Controle de Concorrência com Java e Bancos de Dados Relacionais

Sérgio Soares* Paulo Borba†
Centro de Informática
Universidade Federal de Pernambuco

Abstract

The advent of information systems based on the World Wide Web increased the impact of concurrent programs in society. Such increase demands the definition of guidelines for obtaining safe and efficient implementations of concurrent programs, since the complexity of implementation and tests in concurrent environments is bigger than in sequential environments. This paper defines guidelines for concurrency control in object-oriented systems written in Java and using relational databases. In particular, we show how and where the programming language and the relational database features can be used for concurrency control. The main point of the guidelines is to guarantee system correctness without redundant concurrency control, both increasing performance and guaranteeing safety.

Resumo

O advento de sistemas de informação baseados na *World Wide Web* aumentou o impacto dos sistemas concorrentes na sociedade. Tal crescimento torna mais necessário a definição de diretrizes para a obtenção de implementações seguras e eficientes de programas concorrentes, uma vez que a complexidade de implementação e testes em ambientes concorrentes é maior que em ambientes seqüenciais. Este artigo define diretrizes para o controle de concorrência em sistemas orientados a objetos escritos em Java e que utilizam banco de dados relacionais. Em particular, mostramos como e onde os recursos da linguagem de programação de do banco de dados podem ser utilizados para controle de concorrência. O objetivo principal das diretrizes é garantir a corretude do sistema sem redundância no controle de concorrência, tanto aumentando a performance quanto garantindo a segurança do sistema.

1 Introdução

Com o advento de sistemas de informação baseados na *World Wide Web* temos um grande impacto dos programas concorrentes na sociedade. De fato, ambientes concorrentes elevam consideravelmente a complexidade de implementação e testes dos sistemas, uma vez

*Apoiado pela CAPES. Email: `scbs@cin.ufpe.br`.
†Apoiado em parte pelo CNPq, processo 521994/96–9. Email: `phmb@cin.ufpe.br`.

que erros sutis de implementação levam a mau funcionamento de programas que podem ser difíceis de detectar.

Este cenário indica que precisamos de meios mais adequados para a implementação de programas concorrentes, como a definição de diretrizes que dêem suporte ao controle de concorrência nos sistemas, evitando que sejam feitos controles baseados na intuição do programador. Tais controles podem ter um impacto negativo em eficiência, além de não garantir a segurança do sistema com relação ao acesso concorrente ao mesmo. Além disso, controles baseados na intuição também podem introduzir novas situações de corrida ("*race conditions*") que podem gerar execuções inválidas do sistema em ambientes concorrentes. Além de garantir a segurança, a definição das diretrizes documenta e, portanto, padroniza o controle a ser aplicado, favorecendo também a extensibilidade do sistema, uma vez que melhora a manutenibilidade do mesmo.

Atualmente a maioria dos sistemas utiliza sistemas gerenciadores de banco de dados (SGBD) para garantir a persistência dos seus dados. Um SGBD deve garantir uma série de características, como a atomicidade e o isolamento das operações realizadas pelos mesmos [8]. Estas características fazem com que os SGBDs realizem controle de concorrência, de modo que duas operações de acesso aos dados não interfiram entre si mantendo a consistência dos dados. Além disso, os SGBDs fornecem o serviço de transação, que garante a execução atômica de uma seqüência operações. Logo, o SGBD pode ser visto como um mecanismo para tratar parte da concorrência nos sistemas, a concorrência de acesso aos dados armazenado no mesmo.

Neste trabalho utilizamos a linguagem Java [10] que é orientada a objetos e permite programação concorrente, fornecendo recursos para controlar fluxos de execuções concorrentes. Nas definições das diretrizes utilizamos principalmente o modificador de métodos `synchronized`, que impede a execução concorrente dos métodos sincronizados de um mesmo objeto. Outros recursos fornecidos pela linguagem, como os métodos `wait` e `notify`, são utilizados para aumentar a eficiência do controle em determinadas situações.

Com estes dois mecanismos disponíveis (linguagem de programação e SGBD), surge a dúvida de qual destes utilizar para tratar concorrência, ou em que casos cada um deve ser utilizado, de modo a garantir a corretude do sistema sem realizar controles redundantes. Este é exatamente o ponto que procuramos elucidar neste trabalho. De fato, definimos que controles de concorrência devem ser deixados a cargo do SGBD e que controles devem utilizar os recursos da linguagem, indicando qual mecanismo de controle e em que partes do código utilizá–lo. Estas definições foram feitas baseadas em uma arquitetura de software específica. De posse dos resultados apresentados, programadores têm um guia importante para o controle de concorrência, garantindo segurança e eficiência na execução dos sistemas. Atualmente podemos encontrar vários trabalhos que se preocupam em garantir ganhos de performance eliminando sincronizações desnecessárias de programas concorrentes em Java [3, 1, 7]. Nestes trabalhos podemos observar o impacto negativo em eficiência causado pelos mecanismos da linguagem de programação Java, o que mostra a necessidade de guias para termos um controle de concorrência correto e eficiente.

Este artigo está organizado em cinco seções. Na Seção 2 apresentamos uma arquitetura de software e os tipos de classe que implementam a mesma utilizando padrões de projeto [9, 6]. Em seguida, na Seção 3, definimos as diretrizes para controle de concorrência com o intuito de garantir segurança e eficiência de execução, além de identificar quais dos mecanismos (SGBD e linguagem de programação) utilizar para evitar determinadas situações de corrida. Mostramos o impacto em performance de controles diferentes para concorrência na Seção 4, e por fim, na Seção 5, apresentamos as conclusões e relacionamos nosso trabalho com outros.

2 Arquitetura de software e concorrência

As diretrizes apresentadas neste artigo foram definidas para serem utilizadas em sistemas com uma arquitetura de camadas específica e em padrões de projetos [9, 6] que garantem a estruturação do sistema segundo a mesma. Esta arquitetura de camadas visa separar os aspectos de dados, negócio, comunicação (distribuição) e apresentação (interface com usuário), como mostrado na Figura 1. Tal estruturação, evita o entrelaçamento de código com diferentes propósitos, por exemplo, código de negócio e de dados, e juntamente com os padrões de projeto permite alcançar maiores níveis de reuso e extensibilidade.

