reforçar a aplicação prática das regras e estudar seus impactos no sistema.

Metodologia para aplicar as regras. Durante o estudo de caso, as regras foram aplicadas de acordo com a intuição do desenvolvedor. Seria interessante que houvesse uma metodologia de aplicação de regras que guiasse o desenvolvedor do início ao fim da transformação do programa.

Regras com aspectos. Outro ponto interessante seria a criação de regras que gerenciassem concorrência através de aspectos [CB04], usando a extensão de Java para programação orientada a aspectos - AspectJ [KHH+01].

Avaliar performance. Há a necessidade de se avaliar quais os impactos das aplicações das regras em termos da performance do sistema. Esta avaliação pode influenciar bastante os casos onde se torna realmente interessante aplicar as transformações ou não.

Desenvolvimento de um plugin para o eclipse. Para que as transformações sejam de fato utilizadas, é essencial que haja o suporte de alguma ferramenta de desenvolvimento. Para tanto, O eclipse[1] se mostra uma alternativa muito interessante, pois é bem popular na comunidade de desenvolvedores Java e possui suporte a criação de *plugins* por terceiros.

Provas formais. Para que realmente haja uma maior segurança na a aplicação das regras, é muito importante que elas sejam provadas formalmente. Este trabalho, porém, é bastante árduo, pois envolve complicações como a definição de uma semântica formal para Java (especialmente para as construções destinadas a concorrência) ou o uso de algum formalização existente e as provas em si de cada regra. Caso seja possível usar algum provador de teoremas para tais provas, o trabalho pode ser consideravelmente reduzido, mas caso todas as provas precisem ser feitas manualmente, será muito dispendioso.

Uso de verificadores de modelos. Outra possibilidade no âmbito de assegurar a corretude das regras é o uso de verificadores de modelos (*model checkers*). Esta verificação possibilitaria a detecção de propriedades tais como deadlock, não determinismo, refinamentos e equivalência. Entre os possíveis verificadores de modelos podemos citar o JavaPathFinder [HP00], o Bandera [Dwy02] e o FDR[2] [Gol01] .

[1] `www.eclipse.org`
[2] Mediante tradução para CSP

5.3 CONSIDERAÇÕES FINAIS

Este trabalho é apenas o esforço inicial de um projeto bem mais complexo. Ainda há muitos detalhes envolvidos na aplicação das regras que precisam ser analisados mais a fundo. Uma grande dificuldade encontrada durante a elaboração das regras foi conseguir de fato regras aplicáveis e seguras, pois o modelo de concorrência de Java, apesar de ser considerado um dos melhores entre as linguagens de programação, apresenta muitos detalhes obscuros, que muitas vezes variam entre diferentes implementações da máquina virtual. Assim sendo, apenas com um maior aprofundamento do impacto de todas as regras propostas será possível realizar *refactorings* mais confiáveis com o suporte de ambientes de desenvolvimento.

APÊNDICE A

CÓDIGO FONTE DO ESTUDO DE CASO

Este apêndice mostra na íntegra o código fonte usado no estudo de caso. O código apresentado já possui as regras aplicadas.

A.1 CLASSE BANCO

```
import java.util.Iterator;
import java.util.Vector;

/**
 * @author rafael
 *
 * Classe que representa um banco
 *
 */
public class Banco {

    private Vector<ContaCorrente> contasCorrentes;
    private Vector<Poupanca> poupancas;

    /**
     * Construtor padrão da classe Banco
     */
    public Banco() {
        this.contasCorrentes = new Vector<ContaCorrente>();
        this.poupancas = new Vector<Poupanca>();
    }

    /**
     * Método que adiciona uma nova conta ao banco.
     *
     * @param c
     *            conta corrente a ser adicionada
     */
    public void adicionarContaCorrente(ContaCorrente c) {
        this.contasCorrentes.add(c);
    }

    /**
```

```
     * Método que adicionar uma nova poupança ao banco
     *
     * @param p
     *            poupança a ser adicionada
     */
    public void adicionarPoupanca(Poupanca p) {
        this.poupancas.add(p);
    }

    /**
     * método que remove uma conta corrente do banco
     *
     * @param c
     *            conta corrente a ser removida
     */
    public void removerContaCorrente(ContaCorrente c) {
        this.contasCorrentes.remove(c);
    }

    /**
     * Método que remove uma poupança do banco
     *
     * @param p
     *            poupança a ser removida
     */
    public void removerPoupanca(Poupanca p) {
        this.poupancas.remove(p);
    }

    /**
     * Método que atualiza as contas correntes debitando a taxa de manutenção
     */
    private void atualizarContasCorrentes() {
        synchronized (this.contasCorrentes) {
            Iterator<ContaCorrente> contas = this.contasCorrentes.iterator();
            while (contas.hasNext()) {
                ContaCorrente c = contas.next();
                c.debitarTaxaManutencao();
            }
        }
    }

    /**
     *
```

```
     */
    private void run_atualizarContas() {
        new Thread() {
            public void run() {
                atualizarContasCorrentes();
            }
        }.start();
    }

    /**
     * Método que atuliza as poupanças de acordo com o rendimento
     */
    private void atualizarPoupancas() {
        synchronized (this.poupancas) {
            Iterator<Poupanca> poupancas = this.poupancas.iterator();
            synchronized (Poupanca.class) {
             while (poupancas.hasNext()) {
                    Poupanca p = poupancas.next();
                    p.render();
                }
            }
        }
    }

    /**
     *
     */
    private void run_atualizarPoupancas() {
        new Thread() {
            public void run() {
                atualizarPoupancas();
            }
        }.start();
    }

    /**
     * Método que debita um determinado valor em uma conta
     *
     * @param numeroConta
     * @param valor
     */
    public void debitar(int numeroConta, double valor) {
        for (int i = 0; i < this.contasCorrentes.size(); i++) {
            if (this.contasCorrentes.get(i).getNumero() == numeroConta) {
```

```
                this.contasCorrentes.get(i).debitar(valor);
            }
        }
    }

    /**
     * Método que credita um determinado valor em uma conta
     *
     * @param numeroConta
     * @param valor
     */
    public void creditar(int numeroConta, double valor) {
        for (int i = 0; i < this.contasCorrentes.size(); i++) {
            if (this.contasCorrentes.get(i).getNumero() == numeroConta) {
             this.contasCorrentes.get(i).creditar(valor);
            }
        }
    }

    /**
     * Método que atualiza as contas e poupanças do banco
     */
    public void atualizarBanco() {
        this.run_atualizarContas();
        this.run_atualizarPoupancas();
    }

}
```