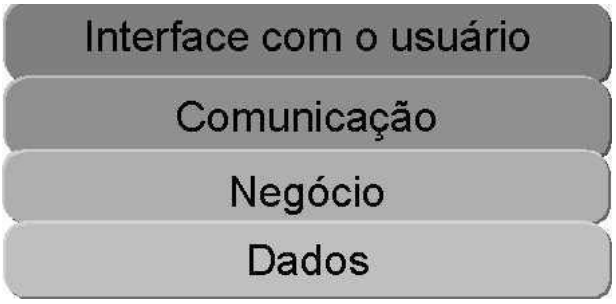

Figura 1: Exemplos de arquitetura em camadas.

Caso os sistemas não tenham esta estruturação lógica, provavelmente haverá um entrelaçamento de código com diferentes propósitos, dificultando o reuso e a extensibilidade. De fato, sistemas sem esta estruturação lógica, podem trabalhar diretamente com tabelas de um banco de dados relacional, sem modelar os dados manipulados pelo mesmo como objetos, apesar de utilizar linguagens de programação orientadas a objetos. Nestes sistemas, o controle de concorrência deve ficar completamente a cargo do SGBD, uma vez que requisições feitas pelo usuário são passadas diretamente ao SGBD, não havendo criação e manipulação de objetos.

Na Figura 2 apresentamos um diagrama de classes de UML [4] que descreve, de maneira geral, como se dispõe as classes da arquitetura de software utilizada na definição das diretrizes para controle de concorrência. Esta arquitetura foi originalmente definida em outro trabalho [18], tendo sido feitas aqui apenas algumas alterações.

Na seção seguinte explicamos o papel de cada tipo de classe mostrada na Figura 2 e definimos as diretrizes para controle de concorrência. Mais detalhes sobre a arquitetura podem ser encontrados em outros trabalhos [16, 2, 13], incluindo variações e restrições aplicadas em determinados tipos de sistemas.

3 Diretrizes para controle de concorrência

Nesta seção indicamos que mecanismos de controle utilizar (linguagem de programação ou SGBD) e em que partes do código cada um deve ser utilizado, de forma a termos um controle seguro, não redundante e eficiente. Para isso definimos diretrizes gerais para o controle, e diretrizes específicas para cada tipo de classe da arquitetura apresentada

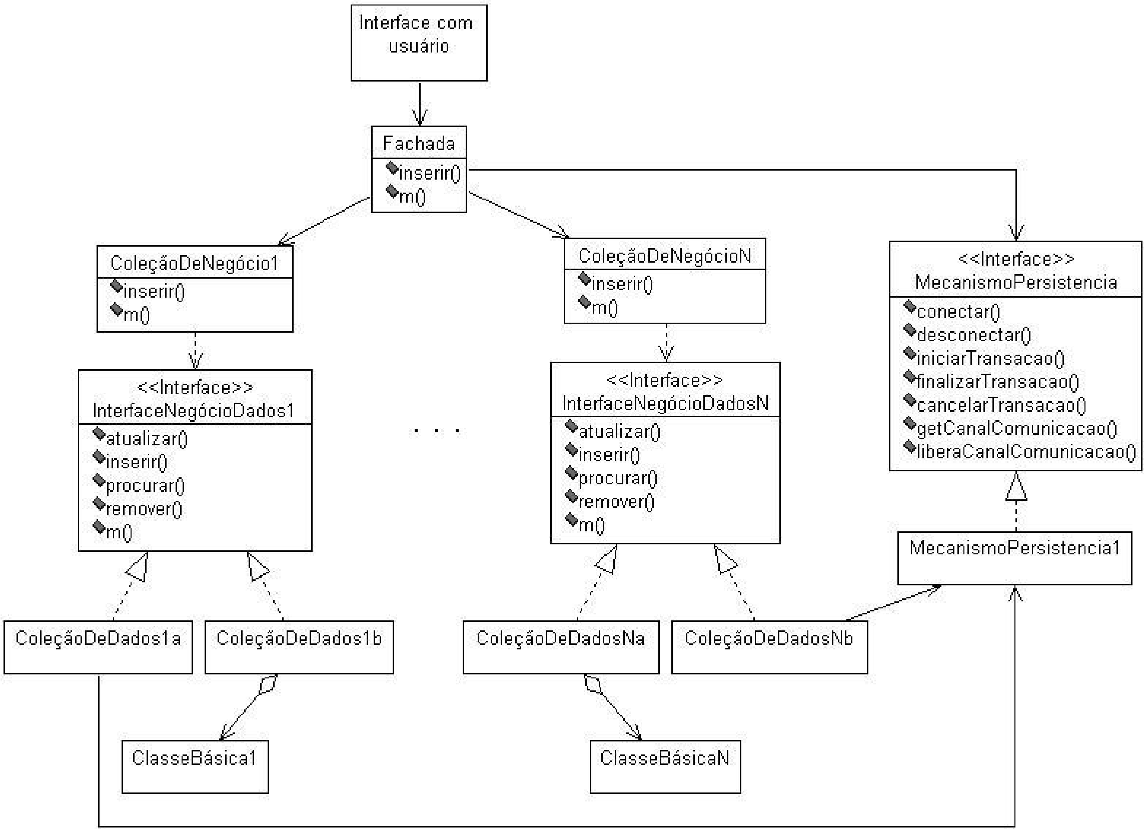

Figura 2: Visão geral do padrão de projeto.

na Seção 2. A definição destas diretrizes evita controles ingênuos como sincronizar todos os métodos de acesso ao sistema (métodos da classe fachada), feito com os recursos da linguagem, ou implementar transações em todos os métodos da classe fachada, feito com recursos do SGBD. No final desta seção apresentamos os controles mais comumente aplicados segundo nossa experiência, adquirida na implementação e na análise de vários sistemas implementados conforme a arquitetura de software utilizada na definição das diretrizes.

3.1 Diretrizes gerais

Na definição das diretrizes gerais procuramos controlar a concorrência sofrida por qualquer um dos tipos de classe ilustrados na Figura 2. Definimos diretrizes que visam impedir a execução concorrente de métodos não atômicos, e de métodos que lêem os atributos alterados pelos métodos não atômicos. Um método não atômico é o método que possui mais de um comando, ou um comando que não é executado atomicamente, atribuições entre variáveis de alguns tipos, por exemplo. Estes métodos podem sofrer interleaving [11, 15], ou seja, execuções concorrente do mesmo podem se entrelaçar entre si, ou entre execuções de outro método. Um exemplo deste tipo de método são os que alteram e os que lêem atributos dos tipos `long` ou `double`, já que a especificação de Java [10] não garante a atomicidade na atribuição a variáveis destes tipos. Desta forma tais métodos deveriam utilizar um recurso da linguagem para impedir sua execução. Java oferece o modificador de métodos `synchronized` que impede a execução concorrente dos métodos sincronizados de um objeto.

Observe que estas diretrizes devem ser aplicadas em classes cujos objetos sofrem acesso concorrente, de modo a impedir a execução concorrente de métodos que não são executados atomicamente e lêem ou alteram atributos da sua classe. Mais exemplos destes tipos de métodos são descritos em outro trabalho [12].

3.2 Diretrizes para Classes Básicas de Negócio

Agora, definimos as diretrizes específicas para as classes básicas de negócio, as quais representam os objetos (básicos) do sistema, como clientes e contas. Inicialmente devemos identificar quais são as classes básicas cujos objetos sofrem acesso concorrente. As coleções de dados têm papel determinante nesta identificação, uma vez que estas classes são as responsáveis pelo armazenamento e recuperação de instâncias das classes básicas de negócio. Existem dois tipos de coleções de dados: as voláteis, que mantêm os objetos em memória, e as persistentes, que armazenam os objetos em meio persistente. Normalmente, nas implementações das coleções persistentes que utilizam um SGBD relacional (SGBDR), é criada uma instância com os dados obtidos do SGBDR ao ser requisitado a recuperação de um determinado objeto. Por exemplo, considere a implementação do método `procurar` da coleção de dados `RepositorioContasBDR` que utiliza a classe `java.sql.ResultSet` [14] (linhas `2`, `7`, e `8`), da API JDBC [19], para recuperar a resposta de um comando SQL (linhas `4`, e `5`) e criar uma instância de `Conta` com os dados obtidos (linhas `7`, e `8`). A classe `ResultSet` representa a tabela de resposta gerada pela execução de um comando `SELECT` no SGBDR.

```
    public class RepositorioContasBDR implements RepositorioContas {
        // ...
        public Conta procurar(String numero) {
 1:         Conta resposta = null;
 2:         ResultSet rs = null;
 3:         try {
 4:             String query = "SELECT * FROM CONTAS " +
 5:                            "WHERE numero='" + numero + "'";
 6:             // ...
 7:             resposta = new Conta(rs.getString("numero"),
 8:                                  rs.getDouble("saldo"));
 9:         }
10:         // ...
11:         return resposta;
        }
    }
```

Observe que a cada execução o método cria uma nova instância de `Conta` (linhas `7` e `8`). Assim, caso dois *threads* requisitem uma consulta a uma mesma `Conta`, os mesmos receberão referências para cópias distintas do objeto armazenado no SGBDR. Desta forma, não há acesso concorrente aos objetos da classe `Conta`.

Porém, implementações alternativas das coleções de dados podem utilizar caches de objetos, para garantir que nunca existirá em memória mais de uma cópia de um objeto armazenado no banco. Esta abordagem de caches é utilizada por APIs de acesso a SGBDs relacionais[1] e orientados a objetos (SGBDOO) [17], o que é feito para evitar inconsistências que podem acontecer, por exemplo, no caso de uma atualização concorrente de cópias de um mesmo objeto. Logo, duas requisições para consultar um mesmo objeto receberão a mesma referência para o objeto armazenado na cache. Desta forma, o objeto poderá ser acessado concorrentemente, o que acontece, por exemplo, quando dois usuários estão usando o sistema ao mesmo tempo e solicitam, via a interface com o usuário, a alteração de um mesmo objeto.

Diretrizes

Tendo identificado que objetos básicos sofrem acesso concorrente devemos então aplicar as diretrizes gerais nas classes destes objetos básicos. Nas classes básicas cujos objetos não sofrem acesso concorrente devemos apenas analisar situações em que não devem ser permitidas atualizações concorrentes de cópias de um mesmo objeto. Por exemplo, considere o mesmo exemplo de aplicação bancária citada anteriormente. A classe `Conta` tem métodos como `creditar` e `debitar`. Na implementação destes métodos o `saldo` é atualizado baseado no valor anterior do mesmo. Em se tratando da atualização concorrente de duas cópias de um mesmo objeto, este tipo de operação pode levar o objeto em questão a um estado inconsistente. Por exemplo, considere uma possível execução onde o `saldo` de cópias de um mesmo objeto da classe `Conta` é alterado concorrentemente executando o método `creditar` e onde o operador || executa dois blocos concorrentemente. Ao falarmos em execução concorrente de dois blocos queremos dizer que as execuções dos mesmos se dá de forma entrelaçada, ou seja, há um *interleaving* [11, 15] na execução dos dois blocos. Vamos adotar a convenção de que a execução do bloco à esquerda do operador || será feita por um *thread* chamado t1 e a execução do bloco à direita por um *thread* chamado t2.

```
                    RepositorioContasBDR rep // ...
Conta c = rep.procurar("2287");   ||   Conta c = rep.procurar("2287");
c.creditar(100);                  ||   c.creditar(50);
rep.atualizar(c);                 ||   rep.atualizar(c);
```

Neste caso, dois *threads* executariam concorrentemente as seguintes operações: procurar por uma mesma `Conta` em uma coleção de dados (que retorna cópias distintas para duas consultas pelo mesmo objeto), creditar um valor na `Conta` retornada pela consulta e atualizar a mesma na coleção de dados. No final desta execução a informação do `saldo` da conta pode estar inconsistente. De fato, esta execução faz com que duas cópias sejam recuperadas e que a primeira cópia da `Conta` que for atualizada seja sobrescrita pela segunda. Desta forma, o valor final do `saldo` será inconsistente, pois ao invés de termos como resultado a atualização do atributo `saldo` da `Conta` com a soma dos créditos (`150`), teremos o `saldo` acrescido de apenas um dos créditos realizados (`100` ou `50`), dependendo da ordem em que as cópias foram atualizadas. Logo, concluímos que tais atualizações devem ser evitadas.