A.2 CLASSE CONTA

```
/**
 * @author rafael
 *
 * Classe abstrata que representa contas em geral do Banco
 *
 */
public abstract class Conta {

    private int numero;
    private double saldo;

    /**
     * Construtor da classe Conta
```

```
     *
     * @param numero
     *            numero da conta
     * @param saldo
     *            saldo inicial da conta
     */
    public Conta(int numero, double saldo) {
     super();
     this.numero = numero;
     this.saldo = saldo;
    }

    /**
     * Método que obtém o número da conta
     *
     * @return o número da conta
     */
    public int getNumero() {
        return numero;
    }

    /**
     * Método que obtém o saldo atual da conta
     *
     * @return saldo atual da conta
     */
    public double getSaldo() {
        return saldo;
    }

    /**
     * Método que debita um determinado valor na conta
     *
     * @param valor
     *            valor a ser debitado
     */
    public void debitar(double valor) {
        this.saldo = this.saldo - valor;
    }

    /**
     * Método que credita um determinado valor na conta
     *
     * @param valor
```

```
     *            valor a ser creditado
     */
    public void creditar(double valor) {
        this.saldo = this.saldo + valor;
    }

    /*
     * (non-Javadoc)
     *
     * @see java.lang.Object#equals(java.lang.Object)
     */
    public boolean equals(Object conta) {
        boolean equals = false;
        if (conta instanceof Conta) {
         equals = ((Conta) conta).getNumero() == this.numero;
        }
        return equals;
    }
}
```

A.3 CLASSE POUPANCA

```
/**
 * @author rafael
 *
 * Classe que representa uma Poupança do Banco
 *
 */
public class Poupanca extends Conta {

    private volatile static double rendimento;

    /**
     * Construtor da classe Poupanca
     *
     * @param numero
     *            numero da poupanca
     * @param saldo
     *            saldo inicial da poupanca
     */
    public Poupanca(int numero, double saldo) {
        super(numero, saldo);
        Poupanca.rendimento = 0.1;
    }
```

```
    /**
     * Método que aumenta o saldo da poupança de acordo com o rendimento atual
     */
    public void render() {
        this.creditar(Poupanca.rendimento);
    }

    /**
     * Método que obtém o rendimento atual das poupanças.
     *
     * @return rendimento atual das poupanças
     */
    public static double getRendimento() {
        return rendimento;
    }

    /**
     * Método que modifica o rendimento das poupanças.
     *
     * @param rendimento
     *            novo valor do rendimento
     */
    public static void setRendimento(double rendimento) {
        synchronized (Banco.class) {
         Poupanca.rendimento = rendimento;
        }
    }

}
```

REFERÊNCIAS BIBLIOGRÁFICAS

[And00] G. Andrews. *Foundations of Multithreaded, Parallel and Distributed Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[AS83] G. Andrews and F. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15(1):3–43, 1983.

[Blo01] J. Bloch. *Effective Java programming language guide*. Sun Microsystems, Inc., Mountain View, CA, USA, 2001.

[CB04] L. Cole and P. Borba. Deriving refactorings for AspectJ. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 202–203, New York, NY, USA, 2004. ACM Press.

[Dwy02] M. Dwyer. Software model checking: the bandera approach. In *FMOODS '02: Proceedings of the IFIP TC6/WG6.1 Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems V*, pages 3–4, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification third Edition*. Addison-Wesley, 2005.

[GM96] J. Gosling and H. McGilton. The java language environment: A white paper, 1996. http://java.sun.com/doc/language_environment.

[Gol01] M. Goldsmith. *FDR: User Manual and Tutorial, version 2.77*. Formal Systems (Europe) Ltd, 2001.

[Hoa74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[HP00] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[JSR04] Java specification request 166: Concurrency utilities, 2004. http://www.jcp.org/en/jsr/detail?id=166.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[Lea99] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Lyo98] A. Lyons. UML for Real-Time Overview. Whitepaper, ObjecTime Limited, April 1998.

[Mah05] Q. Mahmoud. Concurrent programming with j2se 5.0, 2005. http://java.sun.com/developer/technicalArticles/J2SE/concurrency/.

[Mic99] Sun Microsystems. Why are thread.stop, thread.suspend, thread.resume and runtime.runfinalizersonexit deprecated?, 1999. http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html.

[Ros97] A. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, 1997.

[SR98] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Whitepaper, Rational Software Corp., March 1998.

[WAF02] P. Welch, J. R. Aldous, and J. Foster. Csp networking for java (jcsp.net). In *ICCS '02: Proceedings of the International Conference on Computational Science-Part II*, pages 695–708, London, UK, 2002. Springer-Verlag.

[Wel04] A. J. Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, 2004.