A solução que sugerimos para corrigir problemas em atualizações concorrentes de cópias de objetos é implementar um controle de atualização para cada objeto, o que seria feito no SGBD adicionando um *timestamp* [8] a todos os objeto. Este atributo informa a data e a hora da última atualização do objeto. A atualização de um objeto só é permitida se uma cópia do objeto não tiver sido atualizada mais recentemente. Além

[1] A API de JDBC não implementa caches e é a mais comum atualmente.

disso, o método `atualizar` da coleção de dados, que verifica se existe um objeto mais recentemente atualizado, deve ser sincronizado de modo a garantir a atomicidade da operação. Note que esta solução utiliza tanto um recurso de controle de concorrência da linguagem (na sincronização do método `atualizar`), quanto o SGBD para armazenar os *timestamps*. Este mesmo controle pode ser feito em outras situações onde a atualização concorrente de cópias de um mesmo objeto deva ser evitada. Para identificar tais situações devemos considerar se, no sistema em questão, a atualização concorrente de duas cópias de objetos de uma classe deve ser evitada. Ou seja, devemos identificar se, segundo os requisitos do sistema, esta atualização concorrente é uma execução inválida, devendo ser evitada, ou se é da natureza do sistema. Um exemplo disso é a classe `Pessoa` que possui apenas `nome` e `cpf` como atributos. Dependendo dos requisitos do sistema, a atualização concorrente do atributo `nome` de cópias de uma mesma `Pessoa` pode ser, ou não, uma execução inválida. Em geral, este tipo de situação seria considerada válida, uma vez que estas atualizações que aconteceram concorrentemente poderiam acontecer minutos ou horas uma após a outra, e ainda assim o resultado seria o mesmo, o atributo nome seria atualizado com o valor da última atualização.

3.3 Diretrizes para as Classes Coleção de Dados

Como dito na subseção anterior as coleções de dados são responsáveis pelo armazenamento e recuperação dos objetos básicos do sistema. As coleções de dados possuem uma interface (interface negócio–dados) que abstrai a forma de armazenamento dos dados.

Nas coleções de dados persistentes, também podemos encontrar problemas de concorrência no instante da inclusão, atualização e remoção de um objeto no mecanismo de persistência que o armazena. Nestes casos, devemos garantir que os recursos do SGBD, ou de um outro mecanismo de persistência, são utilizados de maneira correta; assim só devemos incluir, atualizar ou remover um objeto do sistema dentro de uma transação. De fato, quando a coleção de dados em análise trabalha com um SGBD, podemos assumir que grande parte do controle de concorrência, no tocante a atualização dos dados, é feito pelo próprio SGBD. Nos resta apenas garantir que a implementação dos métodos seja atômica no que diz respeito ao acesso ao SGBD. Isto é feito implementando o acesso ao SGBD com apenas um comando SQL, ou implementando uma transação em alguns métodos da fachada, utilizando os serviços da interface mecanismo de persistência. Para garantir a atomicidade dos métodos da coleção de dados devemos seguir os seguintes passos:

- Identificar os métodos das coleções de dados que direta ou indiretamente executam mais de um comando SQL;
- Identificar os métodos das coleções de negócio que chamam os métodos das coleções de dados identificados no passo anterior;
- Identificar que métodos da fachada chamam os métodos das coleções de negócio identificados no passo anterior;
- Os métodos da fachada identificados no último passo devem utilizar os métodos `iniciarTransacao`, `confirmarTransacao` e `cancelarTransacao`, da interface mecanismo de persistência, para implementar uma transação na sua execução.

Um exemplo de como um método pode ser implementado como uma transações pode ser visto no método `atualizar` da classe `Fachada`, onde os trechos de código sublinhados são os responsáveis pela implementação da transação.

```
public class Fachada {
    private CadastroContas contas;
    private MecanismoPersistencia mp;
    // ...
    public void atualizar (Conta conta) {
        try {
            mp.iniciarTransacao();
            contas.atualizar(conta);
            mp.confirmarTransacao();
        }
        catch (TransacaoBDException ex) {
            mp.cancelarTransacao();
        }
    }
}
```

A interface `MecanismoPersistencia` define serviços como o estabelecimento de conexão, implementação de transações e obtenção de um canal de comunicação[2] com o mecanismo de persistência.

Normalmente também existem as coleções de dados voláteis que são utilizadas apenas como respostas pelas consultas das coleções de dados persistentes que retornam mais de um objeto. Não é necessário nenhum controle nestas coleções, uma vez que as mesmas não sofrem acesso concorrente.

3.4 Diretrizes para Classes Coleção de Negócio e Fachada

As classes coleção de negócio são responsáveis pelas verificações e validações segundo a política da aplicação. Tais classes utilizam os serviços das interfaces negócio–dados para armazenar e recuperar os objetos das classes básicas. A classe fachada de negócio [9] provê uma interface unificada para todos os serviços do sistema, centralizando todas as instâncias das classes coleção de negócio. Note que a interface com o usuário deve utilizar o sistema através dos serviços da fachada (Figura 2).

Por fim, definimos as diretrizes específicas para as classes coleção de negócio e fachada. Nestas classes a principal preocupação é identificar regras de negócio que levem a situações de corrida. Um exemplo deste tipo de regra é a verificação, antes de incluir um objeto em uma coleção, se existe algum objeto com o mesmo código do objeto a ser cadastrado. Em uma possível execução concorrente que tenta cadastrar dois objetos com um mesmo código, ambos os *threads* fariam a consulta, receberiam a resposta de que não há nenhum objeto com o mesmo código dos a serem inseridos, e tentariam inserir os objetos. Desta forma, o sistema ficaria em um estado inconsistente, uma vez que seria violada uma regra de negócio, ou seria gerado um erro inesperado. Neste caso específico, pode–se implementar uma geração automática de código para os objetos, por exemplo, utilizando o recurso de *sequence* do SGBDR, ou implementando tal recurso na própria coleção de negócio. Assim esta verificação de negócio não seria mais necessária.

Para outras verificações de negócio que gerem situações como a descrita anteriormente, devemos evitar a execução concorrente, sincronizando os métodos responsáveis pelas mes-

[2] O canal de comunicação da interface mecanismo de persistência é do tipo `java.lang.Object`. Cada implementação da interface deve definir o tipo específico do seu canal de comunicação.

mas, ou utilizando um gerenciador de concorrência [16]. O gerenciador de concorrência mantém um conjunto de `String`, e tem um método `iniciaExecucao`, que recebe como parâmetro um `String` e caso o mesmo já esteja no conjunto bloqueia a execução deste *thread* (que está executando o método). Desta forma, podemos evitar execuções concorrentes sobre um mesmo `String`, por exemplo, um mesmo código de objeto, até que o *thread* que está executando utilizando o `String`, libere o *thread* que está bloqueado. Isto é feito pela execução do método `finalizarExecucao`, avisando que terminou a execução sobre um dado `String`.

Devem ser criadas várias instâncias do gerenciador de concorrência. De fato, em geral há uma instância do gerenciador por objeto das coleções de negócio. Além disso, em uma mesma coleção pode existir mais de uma instância do gerenciador. Isto pode acontecer no caso em que os métodos de uma coleção possam executar concorrentemente entre si, mas não cada um separadamente, onde teríamos um objeto gerenciador por método.

Caso as coleções de negócio, ou a classe fachada tenham algum outro atributo além, respectivamente, da interface–negócio dados e da coleção de negócio, as diretrizes gerais devem ser adicionalmente aplicadas nas mesmas.

3.5 Outras diretrizes

Definimos também diretrizes para a classe mecanismo de persistência, que implementa a interface `MecanismoPersistencia`. Resumidamente, na definição das diretrizes para a classe mecanismo de persistência devem ser aplicadas as diretrizes gerais. Estas diretrizes não são apresentadas pela limitação no tamanho do artigo e para permitir uma apresentação mais didática das demais diretrizes. Além disso, a classe mecanismo de persistência só é alterada ou se cria uma nova caso mude o mecanismo de persistência utilizado. A classe é reusada em vários projetos, portanto suas diretrizes não são aplicadas nos mesmos, daí a escolha por omitir detalhes sobre estas diretrizes.

3.6 Controles mais comumente aplicados

Depois de implementar e analisar vários sistemas implementados segundo a mesma arquitetura de camadas e os mesmos padrões de projetos apresentados aqui, podemos identificar quais controles de concorrência são mais comuns, ou seja, são aplicados com maior freqüência.

Normalmente apenas a implementação de transações é feita na classe fachada e nenhum outro controle é necessário nesta classe. Nas coleções de negócio existem algumas chamadas aos métodos do gerenciador de concorrência para evitar interferências causadas por regras de negócio. A sincronização de métodos é feita apenas nos métodos de atualização das coleções de dados que implementam a técnica de *timestamp*, ou em alguns métodos das coleções de negócio que não utilizam o gerenciador de concorrência caso o sistema mesmo tenha poucos usuários simultâneos acessando o sistema e caso estes métodos tenham pequeno peso computacional, conforme mostramos na seção seguinte. Nas classes básicas, o controle se resume à implementação da técnica de *timestamp*, que adiciona o atributo `timestamp`, e os métodos de acesso ao mesmo, sem nenhum controle adicional, uma vez que na maioria dos sistemas os objetos básicos não sofrem acesso concorrente devido às implementações das coleções de dados persistentes. Esta é a nossa alternativa para controles intuitivos que tendem a sincronizar e implementar transações em todos os métodos da classe fachada.

4 Impacto das diretrizes em performance

Nesta seção relatamos e analisamos testes de eficiência feitos com diferentes diretrizes e abordagens de controles de concorrência, incluindo as estudadas e utilizadas na elaboração das diretrizes definidas neste trabalho. Os resultados destes testes mostram que algumas abordagens não são recomendadas, mostrando a vantagem da abordagem sugerida pelas nossas diretrizes. Além disso, os resultados dos testes auxiliam na decisão de que alternativa usar no controle, uma vez que, em alguns casos, as nossas diretrizes sugerem mais de uma solução.

4.1 Preparação dos testes

Para realizar os testes, implementamos um pequeno sistema de cadastro de clientes, que permite cadastrar, procurar e atualizar objetos. Os testes realizam operações de inclusão, consulta, e alteração feitas em diferentes versões do sistema, cada uma com um dos diferentes controles de concorrência analisados. Realizamos três comparações. A primeira compara os seguintes controles:

- Nenhum controle: Sistema sem nenhum controle de concorrência;
- Fachada sincronizada: Todos os métodos da classe fachada sincronizados;
- Fachada com transações: Todos os métodos da classe fachada implementando transações;
- Controle sugerido: Aplicando as diretrizes definidas neste trabalho, com gerenciador de concorrência na coleção de negócio e *timestamp* para a classe `Pessoa`.

Dos quais apenas a abordagem de sincronizar toda a fachada e a que aplica as diretrizes que definimos garantem a corretude dos sistemas se aplicadas isoladamente. A abordagem que não realiza nenhum controle é utilizada para servir de referência na comparação com os demais controles. A segunda comparação procura medir o impacto da técnica de *timestamp* onde comparamos as seguintes versões do sistema:

- Nenhum controle: Sistema sem nenhum controle de concorrência;
- *Timestamp*: Técnica de *timestamp* [16] implementado para a classe básica `Pessoa`;

Por fim, analisamos a alternativa que definimos para a sincronização de métodos comparando as seguintes versões do sistema:

- Sincronização nas Coleções de Negócio: Método `cadastrar` da coleção de negócio sincronizado;
- Gerenciador de Concorrência nas Coleções de Negócio: Método `cadastrar` da coleção de negócio utilizando a classe `GerenciadorConcorrencia` [16];

Para cada um dos diferentes tipos de controle, analisamos também variações nos métodos que receberam tal controle. Assim, pudemos aferir o impacto que diferentes tipos de sistema (representados pelas variações utilizadas) têm em diferentes controles. O aumento do tempo de execução, "peso computacional", de um método foi implementado através da inclusão de um *loop* nos métodos tratados, aumentando o tempo de execução dos métodos originais em aproximadamente 100%.

Outra variação nos testes realizados foi um aumento na carga do sistema. Realizamos testes com várias cargas entre 3 e 600 *threads* acessando o sistema concorrentemente. Com isto pudemos simular uma situação de concorrência extrema, dificilmente encontrada nos picos de acesso em sistemas reais. Por exemplo, dados coletados por um sistema *web* de médio porte, com uma taxa razoável de acesso, têm picos de acesso que não ultrapassam 400 usuários por hora (os tempos de execução dos nossos experimentos são de alguns segundos).

Nossos testes foram realizados utilizando o SGBDR Oracle 8.04, com 12 canais de comunicação disponíveis na classe mecanismo de persistência. A máquina que executa o SGBDR é um Power RS6000 39H (IBM) com 512 MB de memória RAM, rodando o sistema operacional Unix AIX 4.2. A máquina utilizada para executar o sistema teste e a criação de *threads* simulando o acesso concorrente ao sistema é um PC AMD K6-2 500 MHz, com 128 MB de memória RAM, executando o Windows NT 4.0. Foi utilizado o JDK–1.2–V para compilar e executar os testes.

4.2 Análise dos testes de performance

A seguir resumimos os testes de eficiência realizados com as diferentes variações como carga do sistema e "peso" de execução dos métodos.

Abordagens gerais para controle de concorrência

A Figura 3 apresenta um gráfico de barra que mostra o impacto, em relação ao sistema sem nenhum controle, de sincronizar todos os métodos da fachada, implementar todos os métodos da fachada como transações, e aplicar as diretrizes sugeridas neste artigo.

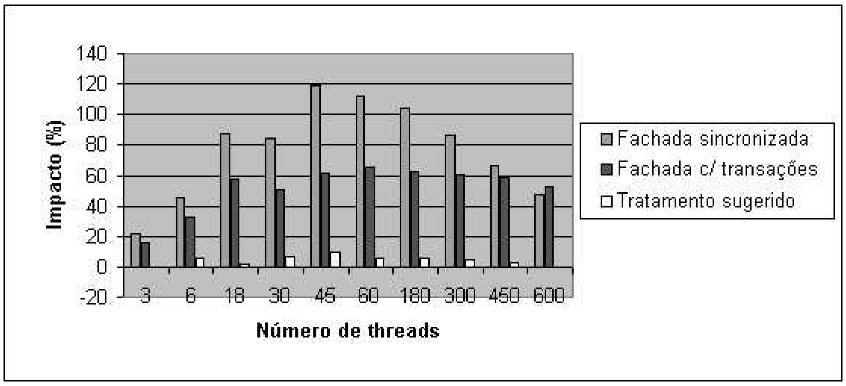

Figura 3: Impacto de diferentes controles de concorrência.

Olhando o gráfico, o controle que sincroniza todos os métodos da fachada é bem caro, chegando a aumentar o tempo de execução em cerca de 120%. Podemos ainda perceber um *overhead* significativo, mais de 50%, na implementação do conceito de transações em todos os métodos da fachada. Isto nos motiva a aplicar as nossas diretrizes para controle de concorrência, as quais indicam, além de outros controles, exatamente que métodos da fachada devem implementar transações. Em relação ao impacto causado pela aplicação das diretrizes definidas neste trabalho, podemos concluir que o impacto é pequeno (menos de 10%), se comparado com as demais abordagens. Todavia, este é um impacto necessário para garantir a segurança e o bom funcionamento do sistema.

Além disto, nos tempos de execução de métodos pesados observamos que houve uma diminuição no impacto dos controles em até mais de 50%, o que nos leva a concluir que quanto mais pesados forem os métodos de um sistema, menor será o impacto do controle no tempo de execução do mesmo.

Timestamp

Similar ao que fizemos na comparação anterior, podemos observar na Figura 4 o impacto de aplicar a técnica de *timestamp* comparado ao sistema sem nenhum controle. O impacto é pequeno, menos de 10%, no caso de métodos leves, mas em métodos mais pesados o impacto é maior, variando de 10% até quase 35%. Isto ocorre devido ao número de canais de comunicação que é um fator limitante, neste caso; como os métodos da coleção de dados são mais pesados, eles demoram mais a executar. Com isto eles acabam mantendo o canal de comunicação por um tempo maior, impedindo que outros *threads* o utilizem. Nestes testes tínhamos 12 canais de comunicação disponíveis no mecanismo de persistência. Novamente, apesar de observarmos um impacto considerável no caso de métodos pesados, este é um controle necessário para garantir a corretude do sistema.

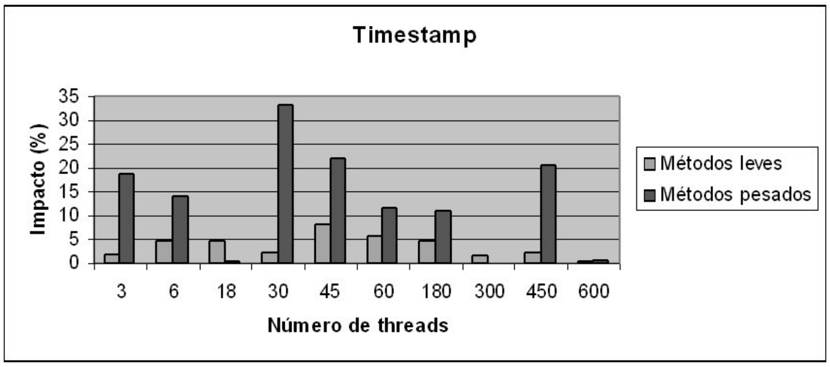

Figura 4: Impacto do *timestamp*.

Gerenciador de Concorrência

A última comparação feita foi entre a sincronização de métodos da coleção de negócio e a utilização de chamadas ao gerenciador de concorrência nestes métodos. Consideramos mais uma variável na realização destes testes. Além da carga do sistema e do "peso" dos métodos, fizemos testes onde ocorrem situações de corrida com relação aos dados, para que as soluções sejam avaliadas segundo este aspecto. Nos outros testes não existia a concorrência em relação aos dados, ou seja, não eram inseridos, atualizados ou consultados objetos com o mesmo código concorrentemente. Fizemos mais esta variação pois a mesma poderia sugerir a utilização de diferentes alternativas para cada caso, uma vez que esta semântica dos métodos é utilizada pelo gerenciador de concorrência para realizar a sincronização quando necessário. Esta variação foi implementada criando conjuntos *threads* que tentam inserir, atualizar e consultar um mesmo objeto.

A Figura 5 mostra o impacto de utilizar o modificador de métodos `synchronized`, em detrimento do gerenciador de concorrência. Podemos observar que a sincronização dos métodos com o modificador `synchronized` na coleção de negócio é de 20% a 40% pior que a utilização do gerenciador de concorrência, a não ser no caso com poucos usuários acessando o sistema concorrentemente, onde os controles têm basicamente o mesmo impacto. Nos casos em que as duas soluções têm sempre o mesmo desempenho, ou seja, sistemas que só são acessados por menos de 6 usuários simultaneamente, sugerimos a adoção da solução que utiliza o modificador `synchronized`, uma vez que é a mais simples de implementar e manter. Nos testes com métodos pesados o impacto do controle com o modificador `synchronized` é menor, mas ainda assim é bem relevante ficando ente 10% e 20%.

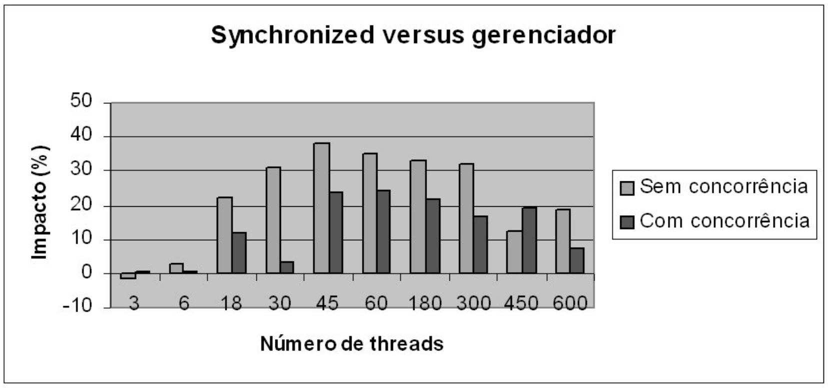

Figura 5: Impacto do `synchronized` em relação ao gerenciador de concorrência.

5 Conclusões

A implementação de sistemas concorrentes é difícil devido a grande complexidade inserida pelo não–determinismo inerente a este tipo de ambiente. Ao implementar um sistema que deve executar em ambiente concorrente devemos controlar a concorrência sofrida pelo mesmo, de modo a evitar que interferências indesejadas levem o sistema a um estado inconsistente. Um problema ao realizar o controle de concorrência é identificar o melhor local para realizar determinados controles e que mecanismos utilizar em cada caso. Neste trabalho estamos considerando sistemas que utilizam banco de dados relacionais e a linguagem de programação Java. Certamente os sistemas podem utilizar as características dos SGBDs para deixar parte do controle a cargo dos mesmos. A outra parte do controle deve ser feita com recursos da linguagem de programação. Mas ainda existe a dúvida de quais destes mecanismos utilizar e em que casos devem ser utilizados cada um de modo a evitar controles redundantes. Outra questão é em que ponto do código realizar o controle de modo a obter uma melhor eficiência na sua implementação. Por isto é de supra importância a definição de diretrizes que dêem suporte na implementação de sistemas concorrentes fornecendo guias para a identificação de que pontos controlar a concorrência e qual mecanismo utilizar em cada caso. Sem dúvida, qualquer controle que não utilize um padrão documentado é passível de erros. A definição de que parte do controle deve ser feita pelo SGBD e que parte deve ser feita pela linguagem evita redundâncias, garantindo eficiência de execução do sistema, uma maior extensibilidade e mais produtividade no desenvolvimento. Isto sem falar dos problemas de extensibilidade e reuso que podem ser ocasionados pela falta de um padrão para a implementação. Certamente, a contribuição maior deste artigo é a definição das diretrizes para o controle de concorrência em sistemas implementados em Java com base na arquitetura e nos padrões de projetos sugeridos pelo Pim [5] e que utilizem SGBDs relacionais, garantindo a corretude dos sistemas com o uso eficiente dos controles.

Em geral, o SGBD resolve boa parte do controle de concorrência, eliminando a necessidade do uso de recursos da linguagem em vários lugares, como nas classes básicas, onde só deve ter controle por parte da linguagem caso seus objetos sofram acesso concorrente, o que segundo nossa experiência não acontece na maioria dos sistemas. Mas os SGBDs não resolvem todos os problemas relacionados a controle, logo, precisamos saber em que pontos é necessário o uso dos recursos da linguagem de modo a evitar controles redundantes e, portanto o impacto negativo em performance. Resolvemos tais problemas através da definição de diretrizes de controle, evitando perdas de 10% a 110% quando comparadas com soluções ingênuas como sincronizar os métodos da classe fachada. As diretrizes foram definidas para controlar concorrência de maneira progressiva, ou seja, após a implementação dos aspectos funcionais, de modo permitir que o programador não se preocupe, inicialmente, com aspectos de controle de concorrência. As diretrizes podem tanto ser utilizadas para analisar sistemas onde já foi implementado o controle, de modo a validar e corrigir, caso necessário, o controle aplicado, quanto para tratar a concorrência de maneira não progressiva, ou seja, ao mesmo tempo em que os requisitos funcionais do sistema são implementados.

Neste artigo analisamos algumas alternativas para solucionar problemas de concorrência e mostramos que em geral o gerenciador de concorrência se mostrou mais eficiente que a utilização do modificador `synchronized`. Além disso, constatamos o impacto da utilização desordenada do modificador de métodos `synchronized`, bem como da implementação desnecessária de transações nos métodos da classe fachada. Os experimentos também nos permitiram afirmar que o impacto da aplicação das diretrizes em um sistema é relativamente pequeno, apesar de ser um impacto necessário. Outra vantagem da nossa proposta é que a mesma é baseada em uma arquitetura de software específica, o que nos permite uma definição e uma aplicação mais exata das diretrizes dando mais suporte ao programador. Apesar de ser específica a arquitetura utilizada serve para a implementação de inúmeros sistemas. A produtividade é favorecida pelo fato das diretrizes irem direto ao ponto, identificando em que classes e de que maneira podem surgir situações passíveis de controle, além, é claro, de definir que mecanismo de controle e onde utilizá–los em cada caso.

Um trabalho relacionado é *Concurrent Programming in Java* [12] onde são propostos modelos para a implementação de programas concorrentes em Java, são descritos padrões de projeto para organizar classes de maneira a garantir que o sistema seja executado com segurança em ambientes concorrentes e são definidas algumas regras para inserir e retirar a sincronização de métodos ou trechos de código. A abordagem descrita no trabalho sugere que o controle deve ser aplicado durante a implementação dos requisitos funcionais do sistema, o que eleva a complexidade da implementação por fazer com que o programador se preocupe desde o início com o controle de concorrência. Apesar de não ser o nosso objetivo, as nossas diretrizes também podem ser utilizadas com este propósito. Porém, o diferencial do nosso trabalho é o fato de nos basearmos em uma arquitetura de software específica, o que facilita a definição de diretrizes mais precisas, permitindo inclusive um alto grau de automatização. Outro diferencial é a definição de regras e padrões para tratar a concorrência em sistemas que utilizam SGBDs relacionais, permitindo utilizar os recursos para controle já fornecidos pelos mesmos.

Resumo das diretrizes

Gerais	
1	Impedir execuções concorrentes de métodos não atômicos
1.1	Métodos que alteram e lêem atributos dos tipos `double` e `long`
1.2	Métodos que utilizam valores dos atributos na realização das operações
1.3	Métodos que realizam atribuições a mais de um atributo
Classes Básicas de Negócio	
1	Identificar classes básicas que sofrem acesso concorrente
2	Aplicar diretrizes gerais nas que sofrem acesso concorrente
3	Evitar atualizações concorrentes de cópias de objetos (se necessário)
Classes Coleções de Dados	
1	Identificar coleções de dados que utilizam um SGBD
2	Garantir que operações que modificam o SGBD sejam atômicas
2.1	Utilizar apenas um comando SQL por método, ou
2.2	Implementar transações na classe fachada
3	Controlar a concorrência sofrida pelas coleções que não utilizam um SGBD
3.1	Impedir acesso concorrente a estruturas de dados em memória, ou
3.2	Utilizar estruturas de dados que suportem acesso concorrente *thread safe*
3.3	Impedir acesso concorrente a outros meios de persistência, e
3.4	Implementar o serviço de transações para outros meios de persistência
Classes Coleções de Negócio e Fachada	
1	Sincronizar métodos com regras de negócio que geram condições de corrida
2	Utilizar o gerenciador de concorrência aumentando a eficiência
3	Implementar/utilizar a geração automática de código (*sequence*)
Mecanismo de Persistência	
1	Impedir acesso concorrente a estruturas de dados em memória, ou
2	Utilizar estruturas *thread safe*

Agradecimentos

Este trabalho foi parcialmente financiado pela CAPES e pelo CNPq, agências financiadoras de pesquisas brasileiras.

Referências

[1] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 207–222. ACM, November 1999.

[2] Vander Alves. Progressive development of distributed object-oriented applications. Dissertação de Mestrado, Centro de Informática – Universidade Federal de Pernambuco, Fevereiro 2001.

[3] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in java. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 35–46. ACM, November 1999.

[4] Grady Booch, Ivar Jacobson, and James Rumbaugh. *Unified Modeling Language – User's Guide*. Addison–Wesley, 1999.

[5] Paulo Borba, Saulo Araújo, Hednilson Bezerra, Marconi Lima, and Sérgio Soares. Progressive Implementation of Distributed Java Applications. In *Engineering Distributed Objects Workshop, ACM International Conference on Software Engineering*, pages 40–47, Los Angeles, EUA, 17th–18th May 1999.

[6] F. Busschmann, R. Meunier, H. Robert, P. Sommerlad, and M. Stal. *A System of Patterns: Pattern–Oriented Software Architecture*. John Wiley & Sons, 1996.

[7] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19. ACM, November 1999.

[8] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison–Wesley, second edition, 1994.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1994.

[10] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison–Wesley, second edition, 2000.

[11] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[12] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, second edition, 1999.

[13] Tiago Massoni. Um Processo de Software com Suporte para Implementação Progressiva. Dissertação de Mestrado, Centro de Informática – Universidade Federal de Pernambuco, Fevereiro 2001.

[14] Sun Microsystems. Package java.sql. Disponível em http://java.sun.com/products/jdk/1.2/docs/api/java/sql/package-summary.html.

[15] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[16] Sérgio Soares. Desenvolvimento Progressivo de Programas Concorrentes Orientados a Objetos. Dissertação de Mestrado, Centro de Informática – Universidade Federal de Pernambuco, Fevereiro 2001. Disponível em http://www.cin.ufpe.br/~scbs/syscoop.

[17] Ardent Software. O2 technology user manual: Java relational binding. Version 2.0, July 1997.

[18] Euricelia Viana and Paulo Borba. Integrando Java com Bancos de Dados Relacionais. In *III Simpósio Brasileiro de Linguagens de Programação*, pages 77–91, Porto Alegre, Brasil, 5–7 de Maio 1999.

[19] Seth White and Mark Hapner. JDBC 2.1 API. Version 1.1. Sun Microsystems, October 1999.